JUNIPER NETWORKS

Distinguished Engineering Series

HARDWARE-DEFINED NETWORKING

MODERN NETWORKING FROM A HARDWARE PERSPECTIVE

Foundation Principles
Tunnels
Network Virtualization
Terminology
Forwarding Protocols
Load Balancing
Overlay Protocols
Virtual Private Networks
Multicast
Connections
Quality of Service
Time Synchronization
OAM
Security
Searching
Firewall Filters
Routing Protocols
Forwarding System Architecture

This hardware perspective of networking delivers a common framework for software developers, dev ops, automation programmers, and all the various networking engineers to understand how modern networks are built.

By Brian Petersen

HARDWARE-DEFINED NETWORKING

MODERN NETWORKING FROM A HARDWARE PERSPECTIVE

Hardware-Defined Networking (HDN) explores the patterns that are common to modern networking protocols and provides a framework for understanding the work that networking hardware performs on a packet-by-packet basis billions of times per second.

These patterns are not revealed in the command line interfaces that are the daily tools of IT professionals. The architects and protocol designers of the Internet and other large-scale networks understand these patterns, but they are not expressed in the standards documents that form the foundations of the networks that we all depend upon.

HDN presents these essential networking patterns and describes their impact on hardware architectures, resulting in a framework that software developers, dev ops, automation programmers, and all the various networking engineers can understand how modern networks are built.

Most networking books are written from a network administrator's perspective (how to build and manage a network), while many new networking books are now written from a software perspective (how to implement a network's management plane in software); *HDN*'s perspective will benefit both the hardware *and* the software engineers who need to understand the tradeoffs of design choices.

"Today, massive compute problems such as machine learning are being tackled by specialized chips (GPUs, TPUs). So, how will specialized hardware handle the massive bandwidths from IoT devices to Mega-Scale Data Centers and equally massive bandwidths from those MSDCs to hand-helds? Here is just the book to find out: every time I open it I learn something new, something I didn't know. Brian Petersen has taken a thoroughly modern snapshot of how it all comes together ."

Dr. Kireeti Kompella, SVP and CTO Engineering, Juniper Networks

"Brian Petersen has accomplished something quite remarkable with this book; he has distilled complex and seemingly disparate networking protocols and concepts into an eminently understandable framework. This book serves as both an excellent reference and as a learning tool for individuals from a broad range of networking disciplines."

Jean-Marc Frailong, Chief Architect, Juniper Networks

ISBN 978-1-941441-51-0

54000

9 781941 441510

Juniper Networks Books are singularly focused on network productivity and efficiency. Peruse the complete library at www.juniper.net/books.

JUNIPer NETWORKS

Hardware-Defined Networking

Modern Networking from a Hardware Perspective

by Brian Petersen

JUNIPER NETWORKS

Published by Juniper Networks Books
Written and Illustrated by: Brian Petersen
Editors: Patrick Ames, Nancy Koerbel

ISBN: 978-1-941441-51-0 (print)
Printed in the USA by Vervante Corporation.

ISBN: 978-1-941441-52-7 (ebook)

Version History: v1, August 2017
2 3 4 5 6 7 8 9 10

http://www.juniper.net/books

About the Author

Brian Petersen's engineering career largely mirrors the growth and progress in networking. After exploring a variety of disciplines, Brian joined 3Com Corporation back when Ethernet's most formidable competitor was "SneakerNet"— floppy discs. From there, Brian did pioneering work on high-density 100 Mbps Ethernet bridges at Grand Junction Networks and, after its acquisition, at Cisco Systems. The volatile early 2000s led to a series of startups (notably Greenfield Networks and TeraBlaze), culminating in several years at Broadcom Corporation and, since 2010, as a Distinguished Engineer at Juniper Networks. From building Ethernet MACs using discrete logic elements to developing packet processing architectures for multi-terabit packet forwarding engines intended for chassis-scale systems, Brian has developed a deep and rich understanding of network architectures and the packet processing required to support them.

1 Preface

Most books about networking have been written for the designers and operators of networks themselves. Another sizable fraction focus on the protocols used by networking equipment to distribute state, routing, and quality of service information from system to system and network to network. This book is focused on what's missing from this body of work: clear and concise descriptions of networking theories, operations, protocols, and practices from the perspective of the hardware that does the work of actually forwarding all of those packets.

The information in this book is gleaned from hundreds of standards specifications as well as decades of practical experience designing and building networking silicon and systems. But this book is not just a summary of standards documents. Standards documents generally suffer from a number of shortcomings. First, there seems to be two diametrically opposed opinions in the standards-writing community regarding the tone, context and coverage of the standards documents. Some standards go so far out of their way to avoid offering anything that seems like helpful advice or common-sense descriptions—the practical implications of the algorithms, structures and rules are deeply buried in bureaucratic obfuscation—that one can feel as though they should have an attorney on retainer while reading those documents. Meanwhile, other standards gloss over their material in a casual, off-hand way that leaves one wondering if something important was accidentally omitted or if their authors are relying on several follow-up documents to fill in the gaps.

Second, the various standards bodies can't seem to agree on terminology, style and even something as basic as the numbering of bits in a word. Bytes vs. octets. Big-endian vs. little-endian. Packets vs. frames vs. datagrams. It's almost as if the standards from one organization are not intended to interact in any way from those of another organization.

Finally, the use of jargon, abbreviations, acronyms and initialisms is so rampant that, unless you're the inventor of the jargon or have been living with it for an extended period of time, actually reading and making sense of a typical standards document is an extraordinarily labor intensive and frustrating exercise. The terminology problems are compounded through the inconsistent use of terms from standard to standard—both between standards bodies and within a single standards body. In this book I've expanded acronyms, reduced jargon and tried to normalize terms and presentation styles as much as possible without straying so far from the source material as to make it unrecognizable.

Ultimately, my goal in writing this book is to present commonly-used protocols in a consistent, readily-understandable manner, with background, history, and examples, providing a framework that can be used to organize and understand the arcane details of the protocols covered in this book, and to provide a mental model for facilitating your understanding of new protocols that are bound to be invented in the coming years.

Brian Petersen, Distinguished Engineer, Juniper Networks
August 2017

2 Introduction

The life blood of all modern societies and economies is movement. Movement of people, capital, raw materials and finished goods. Movement of food, energy, water and other commodities. The movement of all of those items is supported by the parallel, reciprocal and orthogonal movement of information. Without the movement of information, all of those other systems of movement would immediately grind to a halt. Maps, routes, itineraries, permissions, requests, bids, orders, specifications, invoices, payments and many more forms of information underlie the movement of all physically tangible items.

Communications—the movement of information from one place to another, from one brain to another, from one time to another—is fundamental to the human experience and is, indeed, essential for life itself. Biological systems use DNA to communicate vast amounts of information from one generation to the next (fortunately for us, slightly imperfectly). Human spoken, and later, written languages permit the transmission of thoughts and ideas across great distances and, with the development of storage systems—e.g., impressions on clay tablets, ink on (or holes in) paper, magnetic charges, captured electrons, etc.—across vast stretches of time.

Network latency—i.e., the delay between the original transmission of a message and its final reception—used to be dependent upon the speed of some animal or another: human, horse, homing pigeon, etc., or upon the speed of a machine: sailing ship, steam ship, steam train, etc. With the invention of the electric telegraph early in the 19th century, the speed at which information could travel leapt from about 80–140 kilometers per hour (50–90 miles per hour) for a homing pigeon, to 0.5-0.9c (~30,000,000–60,000,000 miles per hour) for an electrical signal flowing down copper wires. For the first time in human history, near real time information could be simultaneously gathered from numerous points across great distances, enabling weather mapping, battlefield intelligence, financial market reports and much more.

Since solving the communications latency problem nearly 200 years ago—jumping immediately from days, weeks or months to near zero—we've made exponential progress on the bandwidth supported by our networks. A good telegraph operator could send about 15 bits per second (bps) while today's optical networks are pushing 1 trillion bits per second (Tbps).[1]

[1] In the not too distant future, 1 Tbps will seem quaint.

The topology of our electronic (or optical) communications networks have also evolved over the years. These networks have gone from the simple point-to-point of early telegraph and telephone networks, to manually operated central switching offices, to automatic central switching offices (e.g., rotary dial, then touch-tone phones), to digital telephony with automatic signaling and call management, to circuit-switched telephony networks, and, finally, to packet-switched networks, the ultimate expression of which is the world-wide Internet.

With the rise of the digital computer in the latter half of the 20th century came a rising awareness of the value of interconnecting computers via networks in order to share data and software and to use the computers and their networks for a variety of forms of communication. Much like the Cambrian explosion 542,000,000 years ago in the evolution of life, a lot of experimental work in the 1970s and 80s led to a vast diversity of packet forwarding methods. Examples of this time include IPX (Xerox), AppleTalk (Apple), SNA (IBM), XNS (Xerox, again) and DECnet (DEC). Essentially, every computer manufacturer developed their own networking protocol in order to connect their own computers with one another.

This diversity of protocols was the main impediment to the development of hardware-based packet forwarding engines. Indeed, the term "multi-protocol" was synonymous with being a router. Hence, all routers up until the mid 1990s were based on general-purpose CPUs. But, with the introduction of the Internet and web browsers to the masses, it became clear that IPv4 was going to be the dominant protocol. This sudden narrowing of focus would obviate the need for general-purpose CPUs and enable purely hardware-based forwarding planes.

With Ethernet dominating media access control and IPv4/TCP dominating internetwork forwarding, life was good, easy, simple and sensible in the networking hardware world. That didn't last long, though. MPLS came along because we became convinced that IPv4 lookups were too hard. Then we started to run out of IPv4 addresses, so IPv6 was born. Then we started building private networks on top of public networks, so a diversity of tunnel encapsulations were born. Now, these protocols are being used to build globe-spanning networks, massive data center networks, enterprise networks and highly mobile networks.

While the diversity of protocol and header types is not nearly what it was during the early days of computer networking, the diversity of ways in which those protocols and header types are being arranged in packets and interpreted by forwarding systems has never been more complex. Compounding this complexity is the operating software that runs in and manages each of the forwarding systems used to build large and complex networks.

In recent years, the concept of software-defined networking has swept through the networking industry, affecting the planning and thinking of network operators and networking equipment vendors alike. Software-defined networking—in its Platonically ideal state—allows centralized controllers to update the operating state of a diversity of hardware platforms through a set of common APIs (application programming interfaces). As of this writing, a lot of energy has been expended toward this goal, and some real progress has been made. Ultimately, we may see networks built from heterogeneous hardware that is all collectively managed by sophisticated, automated controllers that neatly abstract away the nitty-gritty details of the underlying hardware-based networking infrastructure.

However, regardless of how sophisticated and complete this controller software eventually becomes, networks will still be built using hardware that implements those details and dutifully examines each and every packet to ensure that the intent of the controlling software is carried out. Ultimately, it is the capabilities of the underlying hardware that defines what a software-based controller can do to manage a network. Want to use a particular forwarding protocol? Want to terminate a series of tunnels while propagating congestion notifications? Want to search into a forwarding database using a particular assortment of header fields and metadata? You'll need hardware that supports those operations.

There is no getting around the fact that networking hardware is necessarily complex. Fortunately, underlying this complexity and amid the thousands of nitty-gritty details, there is a fundamental logic and, dare I say, beauty to it all. A lot of those nitty-gritty details are, by necessity, presented here in this book, but conveying the logic and structure of networking is this book's true goal. To that end, the very next chapters hold off from presenting the details of various protocols and, instead, bring this logic and structure into focus. As you work your way through the detail-laden chapters, I encourage you to refer back to the first few introductory chapters to help you organize those details within your growing understanding of the logic and structure of networking and the hardware that gives it life.

3 Foundation Principles

In this chapter, we'll build the conceptual foundation upon which all of networking hardware is built.

Bridges and Routers and Switches. Oh, My!

A lot of ink, toner, pixels and hot air has been expended over the years debating the exact definitions of bridges vs. routers vs. switches. In reality, the differences are minor and the forced distinctions just add confusion. To be clear, bridges and routers and switches all receive packets (or frames, if you prefer) via some kind of interface and then forward them to zero, one or more other interfaces where they're then transmitted toward their intended destinations. The exact details of the forwarding methods and rules vary depending on the types of packets being forwarded, but the essentials are the same.

Now, that being said, bridges are generally associated with Ethernet packets while routers are associated with IPv4, IPv6 and MPLS. Even though the forwarding methods of IP and MPLS are as different from one another as IP is from Ethernet—and MPLS even has the word "switching" in its name—IP and MPLS are both forwarded by what we call routers. The only thing that IP and MPLS have in common is the presence of a `timeToLive` field in their headers. So, if it's helpful to think that `router == timeToLive`, then that'll work just fine.

Where it is necessary or convenient to refer to bridge, switch, and router functions interchangeably, the term "forwarding entity" is used. When a collection of bridges and/or routers are assembled within a hardware system, the term "forwarding system" is used.

Layers Upon Layers

Once upon a time, international standards bodies endeavored to bring order and structure to the free-for-all world of networking. They did this by publishing the Open System Interconnection (OSI) network layer model. The layers they came up with are:

1. Physical
2. Data link
3. Network
4. Transport
5. Session
6. Presentation
7. Application

The central premise of the OSI network layer model is that lower-numbered layers present parcels of information to their higher-numbered neighbors while the reciprocal relationship is about layers using the services of their lower-numbered neighbors, all across well-defined logical interfaces.

Back in the 1970s, this wasn't such a bad model. But then the world changed and we've been forcing things into these layers with no real benefit and much real confusion. For example, the data link layer really refers to a single point-to-point connection across a dedicated or shared physical medium. The canonical Layer 2 network, Ethernet, started life as a shared-medium network: a single coax cable snaking from node to node. Every Ethernet packet transmitted onto the coax cable could be received by every node attached to the cable. It was, literally, a broadcast-only network. To ensure that packets got where they needed to go, a globally-unique 48-bit media access control (MAC) address is statically assigned to every node and is carried in a destination address field in every Ethernet packet. Each Ethernet adapter (a network node's connection to the Ethernet network) was trusted to receive and accept only those packets whose destination address matched the address of the node.

This very simple way of building networks did not scale very well, so the transparent bridge was invented. This simple forwarding entity was used to split these shared-media networks into separate segments and to only forward from one segment to another those packets whose destinations were on the far side of these two-port systems. All of a sudden, forwarding decisions were being made at Layer 2. This was supposed to be the job of Layer 3. Yikes! Getting confusing already.

Years later, convinced that longest-prefix matches on 32-bit IPv4 addresses were too difficult to perform at high speeds, label switching was invented. The premise was that it was far simpler to just use a relatively narrow 20-bit value (i.e., a label) as an index into a million-entry table (2^{20} = 1M) to determine how to forward a packet, and multi-protocol label switching (MPLS) was born. Despite the presence of the word "switching" in its name, we have MPLS routers. Go figure. MPLS does have a `timeToLive` field like IP, but I guess that "switching" sounded simpler, faster and sexier than routing at the time, so here we are. Okay, let's call it a routed protocol, just as IP is a routed protocol. Both being routed protocols means that both want to live at Layer 3. Oops! Two protocols in simultaneous use at Layer 3. This is why you'll sometimes see MPLS referred to as a Layer 2.5 protocol since it's slotted in between Ethernet (Layer 2) and IP (Layer 3) in common usage.

When you get to Layer 4, we're no longer dealing with addressing of endpoints, but the addressing of services at endpoints and the imposition of reliable transport (think: sequence numbers, acknowledges and retries). This is pretty straightforward and sensible—you want your email messages to be directed to the email application and your web pages to show up in your browser.

Layers 5 through 7 are not generally the province of hardware-based forwarding systems, so they're not of significant interest within the context of this book. We'll mostly ignore them.

Realistically, you'll probably need to be somewhat conversant in the OSI layer model. But, practically speaking, you can think of Layer 1 (bits on the wire, fiber or radio waves) and then everything else. There are much more effective and simpler models for thinking about networking that actually relate to what exists in the real world and that serve as a useful guide for creating new systems. Let's get into that now and discuss the characteristics of an abstract forwarding entity.

Abstract Forwarding Entity Characteristics

Before diving into the details of specific forwarding protocols and methods, it's useful to consider an abstract model of a forwarding entity. By examining an abstract forwarding entity, you'll build a mental model for forwarding that is stripped of all of the noisy and messy details that are required of actual forwarding entities. The characteristics of our hypothetical abstract forwarding entity are easily mapped to the actual characteristics of real-world forwarding entities such as Ethernet bridges and IP routers.

To best understand the incredibly simple and straightforward definition of the role of a forwarding entity, a handful of essential concepts must be introduced. These will all be explored in much greater detail later.

Packets

A packet is a fundamental unit of network information. In general, its length can range from some protocol-specific minimum to a protocol- or network-specific maximum. A single message from one network endpoint to another may either fit into a single packet or may be split across several packets for longer messages.

Packets are forwarded independently of one another (including packets that may all be conveying parts of the same message). As self-contained units of information, they must include the information required to deliver them from their source to their intended destination. This information is enclosed within one or more headers.

Headers

All packets forwarded by a forwarding entity must contain at least one outermost header that is specific to the type of the current forwarding entity. In sequence, an outer header is one that is located toward the head (or beginning) of a packet. Inner headers are located in sequence away from the head of a packet. In a Platonically idealized world, a forwarding entity only examines the outermost header and that header is of a type that matches the forwarding entity's type. For example, a purely IP router does not know what to do with a packet whose outermost header is Ethernet, it only understands IP headers.

An imaginary outermost header is always prepended to a packet upon receipt by a forwarding entity. This imaginary header is the receive interface header. Its function is to identify the interface via which the packet was received. Certain types of forwarding entities only consider the receive interface header when making a forwarding decision (specifically: cross-connect). More commonly, however, the receive interface header provides vital information that is combined with addressing information from the outermost header to make forwarding decisions. Since the receive interface header never appears on a physical network connection, it can be thought of and handled as packet metadata within a forwarding entity.

Addressing

Forwarding entity-specific headers must contain addressing information that can be used to make forwarding decisions. Optionally, these headers contain source-identifying information that makes it simple to send a reply to a received packet or make other policy-related decisions that are source specific. Address values may be statically or dynamically assigned to network nodes, and they may be global or local in scope.

Not all headers contain addressing information. They may, instead, convey forwarding domain, priority or security information.

Flows

A flow is a connection between any two endpoints on a network (or a one-to-many connection for multicast and broadcast cases). Endpoints may be computers (including servers and storage elements) or services running within a physical endpoint (e.g., web server, etc.). A forwarding entity may also be an endpoint since the control plane of a forwarding entity is, indeed, addressable. Control planes and other aspects of hardware architectures are discussed in detail in Chapter 20 on page 335.

Interfaces

Every forwarding entity must have at least two interfaces. There is no upper limit to the number of interfaces that a forwarding entity may have. Packets are received and transmitted via these interfaces. For our abstract forwarding entity, we can assume that the interfaces are infinitely fast.

Physical, Logical and Virtual

Networks are built of physical things: wires, connectors, bridges, routers, etc. Bridges and routers are also built of physical things: packet buffers, forwarding databases, etc. However, it is often very valuable and powerful to be able to subdivide these physical things into multiple, independent things that have all of the behavioral characteristics of the whole physical thing. Hence, physical ports may be divided into several logical ports. A physical network (i.e., the often complex interconnections between forwarding systems) may be overlayed with any number of, potentially, simpler virtual networks. Finally, the valuable resources within a forwarding entity (e.g., the forwarding databases) may be divided into several virtual tables to support multiple protocols and/or multiple customers without conflict.

Forwarding Domains

Forwarding domains are used to virtualize networks and, more specifically, the forwarding hardware that is used to create and operate those networks. There is a one-to-one correlation between a forwarding domain and an idealized, abstract forwarding entity. Each forwarding entity represents one and only one forwarding domain. The movement of packets from one forwarding domain to another and the restrictions on forwarding imposed by forwarding domains are fundamental parts of networking and are explored in depth later on.

The Forwarding Entity Axiom

Now that some essential concepts have been introduced, we're ready to consider the central axiom of networking that defines the fundamental behavior of each and every forwarding entity:

> A forwarding entity *always* forwards packets in per-flow order to zero, one or more of the forwarding entity's own transmit interfaces and *never* forwards a packet to the packet's own receive interface.

Let's tease that axiom apart.

The “in per-flow order” phrase stipulates that packets belonging to the same flow must be forwarded in the order in which they were received. In-order forwarding is mandated by some protocols (e.g., Ethernet) and is optional for others (e.g., IP). However, in practice, in-order forwarding is expected and required by virtually all applications and customers. The reason for this is that there is a significant performance penalty at an endpoint when packets arrive out of order. Out-of-order delivery for those protocols that do not mandate in-order delivery can be tolerated as very brief transients.

The “to zero, one or more […] interfaces” phrase indicates that a single receive packet may spawn multiple transmit copies of that packet. This is, of course, the essence of multicast and broadcast behavior. Each of the copies of the packet is identical as it arrives at the forwarding entity’s several transmit interfaces. However, as it is transmitted, the packets may have new encapsulations added as they emerge from the forwarding entity. The importance of this behavior is discussed when we delve into multicast operations and tunneling. The reference to zero transmit interfaces allows a forwarding entity to discard a packet if it cannot forward the packet towards the packet’s intended destination or drop a packet if congestion is encountered.

The “forwarding entity’s own transmit interfaces” phrase means that a forwarding entity absolutely cannot forward a packet via a transmit interface belonging to another forwarding entity. Remember that each forwarding entity represents a single forwarding domain. This forwarding restriction limits the forwarding entity to simply forwarding within the domain associated with the forwarding entity. This may seem crazily restrictive; and it is. But, for good reason. Forwarding domains are used to isolate groups of flows so that it is impossible for packets from one group of flows to accidentally leak over to another; a violation that could represent a serious security breach. Do not despair, however, there is a way for packets to move from one forwarding domain to another in a controlled fashion. The exact method for doing so is covered in depth when we discuss tunnels and virtualization.

The “*never* forwards a packet to the packet’s own receive interface” phrase prevents packets from ending up back at their origin (the original sender of a packet certainly isn’t expecting to receive a copy of a packet that it just sent) and to prevent the formation of loops within the network that may spawn copies of packets ad infinitum.

Keep the central axiom in mind and refer back to it as necessary. It applies to every networking protocol, architecture and scenario covered by this book.

4 Tunnels

It may seem odd at this juncture to jump right into what most people consider to be an advanced topic. However, tunneling really is fundamental. Without tunneling, only the simplest and most primitive connections are possible. Specifically, all that can be done without tunneling is direct, physical, point-to-point connections between pairs of endpoints as one might get with an RS-232 serial link (remember those?). With such a link, addressing isn't necessary because every byte sent by one endpoint is intended for the other endpoint and no other destination is possible. Once we define a packet format that includes a header with a destination address value, we've just turned that physical connection into a tunnel.

Yes, even with the simplest possible Ethernet network, tunnels are in use, with the physical medium—copper wires, optical fibers, radio waves, etc.—serving as the outermost tunnel conveying all of the packets.

So, what have we accomplished by conveying, say, Ethernet packets through a gigabit per second twisted pair tunnel? What we've done is abstracted away the contents of the wire and made it possible to just be concerned about the wire itself when building the physical network and not be concerned about what's on the wire—or, more precisely, what's being conveyed by the physical layer tunnel. The contents of the wire are said to be *opaque* to those components of the wire that only concern themselves with the physical layer (e.g., cables, connectors, PHYs, etc.).

Typically, a tunnel can carry many independent flows. Each flow is, in most ways, a peer of the other flows in the same tunnel. These flows can also be tunnels; each carrying its own, independent set of flows. Tunnels within tunnels is referred to as encapsulation. This process of encapsulating tunnels within tunnels can be continued to arbitrary depths.

In our real, non-digital world, road or rail tunnels through mountains and under rivers have entrances and exits. A vehicle may take one of several routes to arrive at a particular tunnel entrance and may subsequently follow one of many routes upon exiting the tunnel. However, while in the tunnel, the vehicle has little choice but to go where the tunnel takes it. Tunnel entrances and exits are described as *origination* and *termination* points, respectively. These origination and termination points have identifier values (i.e., addresses) that make it possible to navigate to a particular point when several such points are available as options.

In a network, the origination and termination points are identified by address values, port numbers and labels (all explained further along in protocol-specific discussions). Typically, these values are carried along with the packets in a series of tunnel-specifying headers. For one particular tunnel type—the physical layer tunnel—this addressing is implied by the port numbers of the forwarding entities that serve as tunnel endpoints.

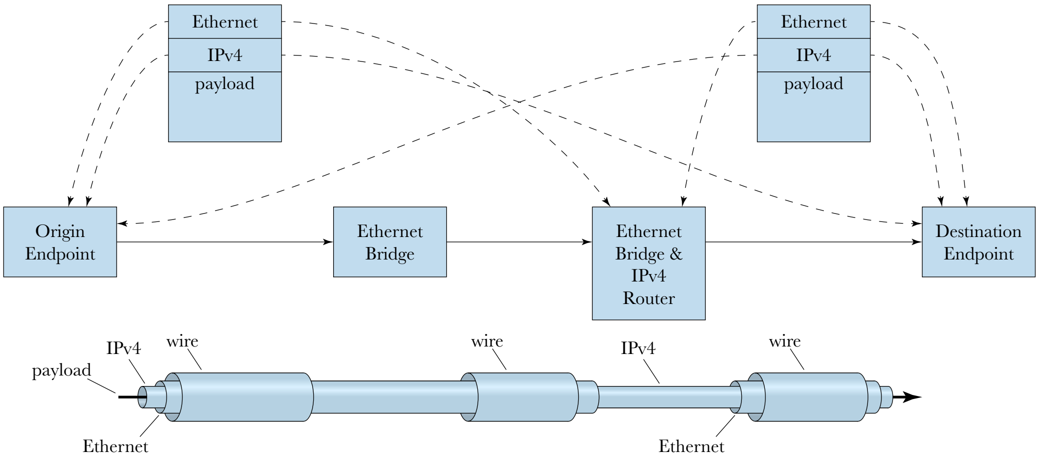


Figure 1 *Tunnels in a Simple Network*

In the following discussion, numerous references are made to Ethernet and IPv4. Do not be concerned if you are unfamiliar with the details of these forwarding protocols. Those details are not important for grasping the basics of tunneling.

In Figure 1, a pair of endpoints communicate via a modest number of intermediate points (i.e., forwarding systems). From left to right, a packet encounters an Ethernet bridge and an Ethernet plus IPv4 forwarding system, respectively. The dashed arrows indicate to which entities headers within the packet are addressed. Left-facing arrows represent source addresses while right-facing arrows represent destination addresses.

All packets are addressed to their intended destination in some manner or another by their headers. Resolving the meaning of the destination address in any particular header leads to one of three possible outcomes:

- The address is unknown.
- The address matches an entry in a forwarding database of a forwarding entity (i.e., its forwarding information base, or FIB).
- The address matches an address that belongs to the forwarding system itself.

If the destination address of a packet's outermost tunnel encapsulation is unknown to the forwarding system, then some protocol-specific action is taken. Options include discarding the packet silently, discarding the packet and informing its source that it was received in error, or forwarding the packet in some default manner that helps get the packet closer to its intended destination.

If the destination address of a packet's outermost tunnel encapsulation is found in a forwarding database of a forwarding system, then the packet is forwarded as specified by the contents of this table. This action either delivers the packet to its destination directly (if the destination is directly attached to the forwarding system), or it gets the packet to the next node in the network that is closer to the intended destination.

Finally, if a packet is received by a forwarding system and the destination address of the packet's outermost tunnel encapsulation matches one of the addresses owned by that forwarding system, then that outermost tunnel is terminated, exposing the tunnel's payload. If the payload's protocol type matches a capability of the forwarding system (i.e., the forwarding system understands how to deal with such a packet), the forwarding system processes the payload as if it were a newly-received packet (possibly exposing yet another payload). The process of decapsulation continues until an encapsulation layer is reached whose destination address is either unknown to the forwarding system or exists in the forwarding system's forwarding database.

Let's return to our example in Figure 1 on page 15.

The origin endpoint (on the left) encapsulates the information that it is trying to convey to the destination endpoint (on the right) into an IPv4 packet. This information is now the payload of the IPv4 packet. The IPv4 packet is, in turn, encapsulated in an Ethernet packet. Finally, by transmitting the Ethernet packet onto the link that spans from the source endpoint to the forwarding entity to which it is directly attached, the source endpoint has encapsulated the Ethernet packet into a physical layer tunnel (e.g., 1000Base-T).

The first forwarding system (an Ethernet bridge) only understands how to work with Ethernet packets. It receives and transmits Ethernet packets and is completely unconcerned with the payloads of the Ethernet packets (i.e., the IPv4 packet). At this forwarding system, the physical tunnel is exited (terminated) and the Ethernet packet is exposed. The forwarding entity examines the Ethernet header (i.e., its tunnel specification) and determines to which port to forward the packet. It is important to note that the Ethernet tunnel is not terminated at this point because the Ethernet packet is not addressed to the current forwarding system; it is addressed *beyond* the current forwarding system.

The transmission of the packet by the first forwarding system effectively encapsulates the packet into a new physical-layer tunnel (i.e., the wire) for its short trip to the second forwarding system.

The second forwarding system understands both Ethernet and IPv4. If an Ethernet tunnel terminates at this point, the forwarding system can forward the packet based on the Ethernet packet's IPv4 payload. Indeed, at this forwarding system, the physical layer tunnel is terminated just as it was at the first forwarding system. However, at this stage of forwarding, the Ethernet tunnel is also terminated because the Ethernet packet's destination address matches one of this forwarding system's own Ethernet addresses. Terminating the Ethernet tunnel (by disposing of the Ethernet header) exposes the IPv4 packet within, allowing the IPv4 packet to exit the Ethernet tunnel. The IPv4 packet is then processed by the second forwarding system and forwarded toward its destination.

Forwarding the IPv4 packet by the second forwarding system requires that two tunnels be entered—one right after the other—before the packet can be transmitted. The first tunnel is an Ethernet tunnel that leads to the destination endpoint. The Ethernet tunnel is entered by encapsulating the IPv4 packet inside a new Ethernet packet (i.e., by prepending a new Ethernet header). The destination address of this new Ethernet header points to the ultimate destination of the packet while the Ethernet source address points to the current IPv4 router. The second tunnel to be entered is a physical-layer tunnel that also leads to the destination endpoint. (You know you're getting close to a packet's ultimate destination when all of its current tunnels terminate at the same place.) The interface number associated with the new physical-layer tunnel is used to direct the packet to the correct physical transmit interface of the second forwarding system.

Upon receipt of the packet by the destination endpoint, the three encapsulating tunnels (physical, Ethernet and IPv4) are terminated by validating their addressing and the associated headers are stripped from the overall packet, exposing the original message from the source endpoint.

Here are some important things to observe about what happened in this example.

- When each tunnel was entered, a new, outermost layer was added to the packet, and that layer was removed (i.e., stripped) from the packet as each tunnel was exited.
- Within a particular tunnel, the inner headers that are part of that tunnel's payload were ignored (i.e., they were treated as opaque content).
- At any particular point in the network, forwarding decisions (which are distinct from tunnel terminations, though both involve the examination of a header's destination address information) were made based on a single header. This single header is known as the forwarding header. Various headers of a particular packet may be used as forwarding headers at different points as that packet traverses the network.

All of this popping into and out of tunnels may seem like a lot of pointless extra work. Why bother? Why not simply address the original packet to its ultimate destination and be done with it? The short answer is scalability.

Tunnels and Scalability

Large networks such as the Internet are not built from a vast and complex web of forwarding entities that are all owned and operated by a single organization. Instead, a hierarchy of networks is built such that a network operator at a lower level of the hierarchy utilizes the services of an operator whose network is at the next higher level of the hierarchy.

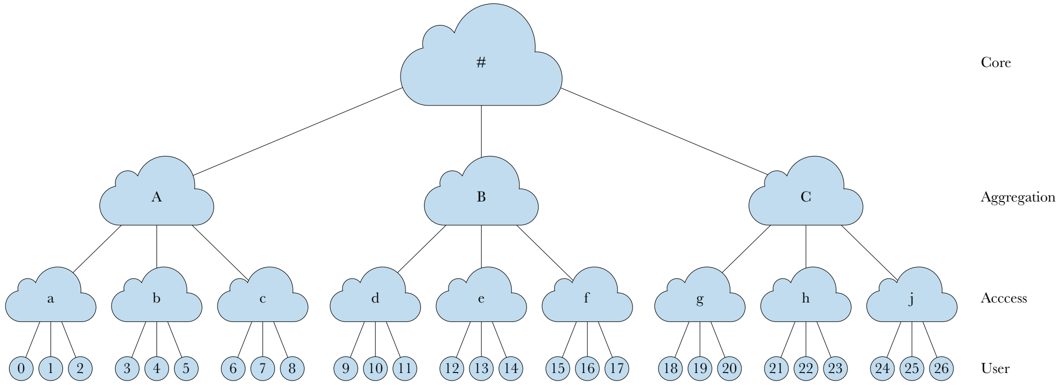


Figure 2 *Network Hierarchy*

Figure 2 depicts a hypothetical network hierarchy. In practice, the number of layers and the names of the layers may be different (and it certainly won't be so neatly symmetrical), but it serves to illustrate the benefits of tunneling.

Let's examine a scenario where User 9 wants to send a packet to User 22. User 9 knows User 22's address, so User 9 encapsulates its packet with a header that is addressed directly to User 22. User 9 also knows that for its packets to get anywhere it has to use the services of Access d. Hence, an encapsulation header addressed to Access d is added to the packet.

The packet that is received by Access d is addressed to it, so it strips off the outermost encapsulating header and examines the contents. Access d doesn't know how exactly to get the packet to User 22, but it does know that to get to Users 21 through 23, it has to go through Access h. So, what Access d does is encapsulate the packet in another new header that is addressed to Access h. To get to Access h, Access d must use the services of Aggregation B by prepending an appropriate tunnel encapsulation header (i.e., one that is addressed to Aggregation B). Access d then sends this packet with its three encapsulating headers to Aggregation B.

By the time we get to Aggregation B, we can start to see the benefits of tunneling. After terminating the tunnel from Access d to Aggregation B, Aggregation B is now working on a packet whose outermost header is addressed to Access h. Aggregation B doesn't need to know anything about all of those User nodes, keeping its forwarding databases small. Further, its network—offering connections to Access d through Access f and to Core #—can operate independently from all of the other networks, using its own preferred forwarding protocol (e.g., IPv6 or MPLS) and running its own routing state protocol (e.g., BGP, etc.) without having to react to state changes within Aggregation A, Aggregation C or any other network. This reduction in scale and complexity of Aggregation B increases its efficiency, performance and reliability.

To continue with the forwarding scenario, Aggregation B adds a further encapsulating headers that are addressed to Aggregation C and Core #. The packet is forwarded to Core # which, in turn, delivers the packet to Aggregation C. Notice here that Core # has the simplest job of all of the networks because it doesn't have to perform multiple encapsulations or decapsulations of the packet. It simply examines the address in the outermost header (after stripping the header addressed to itself) and forwards the packet to the appropriate interface, where it then adds a header addressed to Aggregation C.

At this point, the encapsulating tunnel headers start to come off. With each hop away from the core and down the levels of hierarchy, a tunnel header is added to get to the next hop, but that next hop strips that header (because that header is addressed to itself) and the next header (because the next header is also addressed to itself). This process is repeated with a smaller and smaller stack of tunnel encapsulation headers until the packet finally reaches User 22.

Figure 3 illustrates the encapsulation changes that the packet goes through on each of the links on the path from User 9 to User 22.

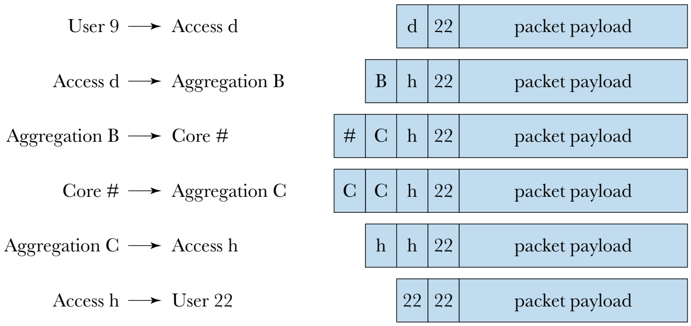


Figure 3 *Packet Encapsulation Life Cycle*

Here are some interesting things to observe about the encapsulation life cycle depicted in Figure 3:

- The outermost (leftmost) header is always addressed to the packet's immediate destination (i.e., the next hop).
- The innermost header is always addressed to the packet's ultimate destination.
- As the packet proceeds from the edge to the core, headers are added that are later interpreted and stripped as the packet proceeds back to the edge.
- Though it appears that there are redundant headers once the packet is heading away from the core (e.g., C and C on the Core # to Aggregation C link), those seemingly-redundant headers are associated with a different level of hierarchy.

Below is a more literal view of the tunnels-within-tunnels aspect of large-scale networks.

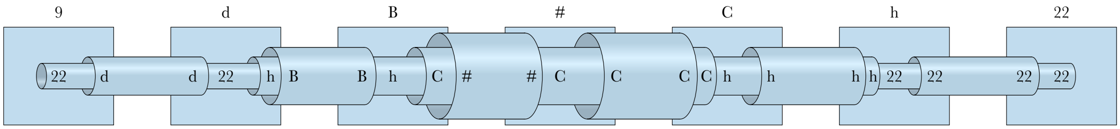


Figure 4 *Tunnels Visualized*

In Figure 4, it is clear that the Core # forwarding entity has no visibility into the h tunnel or the 22 tunnel; it simply treats those tunnels (and the 22 tunnel's payload) as opaque content of the C tunnel.

Tunnels and Isolation

It is important to recognize that the depiction of a network as a friendly little puffy cloud is used simply to abstract away a significant and distractingly large amount of detail. In reality, these little clouds are made up of their own very complex inner structure.

If we presume that the cloud depicted in Figure 5 represents a hypothetical Internet provider network, then two fundamental types of forwarding systems are used to build this network: provider edge systems (depicted as "PE") and provider core systems (depicted as "P"). The provider edge systems provide the outward-facing interfaces to the service provider's network. To serve this role, the provider edge systems must be able to support whichever protocols the service provider's customers choose use, they must incorporate elements outside of the service provider's network into their forwarding information databases, and they must normalize and de-normalize all of the packets that enter and leave the service provider's network.

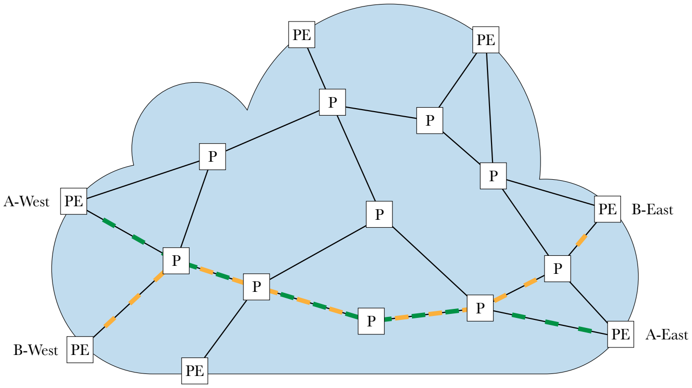


Figure 5 *Inside a Cloud*

The process of normalizing and de-normalizing packets is essentially the tunnel encapsulations and decapsulations described previously. The tremendous benefit of this is that the provider core systems need not be at all aware of the world outside of the service provider's network.

A typical service provider provides service to a large number of customers, each of whom wants their data to be isolated from all of the service provider's other customers. For example, imagine that A Corp and B Corp each have West Coast and East Coast offices. It is, of course, very reasonable for both A Corp and B Corp to want to link their East and West coast offices with high-speed, reliable and private network connections. If, for this simple example, A Corp contracts with the service provider to use one of its East Coast provider edge systems and one of its West Coast provider edge systems, and B Corp does the same with different provider edge systems, the service provider can establish two independent virtual private networks by configuring two separate tunnels: one for A Corp and one for B Corp. All of the provider core systems remain blissfully unaware of A Corp and B Corp and simply forward the opaque tunnel contents from an ingress provider edge to an egress provider edge. The isolation between the two tunnels is maintained even if the paths that each of them follow through the service provider's network share common provider core systems and share common links between provider core systems, and even if there are overlaps in the addressing space used by the two customers.

In a sense, both A Corp and B Corp can view the service provider's network as a single giant, continent-spanning wire to which they each have exclusive access. If a customer has more than two access points to the service provider's network, then the service provider's network appears to that customer as if it were a single, continent-spanning forwarding system. Indeed, emulating the behavior of a specific type of forwarding system using a vast network of heterogeneous forwarding systems is a very real and very important networking function.

Further levels of privacy can be assured through encryption and authentication. Security-related topics are covered in Chapter 16 on page 277.

Generally, source and destination address information is used to derive forwarding domain and receive interface ID parameters for the payloads of terminated tunnels. A tunnel header's destination address identifies a tunnel exit whereas its source address identifies the tunnel's entrance. The tunnel exit is analogous to a forwarding domain while the tunnel's entrance is analogous to a receive interface ID. Several tunnel entrances may all lead to the same tunnel exit. That tunnel exit is associated with a single and particular instance of a forwarding entity within the forwarding system in which the tunnel is being terminated. The tunnel entrances represent the receive interface in that there may be several receive interfaces associated with a single forwarding entity.

5 Network Virtualization

If tunnels permit the assembly of vertical hierarchies of networks, virtualization permits their horizontal scaling. Before virtualization, all endpoints and forwarding entities in a network were visible to one another, and all of the forwarding database (or, commonly, forwarding information base or FIB) resources of the forwarding entities were shared as monolithic chunks of memory. This may, at first consideration, seem like just the right way to build a network—after all, every endpoint can communicate with every other endpoint in such a network—but it quickly leads to administrative problems as the networks grow larger and larger.

It turns out that allowing unfettered communication from each endpoint to every other endpoint is not always the right thing to do. It may be good for everyone in an engineering department to perform peer-to-peer communication, but you would probably want to isolate the engineers from the sensitive data on the machines in the finance department. In situations like this (and many others) it is best to provide trusted intermediaries to allow just the right kinds of connections between departments. As another example, a data center that sells storage and processing services to thousands of customers wants to be able offer those customers the resources of multiple endpoints (i.e., servers) but also provide privacy and isolation so that their data—and the network interconnecting the servers that they're paying for—is safely isolated from all of the data center's other customers.

By dividing a network into a number of virtual networks, this kind of isolation becomes possible while still allowing communication between the virtual networks through specialized portals. Now, the answer to this rhetorical question may seem obvious, but the question must be asked nevertheless: Why not just build separate physical networks? To answer that, we must consider the benefits of sharing.

The Benefits of Sharing

One could certainly build a large network that is, physically, a network of networks where, at the user-access level, each user in a group of peers is attached to the same physical network, and those networks of peers are then interconnected at a higher level of hierarchy where controls are in place to allow only certain types of transactions between the networks to occur (which may be none at all).

There are, of course, problems with this approach. First, physical resources are expensive. They're expensive to acquire, expensive to install, expensive to power and cool, and expensive to reconfigure. This is true regardless of whether these resources are fiber optic links, front-panel ports, forwarding databases, rack-scale routers or even massive data centers. If these resources are poorly utilized, then significant amounts of money are being wasted.

Second, as users or customers of a network come and go and move from place to place, it is prohibitively expensive to add or remove equipment and rearrange the interconnections required to integrate that equipment into the overall network.

Finally, the demands of the users or customers of a network are rarely at constant levels. A network operator is placed in the unenviable position of either allocating the maximum that a customer may need at some future date (thus wasting significant resources) or having to scramble when the customer's demand suddenly spikes.

Sharing network resources and allocating varying fractions of these resources as demand ebbs and flows maximizes the utilization of limited commodities. Virtualization provides the means for multiple independent users, customers or applications to share a common set of physical resources without conflict and to have the scale and performance of their private slice of the network instantly react to changing demand levels.

A secondary, but still important, aspect of virtualization is that it provides a means for breaking up a large, complex system into several smaller pieces. These smaller pieces are then interconnected via a hierarchy of specialty forwarding entities as described in Chapter 4 with the usual benefits: achieving massive scale of scope and capacity while maintaining ease of administration and overall stability and responsiveness.

When we break a network up into several independent virtual networks, what's really happening is we're defining forwarding domains. Just as a packet cannot magically jump from one network to another without some kind of intermediary that knows how to accomplish that, packets cannot be forwarded from one forwarding domain to another without a specialized intermediary. To be concise, when there's discussion of virtual LANs or virtual private networks (VPNs), what's really being discussed is the management and operation of independent forwarding domains.

Virtualization and the Abstract Forwarding Entity

Regardless of whether the term “virtual” or “logical” is applied, the concept is the same; some physical resource or another is subdivided into a number of independent instances of the same type. For example, a physical port (e.g., a front-panel network connection) can be subdivided into a number of logical ports, each configured with its own per-logical-port attributes. As far as the forwarding entity that’s behind the subdivided physical port is concerned, each of those logical ports is an actual, separate port with all of the attributes and characteristics of a physical port.

Perhaps the most common and most powerful form of virtualization is the virtualization of forwarding systems themselves. Virtualizing forwarding systems means that a single physical forwarding system (i.e., a box installed in a rack) may be host to a large number of virtual forwarding entities, each operating independently of the others. Indeed, the virtualization of forwarding systems into a collection of forwarding entities is so powerful and so important that all forwarding entities can be thought of as virtual things sharing a physical resource. Hence, it is unnecessary to prefix the term “forwarding entity” with the “virtual” modifier.

The forwarding entities occupying a single forwarding system are not limited to existing as peers of one another. Complex hierarchies can be constructed so that the structures and behaviors described in Chapter 4 on tunnels can be realized entirely within the confines of a single physical forwarding system. To enable these capabilities, abstract forwarding entities must have the following additional attributes:

- A forwarding entity cannot be subdivided by forwarding domains.
- Forwarding entities within a single forwarding system are connected to one another in a point-to-point manner via their interfaces. These interfaces are logical interfaces and are not exposed outside of the forwarding system.
- Each forwarding entity supports just a single forwarding protocol.
- For a forwarding entity to forward a packet to a forwarding entity of a different type, it must either encapsulate or decapsulate the packet such that the packet’s outermost header is of a type that matches the capability of the next forwarding entity.
- When a packet’s encapsulation changes and it moves from one forwarding entity to another, a new forwarding domain is assigned to the packet that corresponds to the next forwarding entity.

Let’s work through a concrete example to see how these characteristics of idealized abstract forwarding entities express themselves. Consider Figure 6.

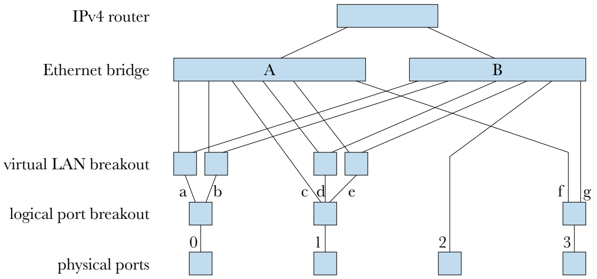


Figure 6 *Bridge/Router Forwarding Entity Scenario*

Each of the boxes in Figure 6 is a forwarding entity. Each of them is specialized for a particular type of encapsulation and forwarding. The physical ports always forward to their opposite interface and provide encapsulation on to and off of the attached physical medium.

The logical port breakout forwarding entities demultiplex packets heading north from the physical ports based on their logical port tags (i.e., a small header that identifies the logical port associated with the packet); and multiplex logical ports in the opposite direction.

The virtual LAN breakout forwarding entities provide similar demultiplexing and multiplexing services based here on virtual LAN (i.e., VLAN) tags. Each of the virtual LAN breakout functions has two interfaces on top; one for each of two VLANs: A and B.

The Ethernet bridge forwarding entities forward based on the packet's Ethernet header. One of each bridge's possible destinations is the IPv4 router. There are two bridge instances: one for VLAN A and one for VLAN B.

The IPv4 router forwarding entity, of course, forwards packets based on the IPv4 header.

In the limited example, above, the IPv4 router can only ever forward a packet to the Ethernet bridge that was *not* the router's source of the packet (in a real-world system, a router forwarding entity may have many thousands of virtual interfaces to thousands of bridging forwarding entities).

Each type of forwarding entity only understands how to work with packets of its own specific type. The packet's type is indicated by its outermost header. So, a logical port header is interpreted by the logical port breakout forwarding entity, a VLAN header is interpreted by the virtual LAN breakout forwarding entity, and so on. In the northbound direction, these headers are stripped from a packet as it progresses, exposing a new outermost header that is appropriate for the next stage of processing. In the southbound direction, new encapsulating headers are added to a packet in order for the next forwarding entity to direct the packet to where it needs to go.

Looking just at the Ethernet bridge and IPv4 router layers, in the northbound direction and considering an Ethernet packet that is addressed to the IPv4 router, an Ethernet bridge knows that it is connected to an IPv4 router via a particular interface, so it strips the packet's Ethernet header prior to forwarding the packet to the IPv4 router, ensuring that the IPv4 router receives a packet whose outermost header is IPv4. In the southbound direction, the IPv4 router must encapsulate each packet in an Ethernet header whose destination address corresponds to the packet's next Ethernet destination and whose source address is set to the router's Ethernet address; thus, an Ethernet bridge can forward the packet to the correct logical port (the VLAN being implied by the bridge's identity).

In the Bridge/Router scenario, above, there are many valid paths that allow a packet to be forwarded back to its own physical receive port without violating the central forwarding entity axiom that specifically prohibits a forwarding entity from forwarding a packet to that packet's own receive interface. The important thing to recognize is that the following scenarios are describing a forwarding system and not a single forwarding entity.

There are two simple ways for a packet received via physical port 0 to be legally transmitted by physical port 0. The first way has the packet tagged with a logical port tag that indicates the packet is in logical port a's domain. If the packet is tagged with a VLAN tag for VLAN A, then Ethernet bridge A may forward that packet via its interface that leads to logical port b. The packet's logical port tag is updated to indicate that the packet is now in logical port b's domain and it is transmitted by physical port 0 without violating any rules.

In the second example, we push the divergence/convergence point up one layer to the VLAN breakout functions. Here, a packet is received via physical port 0 with a logical port a tag and a VLAN tag for VLAN A. Through each stage in the northbound direction, headers are stripped from the packet, ultimately arriving at the IPv4 router as an IPv4 packet. The IPv4 router encapsulates the packet in a new Ethernet header and forwards it to Ethernet bridge B which, in turn, forwards the packet to logical port a with a VLAN tag for VLAN B. Thus, with a logical port a tag added, the packet is transmitted by the physical and logical port via which it was received. It is spared from violating a forwarding rule by belonging to two different VLANs for receive and transmit.

Other interesting things to observe about the example are that logical port c only exists in VLAN A and that physical port 2 is only associated with Ethernet bridge B (and, hence, VLAN B) and has no logical port value. For these cases, the headers for these values are optional as they can be deterministically implied by the physical or logical ports.

The simple example presented here represents just a glimpse into what is possible with virtualization and tunneling. Systems of fantastic sophistication and complexity can be composed from the fundamental building blocks just described. It's simply a matter of adding sufficiently many instances of forwarding entities of the appropriate type and then configuring and interconnecting them (logically speaking) in order to achieve the required behavior.

Real-World Implications

It is certainly possible to build a forwarding system using physically discrete forwarding elements of the various types required and then wiring them together in an appropriate manner. Unfortunately, this would be completely inflexible, extraordinarily complex and prohibitively expensive.

When complex forwarding systems are built in the real world, multiple instances of physical implementations of a forwarding entity's packet processing methods and algorithms are instantiated as often as needed in order to meet the system's performance requirements as measured in packets or bits per second. In other words, a forwarding system may consist of lots of interconnected packet processing chips of the same or similar type. This method for building a system of the required scale is unrelated to the virtualization of network forwarding entities. Instead, each of the instances of the physical devices that are used to build a forwarding system must, themselves, support multiple virtual instances of the abstract forwarding entities that give the physical devices their networking behaviors. When considering the architecture and capabilities of a forwarding system, it is important to distinguish between a forwarding system's physical implementation and its logical or virtual configuration.

To properly support virtualization, every forwarding entity must be able to identify the interface via which a packet is received. This is necessary in order to apply interface-specific attributes to the packet and to ensure that the packet's receive interface is explicitly excluded from the transmit interface solution set. Physical receive ports have hardwired or configurable identifier values. Logical ports are identified either implicitly by a packet's physical receive port ID or explicitly by a logical port header in the packet. In either case, the packet's receive interface is encoded in the packet's metadata when the packet arrives at a forwarding entity.

A packet's forwarding domain is also encoded either implicitly or explicitly in the packet. An example of an explicit forwarding domain encoding is a VLAN header (see *"Virtual Local Area Networks (VLANs)" on page 58*). The VLAN header provides a conceptually simple means for directly specifying the forwarding domain of a packet. However, for a large number of cases, the forwarding domains are implied and are never directly encoded into a packet.

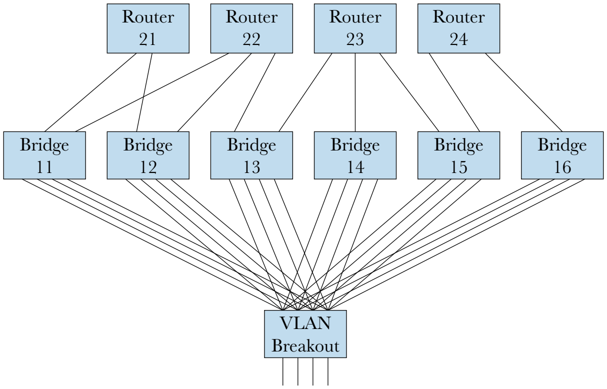


Figure 7 Multiple Bridges, Multiple Routers

Consider, for example, a forwarding system that is made up of several virtual Ethernet bridges and several virtual IPv4 routers. Each bridge may connect to several routers and each router is certainly connected to several bridges. When a packet is being processed by a particular virtual bridge (i.e., an Ethernet forwarding entity), that packet resides in the bridge's forwarding domain. To route a packet using IPv4, the current bridge—which, of course, cannot process IPv4 itself—must deliver the packet to one of several routers via point-to-point logical connections. When an Ethernet bridge forwarding entity chooses a router instance to route a packet, it is also implicitly assigning a new forwarding domain to the packet that corresponds to the forwarding domain associated with the virtual router instance. Similarly, when a virtual router forwards a packet to a virtual bridge instance, the packet adopts the forwarding domain of that bridge. Thus, at each stage of processing, a packet has both an interface identifier and a forwarding domain identifier that is specific to the forwarding entity at that stage.

In Figure 7—where the numbers in the boxes are forwarding domain identifiers—a packet received by Bridge 12 from one of its four exposed interfaces belongs to forwarding domain 12. The packet's Ethernet destination address is associated in Bridge 12's forwarding database with Router 22. When Bridge 12 forwards the packet to Router 22, it not only strips the packet's Ethernet header, it also updates the packet's forwarding domain[2] from 12 (the input bridge's forwarding domain) to 22 (the Router's forwarding domain). Router 22 looks up the packet's IPv4 destination address and determines that the packet's next hop on its path to its destination is reachable via Bridge 13. Thus, Router 22 prepends a new Ethernet header to the packet that is addressed to the packet's next hop and updates its forwarding domain from 22 (the router's forwarding domain) to 13 (the output bridge's forwarding domain).

[2] The forwarding domain designator for a packet is metadata maintained by the forwarding system and is not actually a part of the packet itself.

To virtualize the forwarding entities in a real system, what's needed is to virtualize the resources of these functions: to allow the configuration registers and tables, and the forwarding databases of the several virtual instances of the required forwarding entities to share physical resources without any instance interfering with any other. In practice, this is quite easily accomplished. Interface identifiers and forwarding domain identifiers can be used as indexes into tables to fetch attributes that affect the processing of the packet. When concatenated with addressing values from packet headers (and other relevant extracted or computed values) the forwarding domain identifier value is used to virtualize the forwarding database of the physical manifestation of a forwarding entity. By extending the keys in the forwarding database with the forwarding domain values, it is assured that database keys belonging to one forwarding entity instance can never be confused with the keys from another.

The internal hardware architecture of real-world physical systems almost universally does not resemble the idealized models presented here. Though the specific details of the various forwarding protocols differ, a lot of the underlying mechanisms are very similar and significant implementation efficiencies can be realized by sharing mechanisms such as lookup algorithms and resources such as forwarding databases across these forwarding protocols. It is also reasonable—and common practice—to treat tunnel terminations as being distinct from forwarding operations, even though the packet formats and underlying algorithms are identical for, say, Ethernet, regardless of whether an Ethernet tunnel is being terminated or an Ethernet packet is being forwarded. The great benefit of treating tunnel terminations as being distinct from forwarding is that there are typically several orders of magnitude difference between the number of tunnels that terminate at a physical system and the number of destinations to which that same system may need to forward packets. Hence, a tunnel termination function can be implemented very compactly—and operate very quickly—relative to forwarding, making it practical to perform those operations serially without encountering undue hardware size or complexity.

6 Terminology

Before proceeding into the details of particular protocols, let's get acquainted with networking's terminology.

Any engineering discipline demands its own vocabulary. To be fluent in that discipline means to be fluent in its terminology. Networking, of course, is no different. There are many terms of art that are common to neighboring disciplines, but far more that are unique to the field of networking. There are likely hundreds of terms that you've encountered for which you've not found an adequate or consistent definition. You may be using many of these terms on a daily basis without total confidence of their correctness. Collecting, vetting, and learning the definitions of these terms is a huge undertaking, made more difficult by its dynamic nature. New terms are invented almost daily and most are immediately reduced to short, overloaded acronyms while the original definitions are lost to the sands of time.

This chapter presents cogent definitions for the terms used in this book. Where multiple definitions exist that do not entirely agree with one another, the most modern and widely-used definition is presented.

The vocabulary of networking consists not just of a list of words and their definitions. It is also a visual vocabulary. As the old saying goes, "a picture is worth a thousand words." While that may not be literally true, there are many cases when a drawing, chart or diagram can add immeasurable clarity to an explanation. Many—if not all—of the reference and standards documents upon which the entirety of networking is founded include graphics of some form or another. Unfortunately, the various bodies that publish these documents (e.g., IEEE, IETF, ITU, etc.) can't seem to agree on the basic elements of these figures, using confusingly similar but inconsistent means for describing conceptually identical things.

To address these inconsistencies, this chapter also presents a visual elements reference that—while differing in its format and style in both subtle and significant ways from the referenced standards—allows for a consistency within this book that must, by necessity, span these several standards bodies while presenting information in a clear manner.

Terms

address	A value that uniquely identifies an endpoint on a network or a path through a network.
append	To concatenate to the end of, typically, a packet.
associated data	Data that is associated with a key in a lookup table. A search operation matches a key value in a table's entry and that entry's associated data is returned as the search result.
bps	Bits per second.
big-endian	In a big-endian system, bits and bytes are numbered from zero starting with the most significant bit or byte. The big-endian numbering system is consistent with network byte order. Big-endian numbering is used throughout this book.
bit	The term "bit" is a portmanteau of "binary digit." It represents an indivisible unit of information that is fundamental to computing, communications and all aspects of information theory. A single bit can represent just two states: 1/0, true/false, on/off, etc. Ordered strings of bits are used to represent wider ranges of values. By convention, the most significant bit of a multi-bit string of bits is depicted as the leftmost bit.
body	See "payload."
bridge	A forwarding entity that is characterized by the automatic learning of source addresses to populate its forwarding database, no time-to-live value in the forwarding header and flooding packets with unknown destinations to all available output interfaces.
broadcast	To forward copies of a packet to all destinations on the network.
BUM	Broadcast, Unknown-unicast, Multicast. This is a handy shorthand for described a class of Ethernet packets that all share the same default behavior in an Ethernet bridge (i.e., flood within their VLAN).
byte	A byte is a string of bits of sufficient width to encode a single alphabetic character. Historically, the width of a byte varied according to individual computer architectures. For several decades now, however, popular computer architectures have settled on an 8-bit byte. Indeed, the width of a byte has been codified by an international standard (IEC 80000-13). As a consequence, use of the term "octet" has fallen from favor.
checksum	A simple means for checking the integrity of, typically, a header. A checksum is easily computed and easily updated incrementally when, say, just one field in a header is updated. Some checksums are applied to the payload of a packet as well as a header. Checksums are weak at protecting against certain multi-bit errors or the transposition of data.
CRC	See "cyclic redundancy check."
cyclic redundancy check	A fairly robust means for verifying the correctness (but not authenticity) of a packet. A cyclic redundancy check (CRC)—also known as a frame check sequence (FCS)—is, essentially, the remainder of a division operation performed across the entirety of a packet. It is effective at catching multi-bit and transposition errors. Its effectiveness does diminish with very long packets.

datagram	The IETF's word for a packet's payload. A datagram can be a packet in its own right. The distinction between the terms "frame," "packet," and "datagram" is trivial and not worthy of concern.
decapsulate	To remove (or "strip") a header from the head of a packet, effectively exposing a packet that was encapsulated within another packet. This is typically associated with exiting a tunnel.
destination	The termination point for a packet or a tunnel.
discard	To dispose of a packet due to some kind of exception condition or simply because the packet has no valid destinations according to the forwarding entity servicing the packet. See also, "drop."
drop	To dispose of a packet due to a lack of buffering, queuing or bandwidth resources. See also, "discard."
encapsulate	To add a tunnel-related header to the head of a packet, effectively encapsulating a packet within another packet. This is typically associated with entering a tunnel.
endpoint	The ultimate origin or destination of a packet.
FCS	Frame Check Sequence. See "cyclic redundancy check."
FIB	Forwarding Information Base. A database that is used to associate packet addressing information with a packet's destination. In this book, the term "forwarding database" is used.
FIFO	First In, First Out. Describes the behavior of a queue.
forwarding database	A collection of keys and associated data. The keys are typically based on addressing field types from forwarding headers while the associated data is typically a set of instructions that specify how to forward a matching packet. Commonly referred to as a forwarding information base (or FIB) in standards documents.
forwarding entity	A fundamental, abstract building block of a forwarding system. A forwarding entity is associated with just one forwarding domain and can only forward packets based on its own native header type.
forwarding system	A collection of forwarding entities.
fragmentation	The process of breaking an IP packet into two or more shorter IP packets.
frame check sequence	See cyclic redundancy check.
frame	The IEEE's word for packets.
G	Giga. This magnitude suffix either means 10^9 (for bit rates) or 2^{30} (for capacities).
header	A collection of fields at the beginning of a packet that provides addressing, interface, priority, security or other metadata related to the packet of which it is a part.
IEEE	Institute of Electrical and Electronics Engineers. This standards organization is widely known for their 802 series of standards (802.1 bridging, 802.3 Ethernet, 802.11 Wi-Fi, etc.).
IETF	Internet Engineering Task Force. This standards organization focuses primarily on Internet-related standards. See also, "RFC."

ISO	International Organization for Standardization.
ITU	International Telecommunication Union. A standards organization that is part of the United Nations (UN).
K	Kilo. This magnitude suffix either means 10^3 (for bit rates) or 2^{10} (for capacities).
key	An entry in a lookup table (e.g., a forwarding database) which is matched against search arguments.
LAN	See "local area network."
least significant bit	The rightmost bit of a multi-bit word. In a little-endian system, the least significant bit is bit 0. In a big-endian system, the least significant bit is bit <word width> - 1.
little-endian	In a little-endian system, bits and bytes are numbered from zero starting with the least significant bit or byte. This is the opposite of network byte order.
local area network	Historically associated with bridged Ethernet networks of limited diameter. The term no longer has much meaning outside of standards meetings.
logical	An abstract reference (in contrast to a physical reference). When a physical resource (e.g., a network port) is subdivided or aggregated, the result is a logical resource. See also "virtual."
loopback	To re-direct a transmit packet at or near the packet's intended transmit interface so that it becomes a receive packet. This is useful for diagnostics and certain complex forwarding scenarios where a packet may require additional processing prior to transmission.
M	Mega. This magnitude suffix either means 10^6 (for bit rates) or 2^{20} (for capacities).
MAC	Media access controller. A MAC arbitrates for access to a physical network, manages the order in which bits are transmitted (and their rate), and reassembles packets from bits received from a network. In this book, MAC is synonymous with Ethernet.
metadata	Information that describes other information. Typically, metadata are computed values that are carried with a packet within a forwarding system for the benefit of the various functional parts of that forwarding system.
metropolitan area network	A more recent term that reflects the promoters of Ethernet's desire to break out of the historically limiting LAN category. By taking on certain "carrier grade" features such as time sync and OAM, Ethernet has been steadily supplanting traditional wide-area media access technologies.
most significant bit	The leftmost bit of a multi-bit word. In a big-endian system, the most significant bit is bit 0. In a little-endian system, the most significant bit is bit <word width> - 1.
MTU	Maximum Transmission Unit. The maximum packet length allowed by a network.
multicast	To forward copies of a packet to multiple destinations, but not necessarily to all destinations.
network byte order	Bytes of multi-byte values are transmitted in big-endian order (i.e., most significant byte first). Ethernet—the most ubiquitous Layer 1 and 2 standard—transmits the least significant bit of each byte first.

nibble	This is an example of engineers trying to be cute. Predictably, the outcome is cringe-worthy. According to most English dictionaries, a nibble is a small bite. In computing, a nibble (sometimes "nybble") is half of a byte; or, more simply, a four-bit word. This term is not commonly used.
octet	An archaic term for an eight-bit word. The use of this term is akin to old-timey pilots with goggles and silk scarves referring to an airport as an "aerodrome." Standards bodies persist in the use of octet, but the rest of the world has moved on. See also, "byte."
opaque	Information that is invisible to or otherwise unavailable or uninteresting to a forwarding entity or forwarding system. If, for example, a header parsing function cannot determine what the type of the next header is, that next header and all subsequent headers are opaque. Packet contents may be parsable but still be treated as opaque if the forwarding entity simply chooses to ignore those contents. See also, "transparent."
originate	To mark the entry point of a tunnel. The prepending of a corresponding header is typical at tunnel origination. A tunnel may originate between a packet's original source and its ultimate destination or at the original source itself.
OUI	Organizationally Unique Identifier. Equipment manufacturers are assigned OUI values for use in creating globally-unique MAC addresses. The OUI occupies the most-significant (i.e., leftmost) 24 bits of an Ethernet MAC address.
packet	A unit of self-contained network information. To be "self-contained" means that a packet contains all of the information necessary to forward itself to its final destination.
payload	The contents of a packet as described by its headers. A packet's payload may contain further headers (describing encapsulated packets) that may be opaque to the current forwarding entity.
prepend	To concatenate to the beginning of, typically, a packet.
priority	To assign a level of importance to packets belonging to a particular flow that differs from those belonging to other flows.
queue	An ordered collection of pending items.
RFC	Request For Comments. Many RFCs (technical memoranda) are adopted by the IETF to become Internet standards.
router	A forwarding entity that is characterized by the use of routing protocols to populate its forwarding database, and a time-to-live value in the forwarding header. Packets with unknown destinations may be forwarded to a default output interface or generate a message to the packet's originating endpoint.
search argument	A value that is submitted to a search algorithm in order to find matching key(s) in a lookup table (e.g., a forwarding database).
segment	1. A single TCP data packet. 2. The result of a segmentation process (please don't call it "cellification") where a packet is broken up into smaller pieces for, typically, conveyance within a forwarding system. Not to be confused with fragmentation. 3. A portion of an Ethernet network where endpoints are, effectively (though rarely actually) connected to the same physical medium.

source	The originating endpoint of a packet or a tunnel.
switch	Flip a coin and see either bridge or router.
T	Tera. This magnitude suffix either means 10^{12} (for bit rates) or 2^{40} (for capacities).
tag	A non-forwarding Layer 2 header. For example, a VLAN tag.
terminate	To mark the exit point of a tunnel. The stripping of the corresponding header is typical. A tunnel may terminate midway between a packet's original source and its ultimate destination or at the ultimate destination itself.
TLV	Type, Length, Value. A TLV is a means for creating self-describing data structures. The "type" element defines the type and purpose of the data structure, giving the interpreting system the means for determining the types and locations of various fields in the structure. The "length" element is used by those systems that are not designed to be able to process a structure of a particular type, allowing such a TLV to be skipped over and for processing to continue with the next TLVs (if any). The "value" element is simply a collection of fields that make up the overall TLV structure.
trailer	A data structure that appears at the end of a packet.
transparent	Available to a forwarding entity for processing. This term is also applied to Ethernet bridging or other networking behaviors that operate independently of other forwarding systems and whose behavior does not affect the behavior of other systems. See also, "opaque."
truncate	To shorten a packet by, typically, deleting bytes from the end of the packet.
TTL	Time-to-Live. TTL values are used in the headers of certain forwarding protocols to limit the lifetimes of packets, thus preventing them from circulating around a network forever.
tunnel	The practice of encapsulating a packet within another packet.
virtual	The logical division of a physical resource. See also, "logical."
WAN	See "wide area network."
wide area network	Historically associated with routed IPv4, IPv6 and MPLS networks of significant diameter. The term no longer has much meaning outside of standards meetings.
word	A string of bits that is of any length other than one or eight. A 1-bit word is a bit. An 8-bit word is a byte. There is no agreed-upon width of a word. So, using terms like "word" or "double-word" are not meaningful or helpful. Always prefix the term "word" with a length modifier: e.g., 24-bit word, 64-bit word, etc.

Graphics

A couple of graphical depiction types are widely used when describing networking systems, protocols, and behaviors. These are depictions of packet structures (i.e., the order in which headers appear in a packet) and the details of the headers themselves. The conventions used in this book are defined below.

Packet Depictions

When depicting packets as a whole, what's generally of interest is the order in which headers are arranged within a packet. Depending on the specific use case of the figure, either horizontal or vertical versions are used. Just as in most written languages, the headers proceed from top to bottom or from left to right.

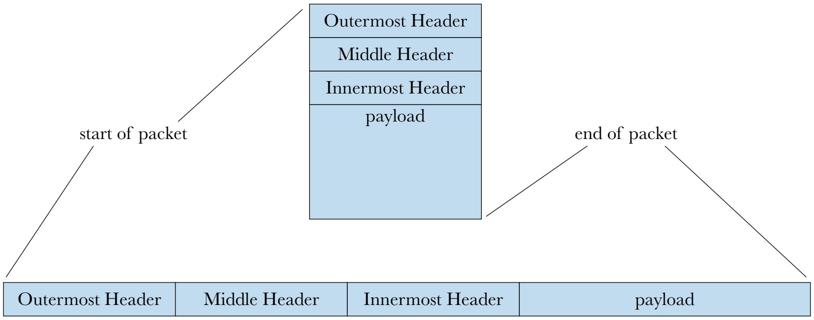


Figure 8 *Packet Depiction Examples*

Header and Data Structure Depictions

Though 64-bit CPUs are commonplace at the time of the writing of this book, I have chosen to stick with 32-bit widths for the words used to contain the fields of packet headers and general data structures. This choice is mostly based on what fits neatly on a page.

Figure 9 depicts a hypothetical header structure diagram that shows the conventions adopted by this book.

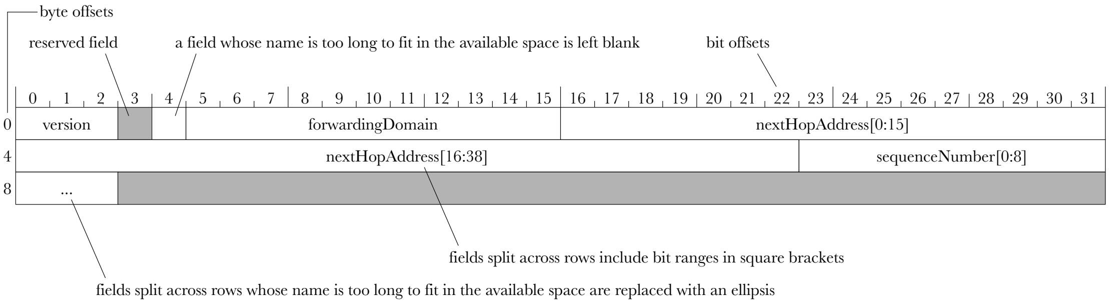

byte offsets
reserved field
a field whose name is too long to fit in the available space is left blank
bit offsets
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
0
version
forwardingDomain
nextHopAddress[0:15]
4
nextHopAddress[16:38]
sequenceNumber[0:8]
8
...
fields split across rows include bit ranges in square brackets
fields split across rows whose name is too long to fit in the available space are replaced with an ellipsis

Figure 9 *Hypothetical Structure Format Diagram*

Every header or data structure diagram is accompanied by a table that defines the fields of the header or data structure and provides the names of the fields that don't fit in the diagram. The field definitions for Figure 9 are listed in Table 1.

Table 1 *Hypothetical Structure Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`version` (version)	3 bits	0.0	Protocol version. The current version of this protocol is `4`.
`reportExceptions` (R)	1 bit	0.4	Enables exception reporting. Exceptions are always reported when this bit set to `1`.
`nextHopAddress` (nh_addr)	39 bits	0.16	Address of the next-hop. Next-hop is simply the next router to which the packet must be forwarded in order for it to progress towards its final destination.
`sequenceNumber` (seq_num)	12 bits	4.23	Sequencing for packet loss detection. This value is incremented monotonically with each packet in a flow. If a gap in the sequence is detected by the receiving endpoint, then packet loss may be presumed and exception is reported if `reportExceptions` is set to `1`.

When diagramming headers that conform to a published standard, the internal names of fields used within the context of the architecture and design of a forwarding system always take precedence and are generally used throughout a system's specification. However, it is important that the standards-based name also be presented for reliable cross-checking.

The width of every field is explicitly listed in the field definition table.

The location of each field within a structure is specified by indicating its offset from the first bit of the first word of the structure (always counting from zero in a big-endian fashion). This offset is expressed in bytes and bits where the number of bytes is a multiple of 4 and the byte offset is separated from the bit offset by a period.

The definition of each field must be precise and as extensive as necessary for the reader to understand its use, limitations and interactions with other fields. The first sentence of a field's definition is always a sentence fragment instead of a complete sentence.

Style

This section describes the style and naming conventions used in this book. These conventions are meant to encourage meaningful names for things and to be in keeping with modern trends in software engineering.

Abbreviations and Acronyms

As a general rule, abbreviations and acronyms are not used. This rule neatly avoids the problem of abbreviating the same word several ways within the same document. The only acronyms that are generally allowed are those whose Google search results are near the top. So, acronyms such as CPU, RAM and LAN are allowed. However, the capitalization rules for words are applied to acronyms when those acronyms are used in an object's name. For example, a field may be named `requestToCpu`.

Naming Guidelines

All objects are given descriptive names. Names aren't any longer than necessary to convey their intent and to disambiguate them from other, similar names. However, little concern is given to names getting too long. Camel-case (i.e., embedded capitals and no underscores) are used for all name types. Fields, variables, enumerations and the like start with a lower-case letter. Headers, structures, types and the like start with a capital letter.

To help set off named objects and certain numeric values from other paragraph text, these names and values appear in a `monospace typeface`.

7 Forwarding Protocols

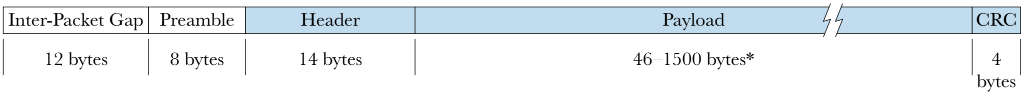
Though there are dozens of header formats in common use, only a handful are routinely used for the actual forwarding of packets. These are:

- Ethernet
- IPv4
- IPv6
- MPLS
- cross-connect

In the several sections of this chapter, the header formats, field definitions and, most importantly, the operational details of these forwarding protocols are presented.

Ethernet

Ethernet is specified by the IEEE 802.3 standard. The Ethernet packet format is shown in Figure 10.



* original standard

Figure 10 Ethernet Packet Format

Compared to other forwarding protocols, Ethernet's packet format has a few interesting characteristics. First, because it is a media-access protocol that is intended to operate at Layer 1 and Layer 2, it has a cyclic redundancy check (CRC) field as a 4-byte packet trailer that protects both the header and payload sections of each packet. It also has two parts that convey absolutely no useful information: the inter-packet gap and the preamble. These exist only in support of Ethernet's legacy media access control (MAC) protocol.

The payload length of an Ethernet packet can range from 46 to 1,500 bytes, at least according to the original standard. Subsequent revisions to the standard have both decreased the minimum payload length and increased the maximum due to expanded headers and a desire to transport more data bytes per packet (less processing overhead per data byte). The overall minimum length (excluding the inter-packet gap and preamble) has remained fixed at 64 bytes while the maximum overall length is the non-standard (and problematic) jumbo packet at 9K bytes. If an Ethernet packet's payload is too short to yield an overall packet length of at least 64 bytes, pad bytes are added between the end of the Ethernet packet's actual payload and the CRC field. It is incumbent upon the Ethernet packet's payload to indicate its own length since the Ethernet packet's length is not a reliable indicator of the payload's length (i.e., the Ethernet packet does not indicate how many pad bytes are present).

The actual Ethernet header occupies 14 bytes and consists of three fields.

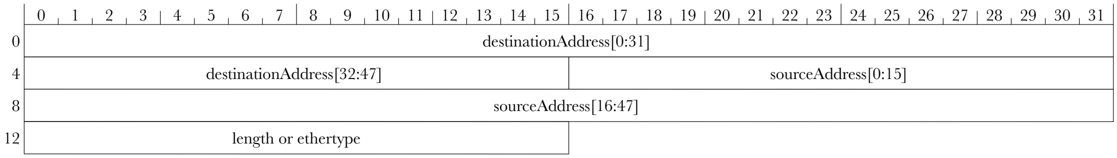


Figure 11 *Ethernet Header Format Diagram*

Table 2 *Ethernet Header Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`destinationAddress` (Destination Address, DA, etc.)	48 bits	0.0	The MAC address of the packet's destination endpoint. Typically an endpoint is issued, a globally-unique, statically-assigned value. An Ethernet header's `destinationAddress` value may be a unicast, multicast, or broadcast value. MAC addresses are depicted in text as six pairs of hexadecimal digits separated by hyphens (e.g., `4c-03-00-12-4e-9a`).
`sourceAddress` (Source Address, SA, etc.)	48 bits	6.0	The MAC address of the packet's source endpoint. A source address must always be a unicast address.
`length` or `ethertype` (Length/Type, EtherType)	16 bits	12.0	The packet's overall length. An Ethernet packet's length is measured from the first byte of the `destinationAddress` field to the last byte of the CRC trailer at the end of the packet. If this field's value is 1,500 or less, then it is interpreted as a `length` value. If this field's value is 1,536 (`0x0600`) or greater, then it is interpreted as a `ethertype` value.

Ethernet's Humble Origins

The official designation for Ethernet is: CSMA/CD. That stands for Carrier Sense, Multiple Access with Collision Detection. That ungainly mouthful of terms describes the nature of Ethernet's media access control protocol.

The original Ethernet was a purely shared-medium network in that every packet transmitted at 3 Mbps (later, 10 Mbps) onto a coaxial cable was visible to all of the endpoints attached to the same cable. The `destinationAddress` field in the Ethernet header ensures that an endpoint can identify itself as a packet's intended receiver and accept a just-received packet. The `sourceAddress` field provides a means for identifying which endpoint sent the packet, enabling two-way communications between the endpoints.

But, if you have a shared medium network with multiple endpoints, how does each endpoint know when to transmit without conflicting with other endpoints on the same cable? CSMA/CD actually works very much like human conversation. Consider a room full of people that have something they want to say. Everyone in the group listens to make sure that no one is talking before they attempt to speak (carrier sense). If silence is detected, one or more people may start speaking without raising their hand or otherwise arbitrating for permission to speak (multiple access). If a speaker hears someone else's voice while they're speaking, they stop speaking (collision detect) and a wait time is chosen. As more and more collisions are detected without successfully getting a sentence out, the bounds on the random wait time increase exponentially. If a maximum number of collisions occur, the attempt to speak fails and the speaker drops that particular sentence. Thus, a bunch of nodes that can listen while they talk are able to share a medium with reasonably fair access.

The requirement to detect a quiet medium is the reason that the inter-packet gap exists. After transmitting a packet or while waiting for another's transmission to end, every node must wait at least 96 bit-times (12 bytes) before starting a transmission. Ethernet packets must be at least 64 bytes in order to allow a transmitting endpoint to detect that its packet has experienced a collision. This requires the packet to propagate the entire length of the Ethernet segment and for the collision indication to propagate all of the way back; all while the source endpoint is still transmitting.

The 64-bit preamble is used to give a phase-lock loop (PLL) function a chance to quickly lock on to the clock encoded in the transmitted data. The 64-bit value transmitted as an Ethernet preamble is: `0x5555_5555_5555_55d5`. The final byte (`0xd5`) is know as the "start of packet delimiter." The start of packet delimiter is required because the number of preamble bits consumed during the PLL lock process can vary quite widely. Hence, counting bits and bytes in the preamble is not possible. The Ethernet preamble appears in serial form as alternating ones and zeros; terminated by a pair of consecutive ones (bear in mind that Ethernet specifies that each byte is transmitted from least significant bit to most significant bit). Hence, the first bit of the preamble is a one and the last two bits are also ones.

At the physical layer, a carrier signal (i.e., a clock) must be integrated with the data in order to facilitate the recovery of packet data even if that data is a long series of ones or a long series of zeros. The original Ethernet standard specified the use of Manchester encoding. This particular encoding scheme—besides being extremely simple—has a very nice property that allows for quick PLL lock-on.

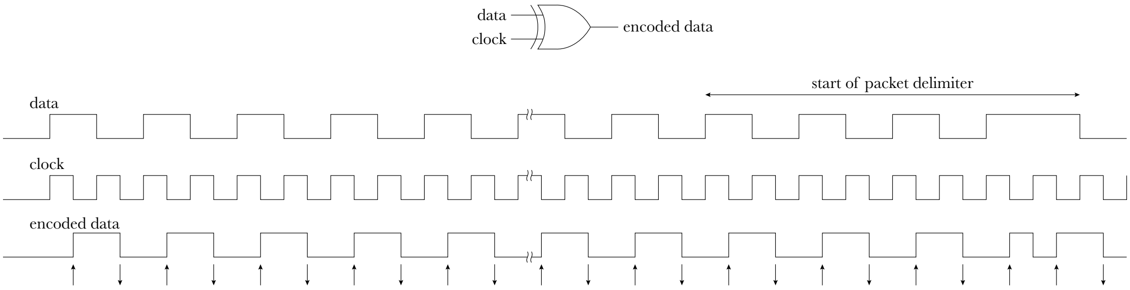


Figure 12 *Manchester Encoding and the Ethernet Preamble*

To encode a serial bit stream using the Manchester encoding technique, it is simply a matter of using an exclusive OR gate on the data and the data's clock. The alternating one-and-zero pattern of the Ethernet preamble yields a steady 5 MHz square wave (for 10 Mbps Ethernet) with a positive-going or negative-going transition in the middle of every bit period. Rising edges of the encoded data represent ones while falling edges represent zeros. As can be seen in Figure 12, the two consecutive ones at the end of the preamble yield two consecutive rising edges. The first bit that follows those two ones is the first bit of `destinationAddress`. Loss of carrier (i.e., the inter-packet gap) is used to terminate each packet.

Modern Ethernet connections no longer depend on CSMA/CD to control media access. Ethernet has long supported dedicated links (i.e., just two link partners on each physical Ethernet connection) and full-duplex operation (i.e., simultaneous transmission and reception). Hence, the only carrier-sense-related behavior that remains in use today is the imposition of a gap between the end of one packet and the start of the next by a single Ethernet endpoint. The preamble is superfluous in an age of continuous carriers. And, there are no collisions on full-duplex networks. However, since there's no compelling reason to change these characteristics, they remain with us today.

Ethernet's relatively low cost, efficient performance and conceptual simplicity led to its eventual widespread adoption. However, the original standard was far from perfect. The coax cable shared medium was awkward to install and prone to failure if the cable was accidentally severed by undoing coupling connectors or rendered inoperative by the removal of the 50-Ohm terminating resistors at either end of the cable. Because of the speed of propagation of the signal down the coax and the attenuation of the signal with distance, the physical extent of an Ethernet network was rather limited. The number of nodes allowed on a segment is limited by the collision backoff algorithm; specifically, a randomly-generated 10-bit number is used to select the backoff period, meaning that, if more than 1,024 endpoints were trying to transmit packets, endless collisions would likely result.

To address the distance limit and the maximum number of nodes allowed on a single Ethernet coax segment, the transparent bridge was invented (discussed in exhaustive detail further below). To address the weaknesses of the shared coax cable, twisted-pair transceivers were developed that could send Ethernet signals down category 3 unshielded twisted pair cables (i.e., office telephone cable) that radiated out from a hub installed in a wiring closet. The simple repeating hub was soon replaced by a multi-port bridge function. The move to twisted pair cabling meant that the transmit and receive signals were now on separate pairs of wire, enabling full-duplex communication. Eventually faster data rates (100 Mbps, 1 Gbps, etc.) followed. Ultimately, the evolution of Ethernet rendered all of the letters in CSMA/CD meaningless.

Ethernet was developed independently of IPv4 and all of the other higher-level protocols that it would one day be called upon to convey packets from endpoint to endpoint. Its initial mission was quite simple: move a string of bytes from one endpoint attached to a coax cable to one or more other endpoints on the same cable. There was, understandably, a bit of shortsightedness in the early Ethernet standard.

Instead of some kind of next-header field, the Ethernet header has a `length` field that indicates the length of the Ethernet packet's payload. Encoding the payload's length into the header is not particularly useful because the MAC (media access control) hardware always indicates the measured total length of each received Ethernet packet. So, at best, the `length` field provides a crude form of error checking or a means of determining a packet's length at the start of packet reception. The lack of a specific next-header field meant that conflicts and confusion were rampant as various applications created ambiguities in their attempts to self-identify their headers by various, incompatible methods.

Eventually, two different means for encoding a next-header value into an Ethernet header were developed. The first was the logical link control and subnetwork access protocol (LLC/SNAP) headers from the IEEE. The LLC header is either 3 or 4 bytes long, immediately follows the Ethernet header's `length` field and is intended to encode the source and destination network access points (i.e., protocols) carried in the Ethernet packet's payload. Unfortunately, only a single byte was set aside to encode these protocol-identifying values. So, before they completely ran out of available codes, the values `0xaa` and `0xab` were reserved to indicate that an additional protocol-identifying header (the 5-byte SNAP) header immediately followed the LLC header. The SNAP header has a 16-bit field—the so-called ethertype field—for identifying an Ethernet packet payload's protocol type.

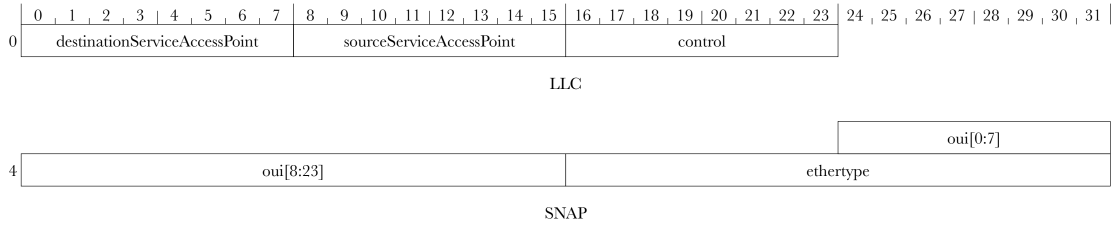


Figure 13 *LLC/SNAP Header Format Diagram*

Table 3 *LLC/SNAP Header Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`destinationServiceAccessPoint` (DSAP)	8 bits	0.0	Destination protocol identifier. For the purposes of an LLC/SNAP header, the value `0xaa` is the only one of any real interest.
`sourceServiceAccessPoint` (SSAP)	8 bits	0.8	Source protocol identifier. For the purposes of an LLC/SNAP header, the value `0xaa` is the only one of any real interest.
`control` (Control Byte)	8 bits	0.16	A demultiplexing value. For the purposes of an LLC/SNAP header, the value `0x03` is the only one of any real interest
`oui` (Organization Code)	24 bits	0.24	The owner of the `ethertype` value. If the reserved value `0x00_0000` is used, then the `ethertype` value is one of the globally-defined, standard ethertype values. Otherwise, it is a private value.
`ethertype` (EtherType)	16 bits	4.16	Identifies the subsequent header or payload.

Typically, `destinationServiceAccessPoint` is set to `0xaa`, `sourceServiceAccessPoint` is set to `0xaa` and `control` is set to `0x03`. If `oui` (OUI stands for organizationally unique identifier and is the most significant 24 bits of an Ethernet MAC address) is set to `0x00_0000`, then the next 16-bit word is a globally-defined ethertype value. Otherwise, if `oui` is non-zero, then `Snap.ethertype` is a private ethertype belonging to the organization identified by the OUI. In summary, if the 48-bit value 0xaaaa03000000 is detected in the six bytes that follow Mac.length, then the 16 bits that follow that six-byte value represent an ethertype value that identifies the protocol associated with the Ethernet packet's payload.

Phew! What a bother.

Fortunately, there's Ethernet II. The DIX consortium (DEC, Intel, Xerox) recognized two important things. First, the length field is not terribly useful since the Ethernet MAC hardware can report the Ethernet packet length upon the completion of the reception of a packet, and that the LLC/SNAP header is a waste of eight bytes. So, what they came up with is elegantly simple and effective: If the length value is less than or equal to 1,500 (the maximum Ethernet payload length) then it is interpreted as a length field. Otherwise, if length is greater than or equal to 1,536 (0x0600) it is interpreted as ethertype. In 1997, the IEEE approved Ethernet II as part of the IEEE 802.3 standard. LLC/SNAP headers are not in widespread use, but they're still out there.

Ethernet Addressing

Ethernet MAC addresses actually have some structure, a single defined bit and a globally reserved value.

The structure of a MAC address is very simple. The most significant 24 bits of the 48-bit address value are known as the Organizationally Unique Identifier (OUI). An organization can purchase an OUI number from the IEEE. The least significant 24 bits of a MAC address are assigned by the owner of the OUI value. Thus, for a single OUI value, up to 16,777,216 Ethernet endpoints may be assigned globally-unique MAC addresses. Organizations can, of course, purchase as many OUI values as they need.

Mac.destinationAddress[7] (the least significant bit of the first byte) is the multicast indicator bit. If this bit is set to 1, then the packet's destination address must be interpreted as a multicast address. Otherwise, it's a unicast address. This bit is in the OUI portion of the MAC address and, because sourceAddress values are not allowed to be multicast addresses, all OUIs are defined with the multicast indicator bit set to 0, but this bit is set to 1 to indicate that a destinationAddress value is a multicast address. Ethernet Multicast behavior is described in the following Ethernet Bridging section, and there is an entire chapter on multicast starting on page 154.

If a packet's destinationAddress value is set to 0xffff_ffff_ffff then it is interpreted as a broadcast address. Note that destinationAddress[7] is set to 1 for a broadcast address, making broadcast a special case of multicast. Broadcast behavior is described in the Ethernet Bridging section, below.

Ethernet MAC addresses are, typically, statically and permanently assigned to each Ethernet media access controller or endpoint at the time of a device's manufacture. This address assignment scheme has the benefits of being extremely simple and robust in that no complex protocols are required for endpoints to either request a dynamic address or self-assign an address that does not conflict with other addresses. The disadvantage of this addressing method is that no inferences can be made about the physical location of an endpoint from its MAC address, nor can it be assumed that nodes within the general vicinity of one another have addresses that fall within some range of values. Thus, when searching for a matching MAC address in a forwarding database, full-width exact matches are generally called for as opposed to range or prefix matches.

Ethernet Bridging

Ethernet bridging has evolved significantly over the years. It was initially developed to solve a couple of simple but significant limitations of early Ethernet networks (i.e., number of endpoints per segment and segment length). As Ethernet has moved from small, local area networks to globally-scoped wide area networks, Ethernet bridging has evolved as needed to accommodate these new applications.

Fundamentals of Transparent Bridging

Ethernet bridging (also known as transparent bridging) was invented to overcome two of Ethernet's limitations when operating on the shared coaxial cable medium: the small number of nodes supported and the limited physical extent of the network. The number of nodes allowed on a shared coax cable is limited for two reasons. First, the collision back-off algorithm specifies a limited range from which random back-off times can be chosen. If the number of active nodes on the network exceeds the upper limit of this range, then it becomes likely that at least two nodes will always choose the same back-off time and collisions will occur ad infinitum. The second reason is simply that each node's throughput decreases as the number of nodes increases. If, for example, you have 100 active nodes on a 10 Mbps shared-medium network, then each of those users will only have access to about 100 Kbps of throughput, even if they're not all trying to send packets to the same destination. The network's physical extent is limited in a coax network by the limits of the collision detection method which is affected by both timing (there's a brief window at the start of packet transmission during which collisions are allowed) and signal attenuation (the amplitude of the signal is a factor in collision detection).

The basic idea behind bridging is to split a single segment that's too long or has too many nodes into a network of two or more smaller segments with bridges connected to each of the segments.

A bridge is, essentially, two or more Ethernet MACs joined together by some specialized logic. A bridge extends the reach of an Ethernet network while isolating its carrier and collision domains to individual segments.

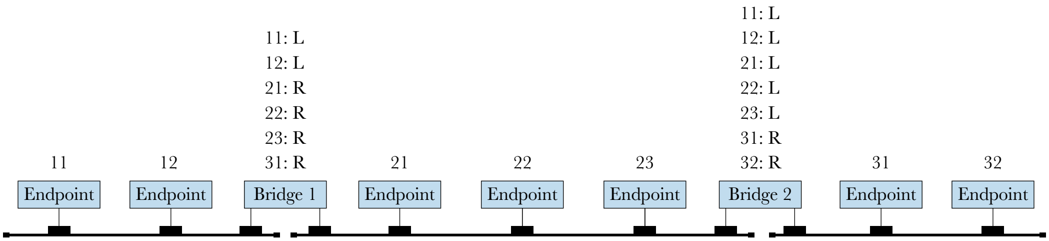


Figure 14 *Coax Network With Bridges*

The specialized bridging logic primarily consists of a forwarding database of MAC addresses associated with interface IDs. The contents of this forwarding database are created automatically and without any special behavior by endpoints or bridges other than the bridge that owns the forwarding database. A bridge accomplishes this neat trick by simply observing the `sourceAddress` values of every packet received by each of its interfaces. When a new `sourceAddress` value is detected, it and the receive interface associated with the packet, form a new forwarding database entry. Finally, a timestamp is associated with each database entry that is updated to the current time value each time a `sourceAddress` value is observed in support of address aging (discussed further below).

Let's work through some forwarding examples.

If Endpoint 11 in Figure 14 sends a packet to Endpoint 12, Bridge 1 receives that packet in the same manner that Endpoint 12 receives it. Upon receipt of the packet, Bridge 1 searches its forwarding database for the packet's `destinationAddress` value. Because that destination address value was previously observed by Bridge 1 as a `sourceAddress` on the left network segment, Bridge 1 knows that the packet's destination is on the left segment and there's no need to forward the packet to the center segment. Therefore, Bridge 1 simply discards the packet. On the other hand, if that packet had been addressed to Endpoint 22, then Bridge 1 would have forwarded the packet to the center segment via its right interface.

Another interesting case is when a packet's `destinationAddress` value doesn't match any of the entries in a bridge's forwarding database. A transparent bridge's policy is quite simple for this case: forward the packet to all network segments except for the one from which the packet was received, ensuring that the packet will eventually reach its intended destination.

Finally, let's consider a case where Endpoint 32 only ever sends packets addressed to Endpoint 31, its immediate neighbor in the same segment. If Bridge 2 had previously learned Endpoint 31's MAC address, then it would never forward Endpoint 32's packets onto the center segment because it knows that Endpoint 31 is on the rightmost segment. That's why Endpoint 32's address is missing from Bridge 1's forwarding database; it's never observed a packet from Endpoint 32. If Endpoint 23 transmits packets addressed to Endpoint 32, Bridge 1 will forward those packets onto the left segment because Bridge 1 has not had a chance to learn Endpoint 32's MAC address. However, once Endpoint 32 replies to Endpoint 23, Bridge 1 will observe Endpoint 32's MAC address as that packet's `sourceAddress` value and add that address/interface association to its forwarding database.

The simple forwarding policy of a transparent bridge works because a packet is forwarded by a bridge if there is a possibility that the packet's intended destination is on the opposite network segment, and only *not* forward it if the bridge is certain that the source and destination nodes of the packet are on the same segment. Bridges rightly err on the side of forwarding packets to too many places instead of too few.

Learning

Ethernet bridges are expected to learn MAC source address values, but they are not required to do so. There is no standard that specifies the rate at which source addresses must be learned or any ratio of observed to learned source addresses. This is so because, fundamentally, flooding packets—the consequence of not learning source addresses—is not an error or failure. Unnecessary flooding simply consumes excess network bandwidth. That's all.

In addition to the learning of new MAC source addresses, a bridge's learning mechanism is also used to detect and account for MAC moves. A MAC move occurs when an Ethernet endpoint is physically moved from one Ethernet segment to another. Until this moved endpoint transmits a packet, any bridge to which it was/is attached will be unaware of this movement and will continue to forward packets to the Ethernet segment where it used to reside. Once this moved endpoint does transmit a packet, however, the bridge to which it is attached will look up its MAC source address in the normal manner. If the moved endpoint was moved from one segment to another segment that is attached to the same bridge, the source address lookup will be successful since that address value had been previously learned on the endpoint's former segment. However, the interface ID associated with that MAC source address will not match the interface ID associated with the just-received packet. All that the bridge has to do to bring its forwarding database up to date is to update the interface ID associated with the MAC source address to that of the interface via which the packet was received.

Very complex policies may be applied to the address learning process. For example, certain receive interfaces may be limited to learning just a fixed number of MAC addresses, or a MAC address may have to exist on some kind of a list of registered MAC addresses in order to be learned. The possibilities are endless and are generally outside of Ethernet-related standards. For this reason, software is generally involved in the address learning process for any reasonably sophisticated Ethernet bridging system.

Aging

Nodes may be added to and removed from a network over time, and the storage capacity of a bridge's forwarding database is finite. Hence, a means is required to purge the forwarding database of obsolete entries. This process is known as "aging." A MAC address ages out of a bridge's forwarding database if that address value hasn't been seen as a `sourceAddress` value in any packets for some substantial period of time. Nominally, the aging period is set to five minutes. A timestamp value stored along with each forwarding database entry is compared to the current time to determine the age of each entry. Entries that have expired are removed from the forwarding database.

Multicast and Broadcast

As described previously in the section on Ethernet Addressing, there exists a special case of the `destinationAddress` value: the multicast MAC address. And there's a special case of multicast: the broadcast MAC address.

Multicast addresses are used when it is desired for a single transmitted packet to be received by a group of endpoints. This is beneficial in two fundamental ways. First, it saves time and resources for the originating endpoint to not have to transmit individually addressed copies of the same packet payload to each of the intended recipients. It's better to let the network do that work instead (this becomes even more powerful of an advantage when we discuss the implications of multi-port bridges, below). Second, the originator of a multicast packet need not know which specific endpoints are supposed to receive the packet. For example, by using a multicast destination address that has a particular meaning (e.g., is associated with a specific protocol or message type), an originating endpoint can simply use that multicast destination address and be assured that all of the endpoints that care about such a message will receive it (barring packet loss, which is always possible).

Multicast addresses cannot be used as MAC source addresses. This is quite reasonable since a multicast source address doesn't make a lot of sense (how can a packet be from multiple sources?). This, of course, means that there's no way for a bridge to add a multicast address to its forwarding database by learning observed addresses. In the normal case, all multicast addresses are treated as "unknown" addresses. And, as you know, packets with unknown destination addresses (i.e., not found in the forwarding database) must be forwarded to all of the interfaces that are *not* the source of the packet, which is also the desired behavior for multicast packets.

There are a variety of network control protocols that allow endpoints to "subscribe" to a multicast flow of packets. By subscribing, all of the bridges between the source and the destination are configured to add a multicast address to the forwarding database. The addresses that are added by some kind of protocol action or administrative action are generally considered static—i.e., they are not subject to aging. Adding multicast addresses to the forwarding database has the effect of pruning Ethernet segments from a multicast address's distribution on which there are no endpoints that are interested in receiving such packets.

The Ethernet broadcast destination address is not just a special case of multicast. It's a degenerate case. The broadcast address doesn't serve any useful purpose that can't be better served by the use of a multicast address. The nature of a broadcast packet is that every endpoint on a bridged network is going to get a copy of that packet even if the majority of the endpoints have no use for the contents of the packet. There's no point in adding a broadcast address to a forwarding database to limit its distribution since, by definition, broadcast packets must go everywhere.

BUM

BUM is an acronym that stands for "broadcast, unknown-unicast, multicast." This term is a useful shorthand because all of these packet types have the same default forwarding behavior: flood.

Flooding and the Forwarding Entity Axiom

As a final word on flooding of BUM packets, it is important to point out how this conforms to the Forwarding Entity Axiom.

The rule for flooding is to forward a copy of the flooded packet to all *available* interfaces. It is important to understand what is meant by "available." A packet's own receive interface is not available. An interface that is blocked or disabled is not available. Finally, an interface that is statically associated with a known forwarding entity is also not available.

Let's take a look at that last point.

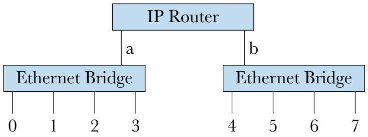


Figure 15 *Bridge + Router Hierarchy*

Figure 15 represents a single forwarding system that consists of three forwarding entities. The connections between the Ethernet Bridge forwarding entities and the IP Router forwarding entity are strictly virtual and do not represent physical interfaces or pathways. When a BUM packet is received by, say, interface 1 of the leftmost Ethernet Bridge forwarding entity, it must flood the packet to all of the available interfaces. The available interfaces are 0, 2 and 3. Interface 1 is not available because that is the interface via which the packet was received. Interface a is also not available because it is associated with an IP Router whose MAC address does not match the `destinationAddress` value of the packet. (The address cannot match because the IP Router forwarding entity's MAC address must be a known unicast address or a specific type of multicast address.) Only packets whose `destinationAddress` value matches that of the IP Router forwarding entity may be forwarded to the IP Router forwarding entity.[3]

This policy has the beneficial side effect of limiting an Ethernet network's broadcast domain (as it is called) to just those Ethernet segments that are associated with a single Ethernet Bridge forwarding entity (i.e., a single Layer 2 forwarding domain).

Loops and the Rapid Spanning Tree Protocol

Multicast packets and, especially, broadcast packets lead to an interesting problem for bridged networks: what happens if a loop is accidentally configured in the network?

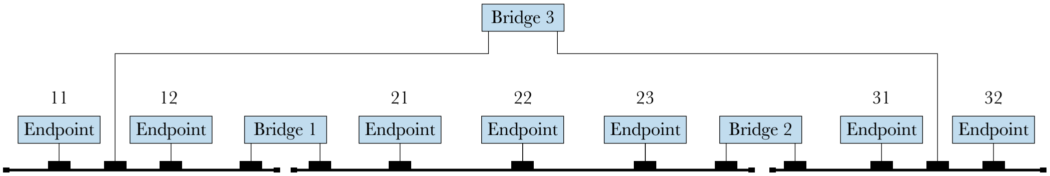


Figure 16 *A Bridged Loop*

Figure 16 shows a loop in a bridged network. A loop exists whenever there is more than one path to get from any point on the network to any other point. When Endpoint 32 transmits a packet to Endpoint 22, it can go two ways to get there: via Bridge 2, or via Bridges 3 and 1. If the packet sent by Endpoint 32 is a broadcast packet, the packet is going to take both paths. However, when these two packets arrive at Endpoint 22, they also arrive at Bridges 1 and 2 via the center network segment. They, of course, forward the broadcast packet up to Bridge 3 and the process repeats itself forever. The network is now fully consumed forwarding copies of the broadcast packet.

Unicast packets also misbehave badly when a loop is present. If both Bridge 2 and Bridge 3 have learned that Endpoint 22 is accessible via their left-facing interfaces,

[3] The behavior of Ethernet and IP multicast forwarding is discussed in detail in Chapter 11: Multicast.

then when Endpoint 31 transmits a unicast packet addressed to Endpoint 22, both Bridge 2 and Bridge 3 are going to forward the packet. Thus, Endpoint 22 ends up receiving two copies of the packet.

The solution to this problem is to have the bridges power-up with all of their ports disabled and for them to cooperatively negotiate a logical tree structure overlayed on top of an arbitrarily complex physical network by only allowing interfaces that are a part of that tree structure to actually forward packets. The original algorithm that performs this work is known as the Spanning Tree Protocol (STP) and was later superseded by the much improved Rapid Spanning Tree Protocol (RSTP).

The steps below are followed by bridges implementing the Rapid Spanning Tree Protocol:

1. Select a root bridge.

   Every bridge as a unique 64-bit ID value that is made up of two parts: a 16-bit priority value concatenated with the bridge's globally unique 48-bit MAC address. The priority value is configurable whereas the MAC address, as per usual, is permanently assigned to each bridge. The priority value occupies the most significant 16 bits of the 64-bit ID value. Therefore, if multiple bridges are assigned the same priority value, their unique MAC addresses are used to break those ties. The default priority value is `0x8000` and must always be a multiple of `0x1000`. The bridge with the numerically lowest ID value serves as the root of the spanning tree. In Figure 16, above, Bridge 1 has the lowest ID number and is, therefore, the root bridge.

2. Determine the least cost paths to the root bridge.

   Every link in a network is assigned a cost value. The cost can be determined by a link's bandwidth, its reliability or its actual dollar cost per bit transmitted. For our simple example network, we'll assume that all of the links are of equal cost. Since Bridge 1 is the root, there's no need to calculate the root path costs for its interfaces. For Bridge 2, its left-facing interface has a cost of 1 since there's just one link between it and the root. Its right-facing interface has a cost of 2 since it must go through Bridge 3 to get to Bridge 1. For Bridge 3, its left-facing interface has a root path cost of 1 while its right-facing interface has a cost of 2.

3. Identify root and alternate interfaces.

   Once the root path costs have been determined, each bridge designates the interface with the lowest cost as its root port. A bridge can only have one root port. An alternate port is simply a root-facing port that's not quite as good as the root port or is just as good but lost a tie-breaker.

4. Identify designated and backup interfaces.

Similarly, each segment (i.e., network link) determines through which of its bridges lies the lowest cost to the root bridge. Network links do not, of course, have any intelligence with which to make such a determination. Instead, the bridges attached to the segment negotiate on the segment's behalf and determine which bridge interface for each segment is going to be the designated interface to carry the segment's traffic toward the root. These bridge interfaces become the designated interfaces. In our example network, the segment that connects Bridge 2 and Bridge 3 has equal costs (i.e., two hops) to get to the root. To break the tie, Bridge 2's lower ID number prevails. This makes Bridge 2's right-facing interface a designated port. Backup ports are ports connected to a bridge that's already connected to the root bridge.

5. Block all other interfaces that lead to the root.

   Finally, every bridge keeps all of the non-root and non-designated ports that lead to the root in the blocked state. All other ports are moved to the forwarding state. Bridge interfaces that cannot possibly face the root (i.e., leaf-facing interfaces in a well-organized tree) are unaffected by the Spanning Tree Protocol.

At the end of the spanning tree network configuration process, every Ethernet segment has exactly one designated interface and every bridge (except the root bridge) has exactly one root interface.

The messages that 802.1D bridges use to pass information to one another are known as Bridge Protocol Data Units (BPDUs). All BPDUs are addressed to the reserved multicast MAC address `01-80-c2-00-00-00`. The MAC address of the bridge's transmitting interfaces is used as a BPDU's `sourceAddress` value. A BPDU is identified as an Ethernet packet's payload with the `ethertype` value `0x0000`. Figure 17 shows the format of a BPDU message.

	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
0	version	type		rootId[0:7]
4	rootId[8:39]			
8	rootId[40:63]			rootPathCost[0:7]
12	rootPathCost[8:31]			bridgeId[0:7]
16	bridgeId[8:39]			
20	bridgeId[40:63]			portId[0:7]
24	portId[8:15]	messageAge		maximumAge[0:7]
28	maximumAge[8:15]	helloTime		forwardDelay[0:7]
32	forwardDelay[8:15]	length[0:7]	length[8:15]	

Figure 17 *Bridge Protocol Data Unit (BPDU) Format Diagram*

Table 4 *Bridge Protocol Data Unit (BPDU) Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`version` (Protocol Version Identifier)	8 bits	0.0	The BPDU protocol version.
`type` (BPDU Type)	8 bits	0.8	The BPDU message type.
topologyChange (Topology Change)	1 bit	0.16	Indicates that a topology change has occurred.
`proposal` (Proposal)	1 bit	0.17	Indicates that the message is a proposal. Proposals must be agreed upon before being acted upon.
`portRole` (Port Role)	2 bits	0.18	Identifies the port role: alternate/backup: 1 root: 2 designated: 3
`learning` (Learning)	1 bit	0.20	Indicates that a port is in the learning mode.
`forwarding` (Forwarding)	1 bit	0.21	Indicates that a port is in the forwarding mode.
`agreement` (Agreement)	1 bit	0.22	Indicates that a proposal has been agreed to.
`topologyChangeAck` (Topology Change Acknowledgment)	1 bit	0.23	Acknowledges a change in topology.
`rootId` (Root ID)	64 bits	0.24	The ID of the root bridge.
`rootPathCost` (Root Path Cost)	32 bits	8.24	The cost to the root bridge.
`bridgeId` (Bridge ID)	64 bits	12.24	The ID of the bridge sending this message.
`portId` (Port ID)	16 bits	20.24	The ID of the port sending this message.
`messageAge` (Message Age)	16 bits	24.8	The age of the message.
`maximumAge` (Maximum Age)	16 bits	24.24	The maximum-allowed age of the message (helps prevent old information from circulating around the network forever.
`helloTime` (Hello Time)	16 bits	28.8	The interval between periodic BPDU transmissions.

Field Name (std. name)	Width	Offset (B.b)	Definition
`forwardDelay` (Forward Delay)	16 bits	28.24	The delay used by STP bridges to transition ports to the Forwarding state.
`length` (version 1) (Length)	8 bits	32.8	Message length. A length of 0 indicates that there is no version 1 protocol information present.
`length` (versions 3 & 4) (Length)	16 bits	32.8	Message length.

IEEE 802.1D specifies four states that a bridge interface may be in. These are:

- `blocking`—Only BPDUs are received and processed.
- `listening`—Receives and processes BPDUs and other information that might affect topology decisions.
- `learning`—Receives but does not forward ordinary Ethernet packets in order to populate the bridge's forwarding database.
- `forwarding`—Normal bridge operation.

All bridge interfaces come up in the `blocking` state. By exchanging BPDUs with other bridges and following the rules specified in the RSTP state diagrams, interfaces are allowed to transition to the `forwarding` state. Since no packets are forwarded in the `blocking` state, the network is assured of never having any loops since the topology of the network is determined before any normal traffic is allowed to flow through the bridges.

BPDUs themselves are immune from loops in the network since they are never forwarded by a bridge. The BPDUs are actually addressed to the bridge and must be terminated by each bridge that receives them. The IEEE has, if fact, defined a number of reserved multicast `destinationAddress` values that are supposed to work this way. These 16 addresses are:

```
01-8c-c2-00-00-0x
```

Bridges are not supposed to forward packets that have `destinationAddress` values that match this range of reserved addresses.

Unfortunately, for a particular class of bridging products, this causes a problem. The IEEE assumes that all bridges are 802.1D compliant, that they support one of the spanning tree protocols and know how to process BPDU packets. However, most of the little 5- and 8-port bridges that are intended for the home or small office do not support spanning tree, but they do obey the IEEE's edict to not forward BPDU packets. This causes a problem because the BPDUs are not processed by these little bridges, but they are also not forwarded. This makes it impossible for an 802.1D-compliant bridge to detect a loop in the network that passes through the little bridge (it's the BPDUs that detect the loops and they must be forwarded by non-spanning tree entities for them to do their job).

The reason that I bring up this little anecdote about these partly-compliant bridges is the importance of not just complying with the letter of a standard, but understanding its context and intent when choosing to implement a subset of a standard. By complying with one part of the 802.1D standard (not forwarding BPDUs, which is simple) and not the whole thing (spanning tree, which is harder), these little bridges create a problem that could have been easily avoided by simply forwarding the BPDU packets in violation of the standard.

Multi-Interface Bridges

All of the bridges that have been described so far have been very simple two-interface systems. To expand to multi-interface forwarding systems is quite simple. Clearly, the forwarding database must indicate the interface with which each MAC address is associated. That's simple enough. Things get a little more complicated when broadcast/unknown-destination/multicast (i.e., BUM) traffic is considered.

Multicast addresses cannot, by definition, be automatically learned by a bridge. They can, however, be statically added to a forwarding database by a number of multicast-oriented protocols. These multicast entries in a forwarding database return a list of intended transmit interfaces. A bridge then simply forwards a copy of the packet to each of the interfaces in the returned list, except for those interfaces that are knocked out for being a source interface or due to their spanning tree state.

Source Interface and Spanning Tree Protocol State Knockout

In keeping with the Forwarding Entity Axiom, a packet's receive interface may never also be its transmit interface. For packets with unicast `destinationAddress` values, it may seem that it takes care of itself because the forwarding database is configured to forward packets toward their destination and a packet's receive interface never gets a packet closer to its destination. However, the forwarding database can be out of date. If an endpoint is moved from one part of a bridged network to another, it is possible for a bridge on that network to receive a packet addressed to another endpoint that is reachable by the same interface on that bridge. The right thing for the bridge to do is to discard the received packet since the packet was already on a network segment via which the destination is reachable. However, if that bridge's forwarding database hasn't re-learned the `sourceAddress` value of the endpoint that's just moved, it'll blithely forward the packet onto its receive interface as directed by the forwarding database. Hence, the importance of source interface knockout. Simply stated, the packet's receive interface on the current bridge instance is removed from a packet's list of transmit interfaces. This is also applied in exactly the same manner to broadcast, unknown-destination and multicast packets.

To comply with whichever variant of the Spanning Tree Protocol that is in use, the spanning tree state of the interfaces must be considered. Packets received by an interface that are not in the `Forwarding` state must not be forwarded. It is permissible for a bridge to receive and process such packets (if they're addressed to the bridge itself), but they must not be forwarded. Furthermore, all transmit interfaces that are not in the `Forwarding` state must be removed from every packet's list of transmit interfaces.

Virtual Local Area Networks (VLANs)

When the spanning tree protocol reduces a rat's nest of network segments into a nice, neat tree structure, potential bandwidth goes to waste as redundant paths (which create loops) are disabled as illustrated in Figure 18.

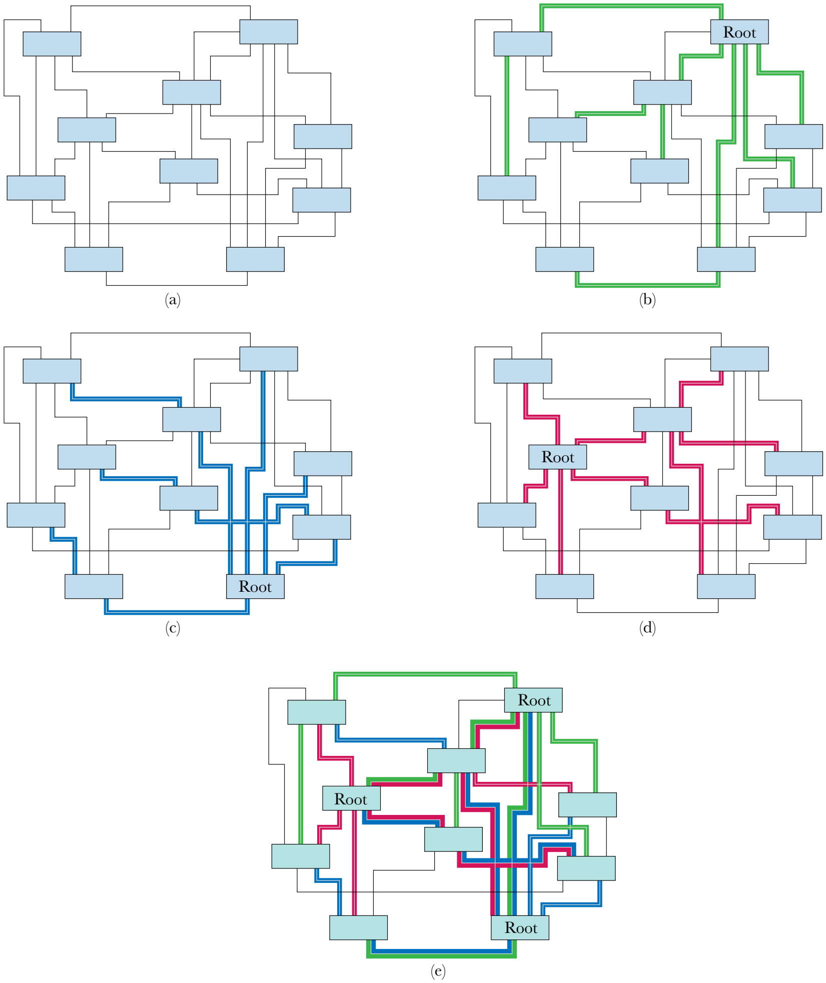


Figure 18 Physical LAN Overlayed With Virtual LANs

Network (a), in Figure 18, shows a complex network of bridges interconnected via Ethernet segments with plenty of redundant paths. There are a lot of ways to get from any point in the network to any other point in the network. In network (b), a spanning tree root has been chosen in the upper right corner and all of the redundant paths have been blocked; the remaining paths being highlighted in green. Networks (c) and (d) show alternate networks based on different root choices. What's common among (b), (c) and (d) is that all of those un-highlighted paths represent wasted potential bandwidth capacity.

One solution to this problem is to divide the network into three separate virtual networks and allow them to operate simultaneously. In (e), above, this is what's done. The green, blue and red networks operate simultaneously, each with its own spanning tree root. There are no loops in any of the virtual networks and there are very few unused paths. Hence, the physical infrastructure is highly utilized. You will note, however, that all three of the virtual networks pass through each of the bridges and several of the links are shared by multiple virtual networks. How is the isolation between these virtual networks established? In short: the VLAN tag.

The VLAN Tag

To enable the isolation of bridged traffic, the VLAN tag was invented and standardized by the IEEE as 802.1Q. A VLAN tag is a very simple thing. It consists primarily of some priority information and a VLAN ID value. And, conceptually, a VLAN is very simple as well: The VLAN ID associated with a packet is used to identify the virtual forwarding entity (i.e., Ethernet bridge) instance within a forwarding system that is supposed to forward the packet. Since each forwarding entity instance has its own private forwarding database and operates completely independently of all of the other forwarding entity instances within the same forwarding system, packets associated with one bridging entity instance can never be forwarded onto a virtual network associated with another bridging entity instance without the intervention of an appropriate intermediary using tunneling behavior.

All very simple, right? Not exactly...

VLANs were invented long after the invention of Ethernet itself and the inventors of Ethernet did not anticipate something like VLANs. In a perfect world, a VLAN ID would be part of a header that precedes the Ethernet MAC header since the VLAN ID modifies how the Ethernet MAC header is interpreted. Alas, Ethernet mandates that the first bytes of an Ethernet packet are always the 14-byte Ethernet header and, for backwards-compatibility reasons, this is unlikely to ever be changed. So, VLAN tags follow instead of precede the Ethernet MAC header.

Ever wonder why VLAN tags are called "tags" instead of "headers"? These so-called tags do, indeed, have all of the characteristics of a header—they convey useful information and they indicate the type of header or information that follows. To help sell the concept of VLANs as an incremental update to Ethernet, a choice was made to depict the VLAN information as something that is *inserted into* an Ethernet header instead of something that is *appended to* an Ethernet header.

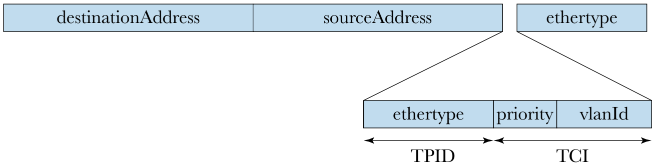


Figure 19 Typical VLAN Tag Depiction

This depiction gives us the somewhat awkward placement of the `Vlan.ethertype` field at the beginning of the tag and, more significantly, it changes the paradigm of the next-header identifying value; rather than identifying what comes next, `Vlan.ethertype` identifies the VLAN tag itself.

This book's preferred depiction is to treat VLAN tags as headers in their own right, appended to an Ethernet header instead of shoved into the middle of one.

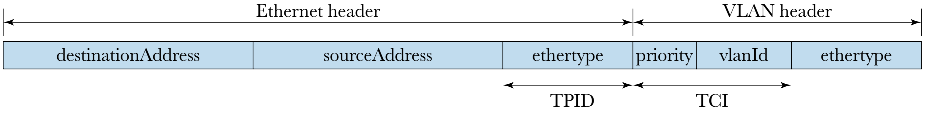


Figure 20 Preferred VLAN Header Depiction

You'll notice that the order of the information in the two depictions (Figures 19 and 20) is exactly the same: a pair of MAC addresses followed by an ethertype that identifies a VLAN tag (the tag protocol ID, or TPID), then priority and VLAN ID values (the tag control info, or TCI) followed by an ethertype value that identifies what follows the VLAN tag. This preferred depiction is merely a cleaner and simpler way of visualizing VLAN headers (or tags, if you prefer). So, after all of that, the details of a VLAN header are shown in Figure 21. VLAN headers are identified, by default, by the preceding header using the ethertype value `0x8100`.

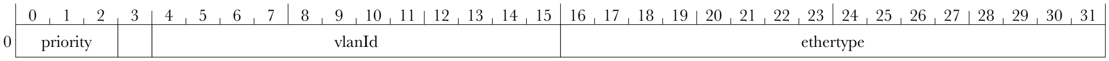


Figure 21 VLAN Header Format Diagram

Adding a 4-byte VLAN tag to a packet does not increase the minimum Ethernet packet length of 64 bytes, but it does increase the maximum from 1,518 bytes to 1,522 bytes. Several other enhancements to Ethernet have pushed the standard maximum length to 2K bytes and non-standard, so-called jumbo packets, are up to 9K bytes long. VLAN header field definitions can be found in Table 5.

Table 5 *VLAN Header Field Definitions*

Field Name	Width	Offset (B.b)	Definition
`priority` (PCP)	3 bits	0.0	The priority code point for the packet. Priority code definitions are network-specific. In other words, a low numerical value does not necessarily imply a low priority. It is permissible (and increasingly common) to combine the `priority` and `dropEligibilityIndicator` fields into a single, 4-bit `priority` code point field.
`dropEligibilityIndicator` (CFI, later DEI)	1 bit	0.3	Indicates that the packet is eligible to be dropped during periods of congestion.
`vlanId` (VID)	12 bits	0.4	The packet's VLAN ID value.
`ethertype` (TPID (in previous header))	16 bits	0.16	Identifies the type of header or payload that immediately follows the VLAN header.

VLAN IDs

Aside from the `priority` value in the VLAN header, the only really interesting information is the `vlanId` value. The 12-bit `vlanId` value allows for a maximum of 4,094 valid VLAN IDs. It's not 4,096 (i.e., 2^{12}) because two of the values are reserved: a `vlanId` value of `0x000` means that the packet does not have an assigned VLAN ID value while a `vlanId` value of `0xfff` is aways interpreted as an invalid VLAN ID. For the `vlanId == 0x000` case, the `priority` and `dropEligibilityIndicator` values are still valid and meaningful. This priority-only behavior was originally defined in IEEE 802.1p and, consequently, a VLAN header whose `vlanId` value is set to `0x000` is known as a "dot one P" header or tag.

VLAN Translation

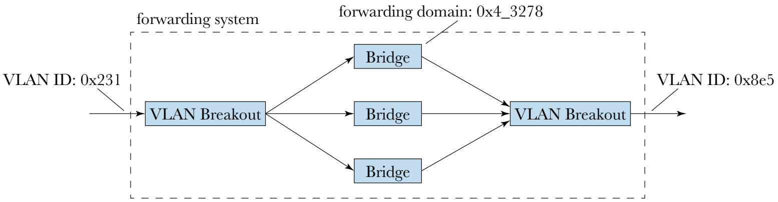


Figure 22 *VLAN Translation*

VLAN ID values as they appear in VLAN headers are merely spatially-relevant tokens that identify a packet's VLAN association on a particular network segment. What this means is that a packet that has a `vlanId` value of, for example, `0x231` when it is received by an Ethernet bridging system may map that VLAN ID to an internal representation (i.e., a forwarding domain) of, say, `0x4_3278`. This new value is used to direct the packet to the appropriate virtual bridge instance (i.e., Ethernet forwarding entity). Notice that the internal representation of the forwarding domain may be much wider than the 12-bit VLAN ID. Just because a single network segment may be restricted to just 4,094 VLANs, this doesn't mean that the total number of virtual bridges that may be active within a large bridging system at any one moment cannot vastly exceed 4,094. Prior to transmission of the packet by the bridge (onto the same VLAN, of course, since we're presuming that the Ethernet tunnel is not terminated at the current bridge), the internal representation of `0x4_3278` is mapped to, say, `0x8e5` for use on the outgoing network segment. Despite the use of three different values used in three different spatial contexts, they all refer to the same VLAN.

Default VLAN

There is also the concept of a default VLAN that must be considered. Each physical or logical interface may be configured with a default VLAN ID. The idea is that, if a packet is received without a VLAN ID (it may still have an 802.1p priority-only header), it is assigned the receive interface's default VLAN. The receive interface may also be configured to discard those packets whose VLAN header's `vlanId` value matches the default VLAN ID of the interface; enforcing an optional restriction that those packets must be received without a VLAN header. Similarly, if a packet's internal VLAN representation maps to the default VLAN ID of the packet's transmit interface, the interface may be configured to strip that VLAN header from the packet, sending the resulting packet as an untagged packet.

Yet another way of dealing with untagged packets is for a receive interface to infer a particular VLAN association for a packet based on that packet's final ethertype; meaning that each protocol conveyed by the Ethernet packets may be mapped to its own VLAN. Ultimately, any aspect of a packet that can be used to identify the flow that a packet belongs to may be used to infer a VLAN association in lieu of using a VLAN header, including, but not limited to, the Ethernet header's `sourceAddress` value.

Private VLANs

Private VLANs were invented to enable isolation between interfaces within a single VLAN, reducing the consumption of scarce VLAN IDs. In a private VLAN, there are three interface types: promiscuous (P), isolated (I) and community (C). As shown in (b), in Figure 23, the promiscuous interface can exchange packets freely with any of the interfaces in a private VLAN (including other promiscuous interfaces). Two communities of three interfaces each are show in Figure 23. In (c), it is shown that interfaces within a community can exchange packets with one another. However, as shown in (d), packets may not be exchanged between interfaces of different communities or between a community interface and an isolated interface. And, as the name implies, an isolated interface cannot exchange packets with any interface other than a promiscuous interface.

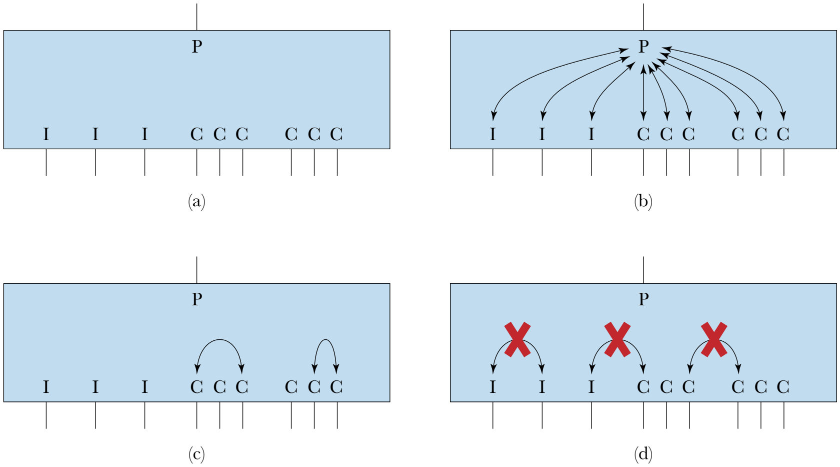


Figure 23 *Private VLANs*

VLANs and the Spanning Tree Protocol

A single Ethernet network can have just one instance of the Spanning Tree Protocol in operation. Such a network can be safely subdivided into multiple VLANs, effectively allowing that single instance of the Spanning Tree Protocol to span (so to speak) several VLANs. It is easy to visualize how this can be done safely. Once the Spanning Tree Protocol has pruned a network to a strict tree structure, it is impossible to introduce a loop into that network by subdividing it into VLANs.

For all intents and purposes, a virtual Ethernet network is indistinguishable from a physical Ethernet network. The same operating rules apply in both cases. This means that a single virtual Ethernet network (i.e., a VLAN) can have exactly one instance of the Spanning Tree Protocol in operation. However, because a single physical Ethernet network can support multiple VLANs, it is possible to have multiple instances of the Spanning Tree Protocol operating on a physical network as long as each one is operating in a separate VLAN or is associated with a set of VLANs that are not associated with any other Spanning Tree Protocol instance.

Running multiple instances of spanning tree across several VLANs is known as Multiple Spanning Tree Protocol (MSTP) and is standardized by IEEE 802.1s. Essentially, Multiple Spanning Tree Protocol allows several VLANs to be associated with an instance of the Rapid Spanning Tree Protocol and for multiple instances of the Rapid Spanning Tree Protocol to operate on a single physical Ethernet network.

Ethernet Tunnels

Provider Bridged Network (aka Q-in-Q)

With just 12 bits of VLAN ID space, there are a number of applications where 4,094 VLAN IDs on a single network segment is a serious limitation. Consider a scenario where a service provider wants to be able to provide private Layer 2 services to a number of customers. Let's presume that each customer maintains a number of VLANs on their networks and they want those VLANs to span from site to site across the service provider's network. As long as the total number of VLANs across all of the service provider's customers' networks does not exceed 4,094, the service provider can map the customer VLAN IDs to VLAN IDs that are only used within the service provider's network without any loss of information and without conflict, mapping them back at the far side of the service provider's network to the customer's VLAN ID numbering space. This does, however, impose severe scaling limitations.

The solution, as originally standardized in IEEE 802.1ad, is to use two VLAN headers. The outer header is known as the service provider tag (or S-tag) and the inner VLAN header is known as the customer tag (or C-tag). The ethertype for the S-tag is, by default, `0x88a8` while the C-tag retains the single VLAN tag's ethertype value of `0x8100`. The S-tag is owned by the service provider and is used to identify a particular customer, confining each customer to its own VLAN within the service provider's network. The C-tag is owned and controlled by the customer and may be used however the customer sees fit.

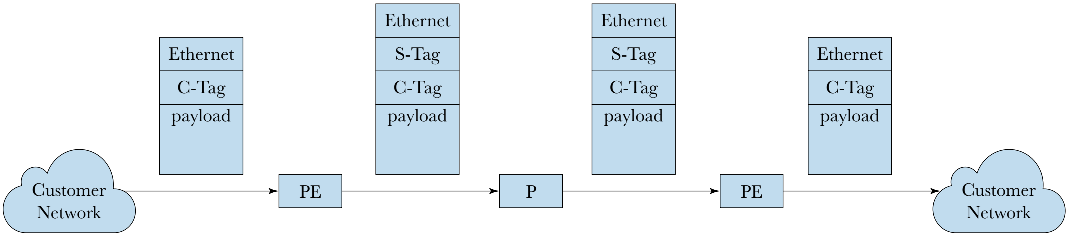


Figure 24 *Q-in-Q Network Example*

In a practical network, bridges at the edge of the provider's network (PE, for Provider Edge) receive packets that have just one VLAN header. The receive interface (which is dedicated to a single customer) adds an S-tag to the packets that identifies the customer associated with that receive interface. This is akin to entering a VLAN tunnel as described in the Tunnels chapter on page 14, but using interface ID information instead of addressing information to perform the mapping.

The bridges in the core of the service provider's network (P) must consider both VLAN tags when identifying which instance of an Ethernet bridge forwarding entity within a P bridging system must forward the packet. This is so because, despite Ethernet MAC addresses supposedly being globally unique, there is no guarantee that a customer doesn't have duplicate addresses in operation across its VLANs. So, just considering the S-tag may expose forwarding ambiguities that wouldn't occur if the customer's VLAN ID values are also considered.

At the far edge of the service provider's network, the S-tag is stripped from the packet as the packet is transmitted onto a customer-facing interface that is dedicated to that customer. Again, very much like classical tunnel exit behavior.

As a variant on the S-tag/C-tag paradigm, it is also possible to treat the two VLAN tags as concatenated tags, with the S-tag (outer tag) providing the most significant 12 bits and the C-tag (inner tag) providing the least significant 12 bits of a resulting 24-bit VLAN ID. This is useful in those applications where what's really needed is a single, very large VLAN ID space instead of a hierarchy of VLANs.

Q-in-Q solves one aspect of the VLAN scaling limitation of the 12-bit VLAN ID value, but it is not a complete solution. First and foremost, there is still a scaling problem. This time it's not due to the narrow width of the `vlanId` field. Instead, it is due to the fact that every P bridge in the service provider's network must now learn all of the MAC addresses of all of the endpoints of all of the customers' networks. This becomes clear when you consider that the only addressing information contained in the Ethernet header and VLAN tags is the `destinationAddress` field from the customer. Hence, the service provider is compelled to have all of its bridges forward based on customer-provided MAC address values and to scale its forwarding databases to accommodate the union of all of its customers' forwarding databases.

Separately, though a new ethertype value was allocated for the S-tag, the original VLAN ethertype value (`0x8100`) was preserved for the C-tag. This means that a forwarding system cannot simply examine the C-tag in isolation and unambiguously determine that the VLAN tag is a C-tag versus a standalone VLAN header. This problem is compounded by the fact that, in some networks, the pre-standard ethertype value `0x9100` and sometimes `0x9200` is used to denote an S-tag instead of the standard `0x88a8` value. The ethertypes that identify VLAN headers must be examined in context in order to be interpreted correctly. A VLAN header's ethertype context is defined by the packet's receive interface and by a VLAN header's preceding VLAN header. For example, simply detecting an ethertype value of `0x8100` is not sufficient to determine that the current VLAN header is the C-tag part of an S-tag/C-tag pair. It is only part of an S-tag/C-tag pair if the preceding VLAN header was an S-tag (according to its associated ethertype value) and the packet's receive interface is configured to interpret these ethertype values appropriately (different receive interfaces may be associated with networks that are independently configured and managed, leading to varying and whimsical uses of ethertype values from interface to interface).

Provider Backbone Bridged Network (aka MAC-in-MAC)

To address the scaling and isolation issues of Q-in-Q, the IEEE standardized the MAC-in-MAC protocol as IEEE 802.1ah. As its colloquial name implies, this standard calls for encapsulating an Ethernet packet inside of an Ethernet packet. The outer Ethernet header is used by a service provider while the inner Ethernet header belongs to the service provider's customers. The significant benefit of MAC-in-MAC over Q-in-Q is that the customer's addressing and VLAN spaces are completely opaque to the core of the service provider's network. This means that the bridges in the core of a service provider's network only need to learn the MAC addresses associated with the service provider's edge bridges, and not the service provider's customers' entire set of network endpoints.

MAC-in-MAC doesn't just jam two Ethernet headers together and call it a day. A service encapsulation header is inserted in between two Ethernet headers in order to provide additional information about the service being provided. This stack of headers is depicted in Figure 25.

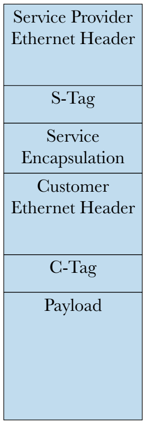


Figure 25 MAC-in-MAC Header Stack

The service provider's Ethernet header is a standard 14-byte Ethernet header whose ethertype is set to `0x88a8` to indicate that an S-tag is present (yes, the same S-tag used in Q-in-Q). The S-tag is used as the VLAN identifier for the service provider's Ethernet network. The ethertype in the S-tag is set to `0x88e7` in order to identify the following service encapsulation header.

The service encapsulation header is unique to MAC-in-MAC (i.e., it's not just another VLAN tag). It provides some priority information, some option-settings flags and a 24-bit service identifier value. The format of the MAC-in-MAC service encapsulation header is shown in the Figure 26.

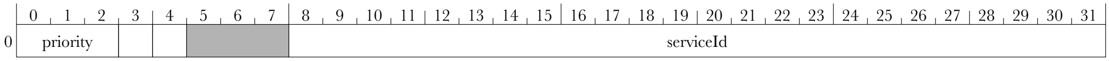


Figure 26 *MAC-in-MAC Service Encapsulation Header Format Diagram*

Table 6 *MAC-in-MAC Service Encapsulation Header Field Definitions*

Field Name	Width	Offset (B.b)	Definition
`priority` (I-PCP)	3 bits	0.0	The priority code point for the packet. Priority code definitions are network-specific. In other words, a low value does not necessarily imply a low priority. It is permissible (and very common) to combine the `priority` and `dropEligibilityIndicator` fields into a single, 4-bit `priority` code point field.
`dropEligibilityIndicator` (I-DEI)	1 bit	0.3	Indicates that the packet is eligible to be dropped during periods of congestion.
`useCustomerAddresses` (UCA)	1 bit	0.4	Indicates that customer MAC address value should be used when multiplexing and demultiplexing service access points (OAM-related).
`serviceId` (I-SID)	24 bits	0.8	The packet's Service ID value. This value can be thought of as a customer ID.

One thing you'll notice right away about the MAC-in-MAC service encapsulation header is that it does not have an ethertype field or any other field that identifies the next header type. This means that the only header type that can ever follow a MAC-in-MAC service encapsulation header is a customer Ethernet header.

The customer Ethernet header is your standard 802.3 Ethernet II header. This header's `ethertype` value may either indicate that a VLAN tag immediately follows (`0x8100`) or that the Ethernet packet's payload immediately follows. To the service provider, this is completely unimportant since everything beyond the service encapsulation header is opaque to the service provider in the core of its network.

The operation of a MAC-in-MAC provider backbone Ethernet network is as one would expect. The usual tunneling behaviors are present. At the edge of the provider's network, customer packets are received via interfaces that are dedicated to individual customers. The identity of the receive interface is mapped to the `serviceId` value used in the service encapsulation header (I-tag). The customer's `destinationAddress` and optional C-tag `vlanId` value are used to perform a lookup into the service provider's edge (PE) bridge forwarding database. A match in the database returns not only the identifier of the interface to use to transmit the packet to the service provider core (P) bridge, but the `destinationAddress` and `vlanId` values to be used in the outer Ethernet and VLAN headers. The `sourceAddress` value of the outer Ethernet header is set to the ingress PE bridge's own MAC address. Once properly encapsulated, the packet is transmitted via the identified interface toward the first provider core bridge.

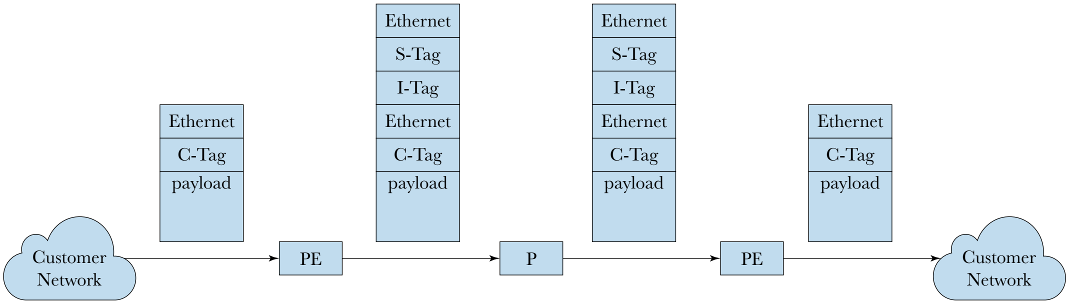


Figure 27 *MAC-in-MAC Network Example*

In the core of the service provider's network, the packet is forwarded normally using just the outer Ethernet and S-tag VLAN headers.

Upon arrival at the egress service provider edge bridge (PE) identified by the packet's outer Ethernet and VLAN headers, the outer Ethernet, outer VLAN and service encapsulation headers are stripped from the packet. Meanwhile, the `serviceId` value from the service encapsulation header (I-tag) and the customer's Ethernet header and VLAN header (C-tag) are used to direct the customer's packet to the correct transmit interface of the egress provider edge bridge (PE).

There is a variety of alternative deployments of provider backbone bridging. For example, since the encapsulated Ethernet packet is a normal Ethernet packet, it is not limited to having just a C-tag. It could, indeed, be double-tagged in the Q-in-Q fashion with both an S-tag and a C-tag. In this case the S-tag's `vlanId` value can be used (along with, or in lieu of, the receive interface ID) to map to the `serviceId` value at the ingress PE bridge (leftmost in Figure 27).

A customer's view of a service provider's MAC-in-MAC network is that of a ginormous, continent-spanning Ethernet bridge, including all of the usual learning, flooding and spanning tree behaviors. To wit, when an ingress PE bridge receives a packet with an unknown customer `destinationAddress` value, the provider's network floods the packet to all of the PE bridges associated with the S-tag's VLAN. The customer's `sourceAddress` value in that packet is learned by the ingress PE bridge and all of the egress PE bridges, associating the customer's `sourceAddress` value with `sourceAddress` value of the ingress PE bridge. Hence, when a reply is sent in the opposite direction, the ingress PE bridge can unicast the packet to the specific egress PE bridge that is attached to the portion of the customer's network where the destination of the customer's Ethernet packet resides. All of the associations between PE bridge interfaces, I-tags and provider S-tags are established administratively.

IPv4

Internet Protocol, version 4 (IPv4) is the protocol that built the Internet. The Internet is a global network and its packets are intended to be forwarded across all kinds of media. IPv4 is not a media access protocol. It does not provide any means for packet encapsulation that is friendly to the physical layer (start of packet delimiters, packet-protecting CRC values, etc.). What makes IPv4 apropos for the Internet is that it includes a number of innovations that lend themselves to operating at a vast scale at low cost, media-type independence and tolerance of unpredictable changes in network topology including temporary loops.

IPv4 packets are typically the payload of an encapsulating Ethernet packet. When this is the case, an ethertype value of `0x0800` is used in the preceding Ethernet or VLAN header. Of course, IPv4 packets may be the payload of a variety of other encapsulating headers, including MPLS, IPv6, IPv4 itself and others.

Figure 28 shows the life cycle of an IPv4 packet as the payload of an Ethernet packet across a simple, but typical network made up of an Ethernet bridge and a couple of IPv4 routers.

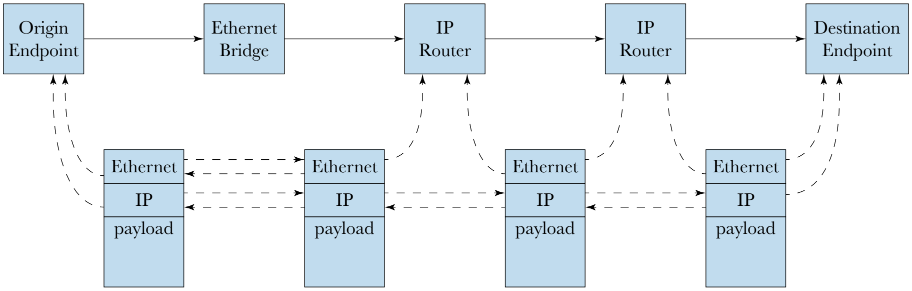


Figure 28 Bridged and Routed Packet Life Cycle

Across the top of Figure 28 are the components of our example network: an origin endpoint, a bridge, two routers and a destination endpoint. Across the bottom are simplified packet diagrams. The dashed arrows emanating from the packet diagrams show to which point in the network each header is addressed; right-facing arrows represent destination address values while left-facing arrows represent source address values. At every stage, the IP header (IPv4 in this case, but the same applies to IPv6) is addressed to the two endpoints, the address values remaining constant all along the path. The Ethernet header's addressing, on the other hand, always points back to the prior IP stage (origin or router) and forward to the next. Thus, the Ethernet header is replaced by each IP router in the path from the origin endpoint to the destination endpoint. This behavior is, essentially, basic tunneling as described in Chapter 4 on page 14. The IP tunnel originates at the origin endpoint and terminates at the destination endpoint. A series of Ethernet tunnels are originated and/or terminated at every hop except for the Ethernet bridge, where the packet is forwarded based on the Ethernet header instead of the IPv4 header.

IPv4 Header Structure and Field Definitions

As shown in the Figure 29, an IPv4 header is considerably more complex than an Ethernet header. This is reasonable since IPv4 is expected to do much more than Ethernet.

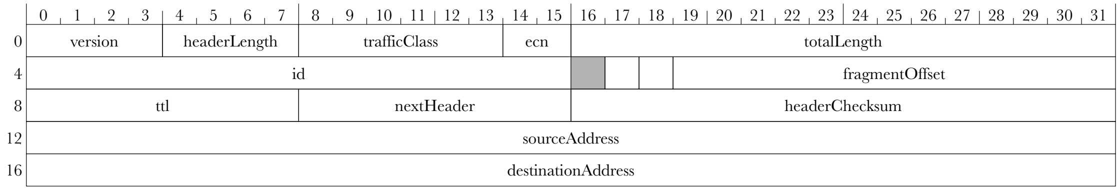


Figure 29 IPv4 Header Structure Diagram

Table 7 IPv4 Header Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`version` (Version)	4 bits	0.0	The protocol version. This value must be set to 4 for IPv4.
`headerLength` (IHL)	4 bits	0.4	The length of the IPv4 header. This value indicates the length of the IPv4 header as measured in 32-bit words. The minimum value for this field is 5 (i.e., 20 bytes). The maximum value is 15 (60 bytes).
`trafficClass` (Type of Service, upper 6 bits)	6 bits	0.8	The traffic class value. Essentially, this is a priority-like value that is used to indicate how the packet must be handled in the face of congestion. See RFC 2474 for details.

Field Name (std. name)	Width	Offset (B.b)	Definition
`ecn` (Type of Service, lower 2 bits)	2 bits	0.14	Explicit congestion notification (ECN). This field is used to convey congestion information to the sources of packet traffic. This value is enumerated as follows: 0 = `notEcnCapableTransport` 1 = `ecnCapableTransport0` 2 = `ecnCapableTransport1` 3 = `congestionExperienced` See RFC 3168 for details.
`totalLength` (Total Length)	16 bits	0.16	The length of the IPv4 packet in bytes. The length of an IPv4 packet is measured from the first byte of the IPv4 header to the last byte of the IPv4 packet's payload (note that, if, say, an IPv4 packet is being conveyed by an Ethernet packet, `totalLength` does not include the Ethernet header(s), padding or CRC). The minimum allowed `totalLength` value is `20` (IPv4 header without options and 0 payload bytes). The maximum allowed is `65,535` (2^{16} - 1). All IPv4-compliant endpoints must support IPv4 packets of least 576 bytes in length.
`id` (Identification)	16 bits	4.0	Packet identifier. Identifies a group of IPv4 fragments belonging to the same, original, unfragmented IPv4 packet.
`doNotFragment` (DF)	1 bit	4.17	Prohibits fragmentation. If this bit is set, then the packet may not be fragmented even if a network segment cannot accommodate the packet's length.
`moreFragments` (MF)	1 bit	4.18	More fragments follow. This field indicates that the current IPv4 packet is not the last fragment of an original IPv4 packet. This bit is always set to `0` for an unfragmented IPv4 packet and for the last fragment of a fragmented IPv4 packet.
`fragmentOffset` (Fragment Offset)	13 bits	4.19	The current fragment's offset. This field indicates the offset of the current IPv4 fragment as measured in 64-bit words relative to the start of the original IPv4 payload. This value is used to place a received IPv4 fragment's payload into the correct position relative to other fragments when reassembling the original IPv4 payload. An unfragmented IPv4 packet and the first fragment of a fragmented IPv4 packet have a `fragmentOffset` value of `0`.

Field Name (std. name)	Width	Offset (B.b)	Definition
`ttl` (Time to Live)	8 bits	8.0	The packet's time to live. This value is decremented by at least one every time its packet is forwarded by an IPv4 forwarding entity (i.e., router). If `ttl` is decremented to `0`, the packet is discarded. If a packet is received with a `ttl` value of `0`, it is discarded.
`nextHeader` (Protocol)	8 bits	8.8	The next header's type. If the type of the next header is known to the current forwarding entity, then that header may be processed. Otherwise, it is likely just treated as opaque payload contents. This field is referred to as a "next header" field instead of "protocol" to better reflect its purpose and to agree with the same field in IPv6.
`headerChecksum` (Header Checksum)	16 bits	8.16	The header's checksum value. This checksum is the 16-bit ones-complement of a ones-complement sum of the 16-bit values that make up the IPv4 header, excluding the `headerChecksum` value itself.
`sourceAddress` (Source Address, SIP, etc.)	32 bits	12.0	The IPv4 address of the packet's origin.
`destinationAddress` (Destination Address, DIP, etc.)	32 bits	16.0	The IPv4 address of the packet's destination.

Addressing

IPv4 addresses are typically depicted in the dotted-decimal style which is four decimal numbers ranging from `0` through `255`, separated by periods. For example: `207.43.0.10`. Setting that rather archaic style aside, an IPv4 address is essentially just a 32-bit number. What really makes IPv4 addressing interesting is the means by which these address values are assigned to network endpoints. Unlike Ethernet MAC addresses which are permanently and statically assigned to endpoints at their time of manufacture (like a person's taxpayer ID), an IPv4 address is generally assigned dynamically in a geographic manner (like a person's postal code). A person's taxpayer ID generally doesn't change over time and remains the same regardless of where they might live within their country. People are, however, free to move about and change their home address, getting a new postal code each time they do so.

So, like postal codes, IPv4 addresses are ephemeral and, also like postal codes, they are hierarchical. If you live on the east coast of the United States, your 5-digit postal code (specifically, a ZIP code in the U.S.) starts with 0, 1, 2 or 3. As you move west, those leading digits get larger and larger. By the time you're in one of the western-most states (California, Oregon, Washington, Alaska, or Hawaii), all of the postal codes start with a 9. Examining subsequent digits of the postal code continues to narrow down the physical location of the associated postal address. The more digits you consider, the smaller the geographic region being represented.

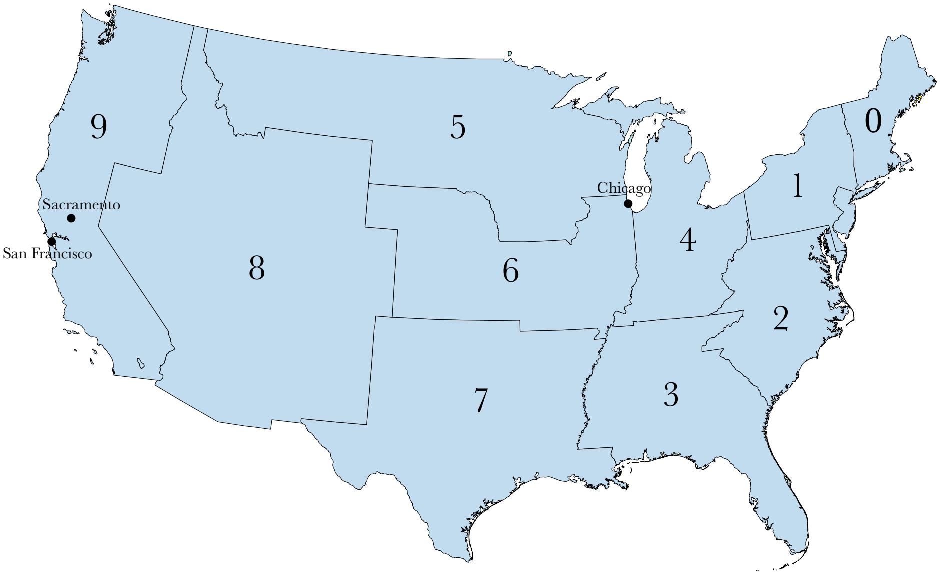


Figure 30 *Map of First Digit of United States Postal Codes (ZIP Codes)*

Okay, so what's the benefit of hierarchical postal codes? Well, if you're sorting mail in, say, Chicago you can place all of the mail whose postal codes start with *1* onto an eastbound airplane and all of the mail whose postal codes start with *9* onto a westbound airplane. Hence, the sorting station doesn't have to maintain a complete list of all possible postal codes just to make a simple east versus west decision. Similarly, IPv4 benefits from hierarchical addressing. By examining just a few of the leading bits of an `Ipv4.destinationAddress` value, an IPv4 router can determine the appropriate interface via which to forward the packet. The router in question may not know exactly where the packet's destination is in the larger network, but that's okay; it doesn't need to know. It just needs to know how to get the packet one step closer to its destination.

The tremendous benefit of geographically-aware, hierarchical addressing is that the IPv4 routers that make up the global Internet—which interconnects billions of endpoints—can fully operate with a forwarding database on each router that is on the order of a million entries. IPv4 routers do this by maintaining forwarding databases of IPv4 prefixes of varying widths instead of full-width host addresses. An IPv4 endpoint is called a "host" and a prefix refers to a "route" in Internet parlance.

An IPv4 forwarding database (aka forwarding information base, or FIB), is a list of address prefixes. Prefix keys have two components: the underlying IPv4 address value and a prefix length value. The prefix length value indicates how many bits (starting with the most significant bit and extending to the right) are valid. This is depicted thusly: `24.201.0.0/16`. The "`/16`" indicates that only the leftmost 16 bits of the 32-bit IPv4 address may be considered when comparing the address value in the forwarding database with the `destinationAddress` value from the packet being forwarded.

A forwarding database does not, of course, consist of prefixes of uniform length. There may be some `/4` entries as well as a bunch of `/24` entries and every other possible prefix length. Endpoint addresses (i.e., `/32`) may also be in the forwarding database. Given that some number of bits in an IPv4 address are ignored during a particular lookup, it is possible for a packet's `destinationAddress` to match multiple entries in the forwarding database. All that's necessary for this to happen is for several prefix entries of different widths to share common values in their most significant bits. Let's return to our postal example to illuminate this.

We know that our Chicago mail sorting facility must send all mail whose postal code starts with 9 on a westbound airplane. The westbound plane lands in Sacramento, in central California, where further digits of the postal code are examined in order to load the mail onto the appropriate trucks. However, let's presume that the postal service sends a lot of urgent mail to San Francisco, so the Chicago office knows to load mail whose postal code start with 941 onto its San Francisco-bound airplane, saving significant time in the delivery of that mail. If a letter is posted in Chicago whose postal code is 94109, it'll match two entries: 9 (go west) and 941 (go to San Francisco). Which entry is the correct one to choose? The 941 entry matches a longer prefix of the 94109 postal code, so it is a more accurate answer than simply matching the first digit. The longest prefix is the best answer. This is known as a longest-prefix match and it is fundamental to IPv4 routing.

When an IPv4 router receives an IPv4 packet, it submits `destinationAddress` to a longest-prefix match lookup within its forwarding database. The longest matching prefix in the forwarding database represents the finest-grained and best option for forwarding the packet. In addition to a longest-prefix match lookup, the IPv4 header is checked and updated (see *Time-to-Live*, page 81, and *Header Checksum*, page 82) and, in the case of IPv4 tunneled within Ethernet, the encapsulating Ethernet header must be stripped off and replaced with a new Ethernet header as required by the rules of tunneling.

Addressing Evolution

Originally, IPv4 had a fixed, 8-bit width for the route portion of an address value, the remaining 24 bits specifying the host (i.e., endpoint). This was quickly shown to not scale very well, so a series of address classes known as Class A through Class E were defined. The prefix length associated with each class was encoded in the first few bits of `destinationAddress` as defined in Table 8.

Table 8 *Classical IPv4 Addressing*

Class	Leading Bits	Prefix Length	Comments
A	0xxx	8 bits	General-purpose unicast.
B	10xx	16 bits	General-purpose unicast.
C	110x	24 bits	General-purpose unicast.
D	1110	28 bits	Multicast
E	1111	-	Reserved for experimental use.

Unfortunately, the Internet started to grow very rapidly with the rise of HTTP and HTML (i.e., web hyperlinks and browsers) in the 1990s. The coarse-grained allocation of IP address blocks to organizations meant that hundreds of millions of IP addresses were not in use by their owners and were not available to others.

Enter classless inter-domain routing (CIDR). With this addressing architecture, prefix lengths were free to span from `/0` through `/32`; enabling fine-grained allocations and freeing up IP addresses that would otherwise be trapped (i.e., allocated but not used). However, there was no longer any reliable correlation between upper address bits and route number width. This meant that we'd have to get much more clever when performing our longest-prefix match lookups. We'll explore lookups (i.e., searching) in Chapter 17 on page 302.

Default Route

In Ethernet bridging, the 802.1D and 802.1Q standards are very clear about what to do if a `Mac.destinationAddress` value is not found in a bridge's forwarding database (specifically, flood the packet to all available interfaces). IPv4 routers, on the other hand, don't experience "entry not found" exceptions when performing destination address lookups. This is because IPv4 forwarding databases must include a single `/0` default route entry. This is a prefix of zero length. It matches every possible `Ipv4.destinationAddress` value. It is also the shortest possible prefix length, so it is only used if every other prefix fails to match the submitted search argument.

The default route specifies the forwarding behavior for all packets that fail to match actual non-zero-length prefixes in the forwarding database. It essentially means, "If you can't figure out what to do with a packet, follow these instructions." Those instructions may specify that the packet be discarded or forwarded to a particular router. This is significantly different from Ethernet's flooding behavior.

Special-Purpose Addresses

IETF RFC 6890 defines a number of IPv4 addresses that serve special purposes. Some of the more interesting special-purpose addresses are listed in Table 9.

Table 9 Special-Purpose IPv4 Addresses

Address	Meaning	RFC
0.0.0.0/8	This host on this network.	RFC 1122
10.0.0.0/8	Private use.	RFC 1918
127.0.0.0/8	Loopback.	RFC 1122
169.254.0.0/16	Link local.	RFC 3927

Options

Ethernet has its tags. IPv4 has its options. They're both a pain in the ass.

IPv4 options are used to convey special information along with the IPv4 header. The presence of an IPv4 option is determined by examining the `headerLength` field. The length of an IPv4 header without options is 20 bytes. `headerLength` is encoded in units of 32-bit words, so 20 bytes is encoded as `5`. Any `headerLength` value greater than `5` indicates that at least one IPv4 option is present.

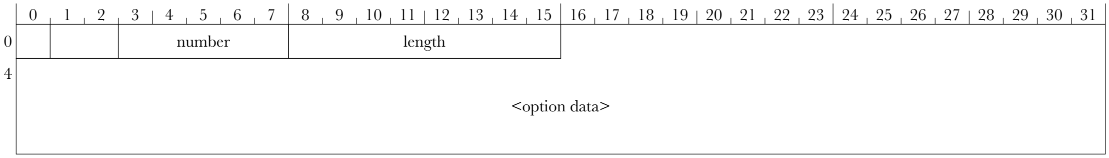


Figure 31 IPv4 Option Format Diagram

Table 10 *IPv4 Option Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`copyOnFragmentation` (copy)	1 bit	0.0	Copy option if packet is fragmented. If this bit is asserted and the packet associated with this option is fragmented, a copy of this option must be a part of all of the fragments' IPv4 headers.
`class` (option class)	2 bits	0.1	Option class. Two option classes are defined: 0 = Control 2 = DebuggingAndMeasurement
`number` (Number)	5 bits	0.3	Identifies the option type.
`length` (Length)	8 bits	0.8	The length of the option in bytes (optional). Not all options require an length value because their length may be inferred from class and number.
option data (Data)	variable	0.16	Option data (optional). Any data associated with an option is conveyed in this field. Not all options require data.

IPv4's 1980s heritage is clear when one looks at the options mechanism. In those byte-stingy times, providing a one-byte option was heroic work.

Sharp-eyed readers will notice that it's possible to have options whose total length is an odd number of bytes, but `headerLength` is expressed in units of 32-bit words. What this means is that up to three pad bytes may follow the last IPv4 option in order to fill out the IPv4 header's length to be a multiple of four bytes.

Making the `length` field optional is, in practice, horrible. It means that any hardware that must support one or more IPv4 options must be able to recognize all of the possible `number` values in order to determine whether or not each option has a `length` field and, if not, what the length of the current option is. This awareness of all of the option types is necessary in order to be able to traverse an IPv4 header's list of options to find the one or more that are of interest. Fortunately, if an IPv4 router is designed or configured to ignore all IPv4 options, then the `headerLength` field may be used to skip over all of the options and go straight to the payload of the IPv4 packet.

The IPv4 options that are available cover functions such as security (obsolete), stream identifier (obsolete), source route (bad idea), record route, and timestamp (not terribly useful). Of these options, the record route option is the only one that may be at all useful. However, it is not at all unreasonable for a modern IPv4 router to not support any IPv4 option functions. This leaves it as a policy decision to determine whether to ignore IPv4 options (i.e., allow them to pass as opaque data) or discard packets whose IPv4 headers have options.

Fragmentation

IPv4 is intended to operate over a variety of link layer (i.e., Layer 2) protocols and technologies and for the link layer type to change from hop to hop as each IPv4 packet makes its way to its destination. Each link layer protocol may impose a different maximum packet length (aka maximum transmission unit, or MTU). To allow its packets to be successfully forwarded through such a hostile environment, IPv4 supports fragmentation—the dividing up of packets into smaller packets that are then reassembled either at their final destination (the usual approach) or after getting past the MTU bottleneck.

Just three fields in the IPv4 header are used to control and manage fragmentation. They are:

- `doNotFragment`
- `moreFragments`
- `fragmentOffset`

If `doNotFragment` is set to `1`, then an IPv4 router is not allowed to fragment that packet. If that packet's length exceeds the MTU of the packet's transmit interface on the router, then the router must discard the packet. However, the packet is not discarded silently. Instead, a message is sent back to the source of the packet informing the source that the packet was too long and could not be fragmented. The Internet Control Message Protocol (ICMP) is the means by which this feedback is delivered to the origin of discarded packets. ICMP is discussed in more detail further along in this section.

The `moreFragments` bit is set to `1` to indicate that the current IPv4 packet is a fragment of a larger original packet, but it is not the last fragment. The 13-bit `fragmentOffset` field is used to indicate the offset of the first byte of the current IPv4 packet's payload relative to the first byte of the first fragment's IPv4 payload. Only the first fragment has an `fragmentOffset` value of `0`. Table 11 summarizes the behavior of `moreFragments` and `fragmentOffset`.

Table 11 IPv4 Fragmentation Control Fields

moreFragments	fragmentOffset	Meaning
`0`	`0`	The packet is not fragmented.
`0`	`> 0`	The packet is the last fragment.
`1`	`0`	The packet is the first fragment.
`1`	`> 0`	The packet is a middle fragment.

The mechanics of IPv4 fragmentation are fairly simple. When a router detects that a packet's length (total length including any necessary encapsulation) exceeds the MTU of the network attached to the packet's intended transmit interface and the packet's `doNotFragment` bit is set to `0`, then the router must fragment the packet. Several strategies may be employed by the IPv4 router in its choice of fragment sizes. It may maximize the size of all of the fragments except for the last one. It may attempt to create equally-sized fragments except, again, the last fragment. It may also create a lot of small fragments in order to avoid further fragmentation by subsequent IPv4 routers. Ideally, however, an IPv4 router generates as few fragments as possible.

Let's consider an example where an IPv4 packet's `totalLength` (IPv4 header bytes plus IPv4 payload) is 4,000 bytes and the network that leads to the packet's next hop has an MTU of 1,500 bytes (i.e., standard Ethernet). The router creates three fragments with the following characteristics.

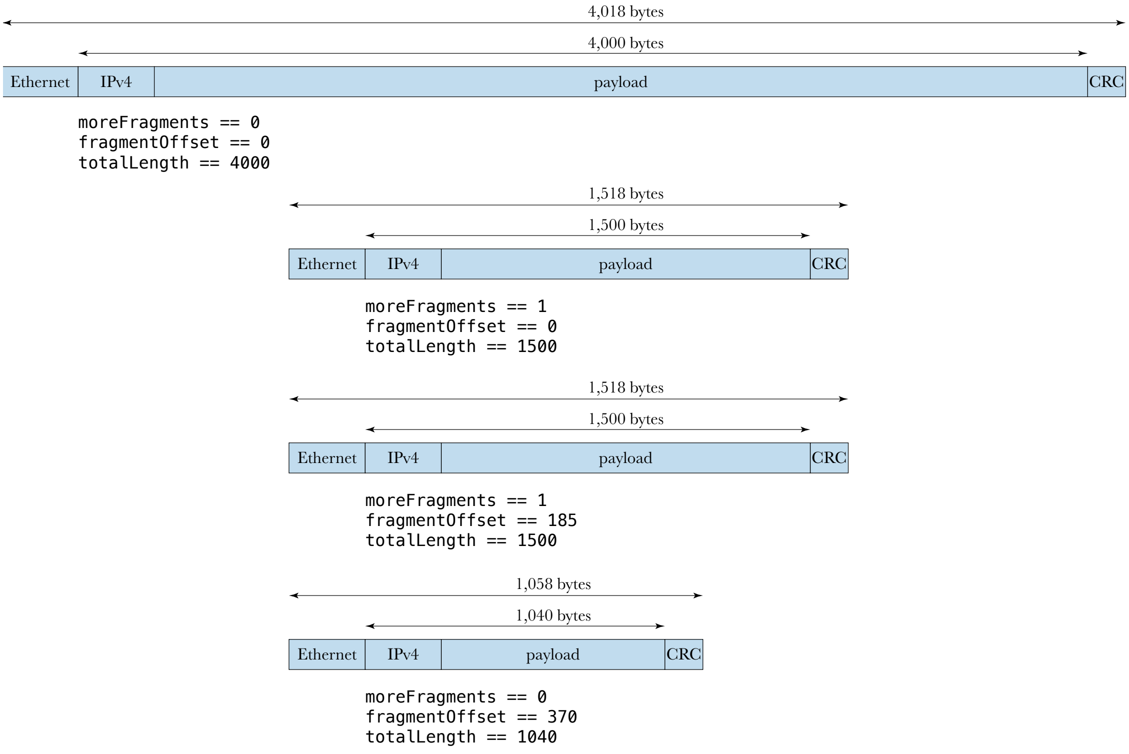


Figure 32 *IPv4 Fragmentation Example*

In our example, the first two fragments are of the same length; the length of the transmit network's MTU. The first fragment's `fragmentOffset` value is `0` whereas the second fragment's offset is `185`. Bear in mind that the offset is in units of 8 bytes and only measures the offset of the pre-fragmentation IPv4 payload. Hence, a `fragmentOffset` value of `185` means that the second fragment is offset from the first by 1,480 bytes, which is 1,500 minus the IPv4 header length of 20 bytes (remember, `totalLength` includes the IPv4 header itself). After two fragments, we've moved 2,960 bytes of the original IPv4 packet's 3,980-byte payload. Hence, the third and final fragment's total length is 1,040, which is 3,980 - 2,960 + 20, and it has a `fragmentOffset` value of `370`, which places the third fragment's payload 2,960 bytes after the start of the first fragment's payload.

The use of a fragment offset value instead of a fragment index value, where fragments are simply given a monotonically increasing value, for example, 0, 1, 2... , means that it is possible for a fragment to be fragmented itself. Indeed, since each fragment is a complete and self-contained IPv4 packet, the process of fragmenting fragment packets may be repeated as often as necessary; it's simply a matter of computing the correct offset to value to make sure that the fragments are reassembled in the right order. If index values were used instead of offset values, fragmenting a fragment would have required fractional index values, a very ungainly and unlikely scenario.

When a packet actually fragments, a single IPv4 packet is used to spawn multiple IPv4 packets. Each fragment has its own link layer encapsulation (e.g., Ethernet) and IPv4 header. The link layer headers of all of the fragments of a packet are typically identical. Each fragment's IPv4 header must, of course, be different. They all have the same addressing and `id` information since they're all going to the same place, but the `fragmentOffset`, `headerLength`[4], `totalLength` and `headerChecksum` fields must all be updated accordingly.

Unfortunately, fragmentation breaks some things and, operationally, isn't at all necessary; it's a remnant from the early days of the Internet that we're pretty much stuck with.

Fragmentation breaks things because, when a packet is fragmented, only the link layer (i.e., Ethernet) and IPv4 headers are reproduced with each fragment. This means that any inner headers that may have followed the IPv4 header now exist only in the very first fragment. Any kind of load balancing or security policy that depended upon access to those headers is now severely compromised. It doesn't work to presume that all of the fragments of a packet will arrive in sequence and with no interstitial packets belonging to other flows. Even the router that performed the fragmentation is not compelled to transmit the fragments as a contiguous sequence of packets. Intermediate routers may, of course, introduce all

[4] The `headerLength` value is subject to change when an IPv4 packet is fragmented because not all of a packet's IPv4 options are necessarily copied to the fragments, changing the length of the IPv4 header from the first fragment relative to all of the rest.

kinds of unrelated packets into a string of packet fragments.

And what about tunneling? Let's presume that a router is transporting IPv4 packets that are, themselves, transporting encapsulated packets of some kind. And let's presume that this mid-tunnel IPv4 router has the temerity to fragment a packet with which it has been charged the responsibility of forwarding. When the packet fragments reach the tunnel termination point, the first fragment will be just fine since the headers from the encapsulated packet are all present. The subsequent packets, on the other hand, are simply conveying the encapsulated packet's payload bytes. There aren't any meaningful headers to be found among the detritus of payload bytes. The tunnel-terminating router's only choice is to reassemble the packet fragments back into the original packet before forwarding. This is a wildly complex and expensive undertaking for a router in the middle of a large network because the millions of packet flows running through such a router mean that the amount of state and packet data that must be maintained in order to perform reassembly is staggering. Reassembly by an endpoint is trivial by comparison.

To prevent all of this confusion and breakdown, IPv4 routers that serve as IPv4 tunnel entrance points must mark the encapsulating IPv4 headers as "do not fragment" by setting `doNotFragment` to `1`.

The very unfortunate part of all this is that fragmentation is completely unnecessary. The IPv4 control protocol, ICMP, has messaging mechanisms built into it that make it possible for an IPv4 router to report not only that fragmentation was necessary, but also the MTU size that made a fragmentation attempt necessary. Hence, a source endpoint can determine what the smallest MTU is along the intended forwarding path and never send a packet requiring fragmentation ever again. All the source endpoint has to do is set `doNotFragment` to `1` for every packet.

Time-to-Live (TTL)

To guard against temporary loops or misbehaving routers, IPv4 packets include a time-to-live value. The original intent of this value was to limit the lifetime of a packet on the Internet, in recognition of the idea that packets not only lose relevance beyond a certain age threshold, but can actually cause problems when a higher-layer protocol responds to an apparent loss of a packet by generating a replacement packet which is eventually met by a much-delayed original. This would be like you going back in time and meeting your past self. As any aficionado of time-travel movies will tell you, this never goes well. A source of an IPv4 packet declares the number of seconds the packet is allowed to live. Each router in the path from the packet's origin to its destination decrements the time-to-live value by the amount of time required to forward the packet, but never by less than one second. In practice, however, IPv4 routers always decrement `ttl` by one, and `ttl` serves as a hop limit counter instead of a time limit.

The policies associated with `ttl` are fairly simple. All IPv4 packets are generated with a non-zero `ttl` value. Every IPv4 router that forwards the packet decrements `ttl` by one. If `ttl` is decremented to zero, the packet is discarded. An IPv4 router or source endpoint must never transmit an IPv4 packet with `ttl` set to zero. If an IPv4 packet is received with its `ttl` value set to zero and that packet is addressed to the router or destination endpoint, that packet must not be discarded; it must be received and processed.

Next Header

IPv4 packets can convey a wide variety of payload types. A couple of common examples include TCP and UDP. In keeping with the general practice of a preceding header identifying the type of the subsequent header, IPv4 includes `nextHeader` as one of its header fields. The definition of this 8-bit field is maintained by the Internet Assigned Numbers Authority (IANA).

Header Checksum

In a belt-and-suspenders move, IPv4 includes a 16-bit header checksum that protects the header despite Ethernet having a 32-bit CRC value that provides better protection for the entire Ethernet packet, including its IPv4 packet payload.

`headerChecksum` is the ones-complement (simple bitwise inversion) of the ones-complement sum (end-around carry) of all of the 16-bit words of the IPv4 header. For the purposes of computing the checksum, `headerChecksum` is either skipped or set to zero during the computation of the ones-complement sum. When checking the checksum upon reception of an IPv4 packet, the `headerChecksum` value is included in the computation of the checksum. An IPv4 header with a correct `headerChecksum` value yields a checksum value of `0x0000`.

Ones-Complement Sums

The `headerChecksum` value depends upon computing a 16-bit ones-complement sum. A ones-complement sum is a sum where the carry bits are added into the sum. For a simple example, consider the ones-complement sum of two 4-bit values: `0xc + 0x9`. A simple twos-complement sum yields a value of `0x15`. The leftmost digit of this sum is the carry value (i.e., `0x1`). Adding `0x1` to `0x5` yields `0x6`. This is the ones-complement sum of `0xc` and `0x9`. If adding the carry bits to the sum produces a carry of its own, then that carry is added to the sum as well.

When an IPv4 header is updated, as often happens when, for example, `ttl` is decremented or fields are updated when fragmenting a packet, `headerChecksum` must be updated as well. This can be done either by computing an entirely new `headerChecksum` value, or it can be done incrementally as follows:

```
HC' = HC + ~m – m'
where
HC = old header checksum
HC' = new header checksum
```

```
m = old value of a 16-bit field
m' = new value of a 16-bit field
```

See IETF RFC 1624 for more details.

Internet Control Message Protocol

The Internet Control Message Protocol (ICMP, IETF RFC 792) is used to send messages for diagnostic or control purposes or to report errors from the point where the error occurred to the source of the packet experiencing the error. A couple of examples of error messages are “time-to-live expired” and “fragmentation required with `doNotFragment` set to `1`.”

ICMP messages are encapsulated in IPv4 packets and use the `Ipv4.nextHeader` field value `0x01`. Figure 33 depicts the format of an ICMP message header.

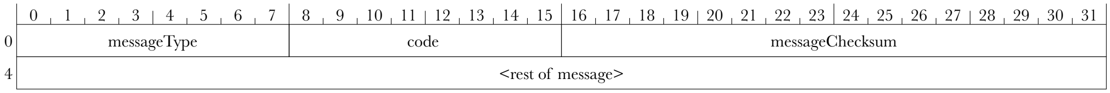


Figure 33 *ICMP Message Header Format Diagram*

Table 12 *ICMP Message Header Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`messageType` (Type)	8 bits	0.0	The message’s type. The `messageType` and `code` values are combined to determine the specific type of the message.
`code` (Code)	8 bits	0.8	The code for a particular message type. This value demultiplexes the `messageType` value to arrive at a specific message type.
`messageChecksum` (Checksum)	16 bits	0.16	A checksum computed over the entire message. This checksum is the same type of ones-complement checksum as `headerChecksum`. For `messageChecksum`, however, it is computed over all of the fields described in this table as well as all of the 16-bit words that make up the data portion of the ICMP message that follows the ICMP message header.
<reset of message>	32 bits	4.0	An optional 32-bit value that is message-type specific.

There is a data section that follows every ICMP message header. The data section is the IPv4 header from the packet that spawned the ICMP message followed by the first eight bytes of that IPv4 packet’s payload.

Network Address Translation (NAT)

Network address translation was developed with the best of intentions and it does have some practical benefits. Unfortunately, it breaks a lot of things and it has been used for years as a security crutch that has delayed the proper hardening of networking protocols and operating systems.

As the name implies, network address translation is used to translate network addresses: specifically, IPv4 addresses. Why would one want to do this? There are actually a number of cases where there are practical benefits to doing so. For example, when connecting two networks that use private IPv4 addresses (i.e., not routable on the public Internet) and have overlapping address assignments (i.e., not every endpoint has a unique address in the newly combined network), network address translation can be used to map the private addresses in one part of the network to non-conflicting addresses for use in the other network.

For example, a `destinationAddress` value of `10.0.0.110` in a packet that originates in network A may be translated to `10.0.0.10` as it transitions from network A to network B. The source endpoint's `sourceAddress` must be similarly translated from `10.0.0.2` to `10.0.0.102` in order to avoid address space conflicts.

The packet's destination endpoint replies to this packet using `10.0.0.102` as its packet's `destinationAddress` value. The network address translation function that acts as an intermediary between network A and network B performs a reciprocal address translation of `destinationAddress` from `10.0.0.102` to `10.0.0.2`, ensuring that the packet reaches its destination in network A.

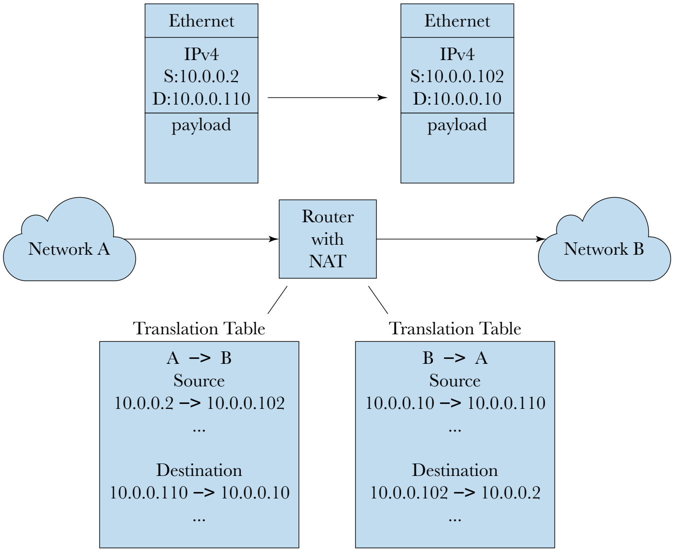


Figure 34 Network Address Translation (NAT) Example

One of the most widespread uses of network address translation is actually network address *and* port translation (NAPT, sometimes known as NATP). This variant of NAT is used in most residential Internet routers. It enables multiple IPv4 endpoints in a private network to access the global Internet while consuming just a single public IPv4 address. In other words, as far as all of the endpoints on the Internet are concerned, the router that connects the private network to the public Internet presents just a single IPv4 address. It is fairly simple to see how, for example, the private `sourceAddress` values `10.0.0.1`, `10.0.0.2` and `10.0.0.3` could all be translated to, say, `145.34.200.153` when forwarding packets from the private network to the public Internet. But, when a response from an endpoint on the public Internet comes back that's addressed to `145.34.200.153`, how does the NAPT function in the router determine to which endpoint on the private network to forward the packet? That's where the P in NAPT comes in.

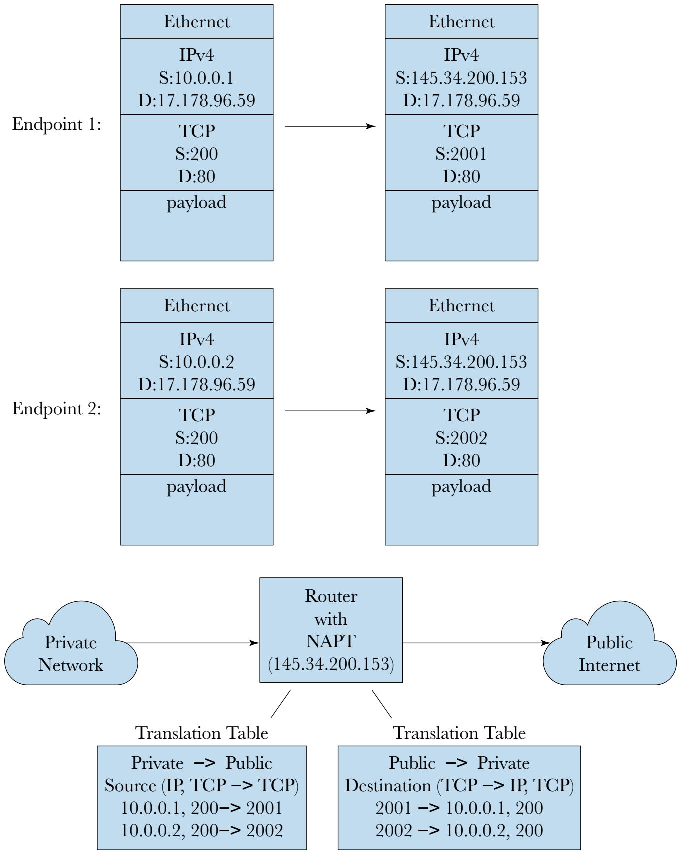


Figure 35 *Network Address and Port Translation (NAPT) Example*

Packets sent from the private network to the public Internet are pretty much universally TCP or UDP packets. Meaning, the payload of the IPv4 packets have 16-bit port numbers that are used to address specific functions or applications within an endpoint (e.g., to distinguish between a web server or email server that may be running on the same endpoint and, consequently, sharing an IPv4 address). When an endpoint on the private network originates such a packet, it also populates the `Tcp.sourcePort` (or `Udp.sourcePort`) field, typically with the same value that it placed in the `Tcp.destinationPort` (or `Udp.destinationPort`) field, though such symmetry is not required. The NAPT function in the router that maps a private endpoint's `sourceAddress` from `10.0.0.1` to `145.34.200.153`, also maps `sourcePort` from `200` to, say, `2001`. Meanwhile the same kind of packet coming from a different endpoint in the private network has its `Ipv4.sourceAddress` translated from `10.0.0.2` to `145.34.200.153` (the same as the other endpoint) but has its `sourcePort` translated from `80` to `2002`, for example. When the IPv4 endpoint on the public Internet replies to these packets, it uses the received `sourcePort` values as `destinationPort` values. This provides a means for the NAPT function in the private network's Internet-facing router to unambiguously map the shared `145.34.200.153` `destinationAddress` value to either `10.0.0.1` (based on a `destinationPort` value of `2001`) or `10.0.0.2` (based on a `destinationPort` value of `2002`) and to map the `destinationPort` numbers to their original `sourcePort` values (`200` and `80`). Sharing IPv4 addresses in this manner has contributed significantly toward delaying the utter exhaustion of IPv4 addresses.

One particular characteristic of NAPT has made it a rather popular but poor security tool: an endpoint on the public Internet cannot initiate a conversation with an endpoint on a private network that's behind a router using NAPT. An external endpoint has no idea what combination of IP destination address and TCP or UDP destination port is going to match a mapping maintained by the NAPT router. These external endpoints must wait for the endpoints on the private network to contact them. When this happens, the external endpoint knows the right values to use when formulating a reply and the NAPT router has also had a chance to establish an appropriate mapping table entry. By hiding an endpoint's IP address behind NAPT, these private networks could operate with a presumption of relative safety since the private network was essentially unreachable from the outside world. Predictably, this crude form of security-by-obscurity was easily overcome.

There are, unfortunately, a number of protocols that are broken by network address translation. For example, it is oftentimes desirable for an endpoint to be able to react to a conversation that is initiated by an endpoint somewhere on the public Internet—an incoming voice over IP call, for example. The use of NAT generally means that some kind of agreed-upon intermediary system is required to put such a conversation together. Also, it turns out that a number of protocols put the IP address of the initiating endpoint in the message body with the intent that a response to the message be addressed to the IP address contained in the message body. A NAT or NAPT function must interpret such a message and perform the translation within the message body as well. The file transfer protocol

(FTP) represents a particularly tricky case. It encodes the IP address and TCP port number in ASCII in the message body. Not only do bytes in the message body have to be updated by NAT, the message length may also have to change since the ASCII representation of an IPv4 address (i.e., dotted-decimal) uses one byte to encode the value of, say, `5` and three bytes to encode the value of, say, `210`. These are solvable problems—and RFCs exist that propose solutions—but they're complex and a bit fragile.

The Nuts and Bolts of Network Address Translation

Network Address Translation requires that a router (an IPv4 router, specifically) maintain a mapping table that is used to map IPv4 addresses (or IPv4 address and TCP/UDP port numbers) from one numbering space to another. Most often, one-to-one mapping is called for. However, many-to-one mapping is also sometimes used.

For one-to-one NAT, a `destinationAddress` value is used as a search argument into the mapping table and a `destinationAddress` is returned as the lookup result; the returned value is then substituted into the IPv4 header. The same is repeated for `sourceAddress`.

For one-to-one NAPT, an `Ipv4.sourceAddress/Tcp.sourcePort` pair is used as a search argument and a `Tcp.sourcePort` number value is returned as the lookup result, the returned `Tcp.sourcePort` number is substituted into the packet and an `Ipv4.sourceAddress` that belongs to the NAPT router is substituted into the packet's `sourceAddress` field. In the reverse direction, the packet's `destinationPort` number is used as a search argument and an `Ipv4.destinationAddress/Tcp.destinationPort` pair is returned for substitution into the packet.

Of course, any changes to fields in the IPv4 header require an update to its `headerChecksum` field.

There is a lot more to network address translation than can be captured in just a few paragraphs. Network address translation was initially described in IETF RFC 1631 (obsoleted by RFC 3022) and is further described by RFCs 2662 and 4787.

Virtual Router Redundancy Protocol (VRRP)

A lot has been written in the preceding pages about how to get a packet to its next hop, and the algorithms for identifying the best next hop are explored in Chapter 19 on page 321. But, what about getting a packet to its first hop? Endpoints such as notebook computers are excluded from the routing protocols that map out a network and identify the most efficient paths through that network. Endpoints rely, instead, on being assigned the IP address of a so-called default router. This router then becomes the first hop for all packets transmitted by the endpoint.

This, of course, introduces a rather obvious single point of failure; if that router fails, then that endpoint is cut off from the network even if other routers are reachable on the same Ethernet network (i.e., accessible through Ethernet bridging). The

Virtual Router Redundancy Protocol (version 3 is described by IETF RFC 5798) is intended to solve exactly this problem.

Conceptually, it's quite simple. Two or more routers cooperate by exchanging information such as which one is the master, the addresses of the routers, and their health. Endpoints that need access to the Internet are configured to use reserved Ethernet MAC addresses to reach their first-hop router (i.e., their default router). Any particular physical router instance may be configured to respond to a number of addresses within this range. If a physical router should fail, this event is detected by the other routers in the group and the surviving routers are immediately configured to respond to the Ethernet MAC addresses previously associated with the just-failed router. Thus, the hapless endpoint continues to use the same MAC address and is none the wiser that its first-hop routing is being performed by a different physical router.

The reserved MAC address range is `00-00-5e-00-01-xx` (for IPv4) and `00-00-5e-00-02-xx` (for IPv6). The `xx` portion of the reserved MAC address indicates that up to 255 virtual router may be configured on the same Ethernet network (only the values 1–255 are permitted to substitute for `xx`).

VRRP applies to both IPv4 and IPv6. It is described in the IPv4 section simply as a matter of convenience.

IPv6

When it became clear that IPv4's 32-bit address fields were insufficient to handle the long-term growth of the Internet, the IETF set about designing its replacement. In addition to widening the address fields, the IETF also took the opportunity to clean up and simplify the IP header. The result of this effort is IPv6. Its header structure is shown in Figure 36.

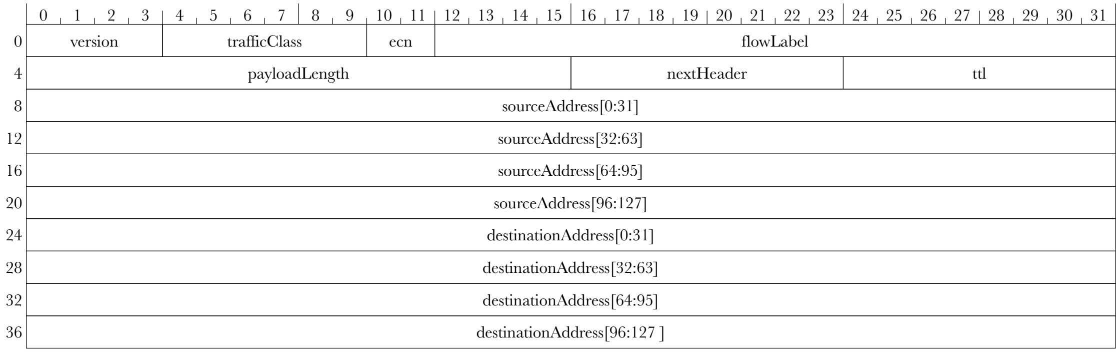


Figure 36 IPv6 Header Structure Diagram

Table 13 IPv6 Header Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`version` (Version)	4 bits	0.0	The protocol's version. This field must be set to `6` for IPv6.
`trafficClass` (Traffic Class, upper 6 bits)	6 bits	0.4	The packet's traffic class value. Essentially, this is a priority-like value that is used to indicate how the packet must be handled in the face of congestion. See RFC 2474 for details.
`ecn` (Traffic Class, lower 2 bits)	2 bits	0.10	Explicit congestion notification. This field is used to convey congestion information to the sources of packet traffic. This value is enumerated as follows: 0 = `notEcnCapableTransport` 1 = `ecnCapableTransport0` 2 = `ecnCapableTransport1` 3 = `congestionExperienced` See IETF RFC 3168 for details.
`flowLabel` (Flow Label)	20 bits	0.12	A flow-identifying value. A non-zero value in this field indicates that the packet has been marked with a flow ID value. It is expected that all packets belonging to the same flow—as identified by the 3-tuple: `sourceAddress`, `destinationAddress`, `flowLabel`—all receive the same treatment.
`payloadLength` (Payload Length)	16 bits	4.0	The number of bytes in the payload. This value indicates the length of the IPv6 packet's payload, which is exclusive of the 40 bytes that make up the IPv6 header. Any IPv6 extension headers that may exist between the IPv6 header and its payload are included.
`nextHeader` (Next Header)	8 bits	4.16	The type of the next header or payload. This protocol-identifying field does not necessarily identify the type of a transport-layer protocol such as TCP or UDP. It could just as well identify that the next header is an IPv6 extension header.
`ttl` (Hop Limit, TTL, etc.)	8 bits	4.24	The number of hops that the packet is allowed to make. Referred to in the standard (IETF RFC 2460) as a hop limit value, it serves exactly the same purpose and has the same semantics as `Ipv4.ttl`.
`sourceAddress` (Source Address, SIP, etc.)	128 bits	8.0	The address of the originator of the packet.
`destinationAddress` (Destination Address, DIP, etc.)	128 bits	24.0	The address of the packet's destination.

When comparing the structure of the IPv6 header relative to the IPv4 header, it is abundantly clear that, though larger, the IPv6 header has considerably fewer fields (14 for IPv4 vs. 9 for IPv6). One of the IPv4 fields that's missing from IPv6 is a field that indicates the length of the IPv6 header. This is because the IPv6 header is fixed in length at 40 bytes. Where IPv4 has options that are *added* to and become part of the IPv4 header, IPv6 has extension headers that are *concatenated* to the IPv6 header as standalone headers in their own right. You'll also notice that all of the fragmentation-related fields from IPv4 are gone. Indeed, IPv6 routers don't perform fragmentation. IPv6 endpoints (originators and destinations of IPv6 packets) are entirely responsible for fragmenting packets prior to their initial transmission and for the reassembly of the fragments after their final reception. Fragmentation information in IPv6 is conveyed by an extension header.

Ethernet Encapsulation

IPv6 packets are commonly the payload of Ethernet packets. When this is done, the final ethertype of the Ethernet packet is set to `0x86dd` to indicate that the next header is IPv6. Two separate ethertype values are used for IPv4 and IPv6 even though they have different `version` values. This is so because a lot of early IPv4 routers failed to check `Ipv4.version` to ensure that it was set to `4`. Please, everyone, fasten your seatbelts and check your version numbers.

Addressing

Wanting to never run out of IP addresses again, the IETF gave IPv6 128-bit addresses. That's not only large enough to give every person on the planet hundreds of IP addresses, it's enough to give every atom of every person its own IP address. I think we'll be fine for a while.

Address Depiction

IPv6 addresses are typically depicted as a series of eight 16-bit hexadecimal numbers separated by colons. Leading zeros may be omitted from each number. Since it is common for IPv6 addresses to have long strings of zeros, it is also permitted to represent an arbitrary number of all-zero, 16-bit words by simply representing them as two consecutive colons. The number of zeros represented by the two consecutive colons is simply the number of zeros required to form a 128-bit address value. Two consecutive colons may only appear once in an IPv6 address value. Here are some examples of IPv6 addresses:

```
fedc:ba98:7654:3210:1111:2222:3333:4444
4002:0:0:0:8:80:33de:7ef2
200d::238a
::1
::
```

Address Prefixes

Like IPv4, IPv6 relies upon prefixes and longest-prefix matching lookups for packet forwarding. Prefixes are depicted by appending `/n` to the end of an IPv6 address value where `n` represents the size of the prefix in bits:

```
7002:ef84::/32
```

Address Types

There are a few IPv6 address types that are of note. An address's type is indicated by its most significant bits.

Table 14 *IPv6 Address Types (Partial)*

Prefix Bits	Meaning
001	Aggregatable global unicast
1111_1110_10	Link-local unicast
1111_1110_11	Site-local unicast
1111_1111	Multicast

As shown in Table 14, the term "aggregatable" means that a multitude of different routes may be represented by a prefix value that is common among those routes that share a common next hop destination. In other words, if the first, say, 20 bits of route-identifying bits for several forwarding database entries are the same, and those entries indicate that matching packets must be forwarded in the same manner to the same transmit interface, then those forwarding database entries may be collapsed to a single entry with a mask width of `/20`, even though bits beyond the 20th bit of those forwarding database entries differ from one another. This is, of course, normal IP longest-prefix matching behavior.

Subnets vs. Interface IDs

The IPv6 addressing architecture standard stipulates that unicast addresses must have a 64-bit interface identifier. The interface identifier occupies the least significant 64 bits of the 128-bit IPv6 address value. The remaining bits of a unicast IPv6 address—i.e., the most significant 64 bits—serve as a subnet prefix. It is the subnet prefix that an IPv6 router is supposed to use to actually forward an IPv6 packet onto the appropriate network segment. The subnet prefix value is subject to longest-prefix match lookups and forwarding. All of this means that an IPv6 router only needs to perform a longest-prefix match on the most significant 64 bits of `destinationAddress` and may safely ignore the least significant 64 bits.

Unfortunately, in practice, longest-prefix matches on the full 128 bits of an IPv6 destination address are often required. It is required for practical purposes because there may be IPv6 endpoints directly attached to their own dedicated interfaces of an IPv6 router. The configuration protocols may not assign each of these directly-attached endpoints their own subnets, meaning that the most significant 64 bits of the IPv6 addresses of more than one of these directly-attached endpoints may hold the same value, making it impossible for the router to forward the packets to the correct interface without considering the entire 128 bits of the packets' `destinationAddress` value. This could, of course be accomplished by a 128-bit exact-match lookup, which is typically far less expensive to perform than a longest-prefix match lookup. Of course, if the 128-bit exact-match lookup failed to find a match, a longest-prefix match lookup is required to resolve the packet's destination, and the time and resources required to perform both lookups must be budgeted into the hardware design. Further complicating what should be a nice clean picture is that some influential operators of networks (and, hence, purchasers of equipment) insist on moving that dividing line between subnet address and interface ID to the right, increasing the required width of the longest-prefix match lookups. It is often best to just support full-width, 128-bit longest-prefix match lookups despite their higher cost.

Special-Purpose Addresses

IETF RFC 6890 defines a number of IPv6 addresses that serve special purposes. Some of the more interesting special-purpose addresses are described in Table 15.

Table 15 Special-Purpose IPv6 Addresses

Value	Description	Reference
`::1/128`	Loopback	RFC 4291
`::/128`	Unspecified address	RFC 4291
`64:ff9b::/96`	IPv4-IPv6 Translation	RFC 6052
`::ffff:0:0/96`	IPv4-mapped address	RFC 4291

Extensions

IPv6 extension headers are used to handle a variety of exceptional cases in IP routing. These include fragmentation, hop-by-hop options, authentication, mobility and more. Virtually all of the extension headers follow a common format so that, even if a particular extension header type is not supported by an IPv6 router, that router can parse past that unknown extension header and continue working on the packet. Figure 37 shows the shared format of IPv6 extensions.

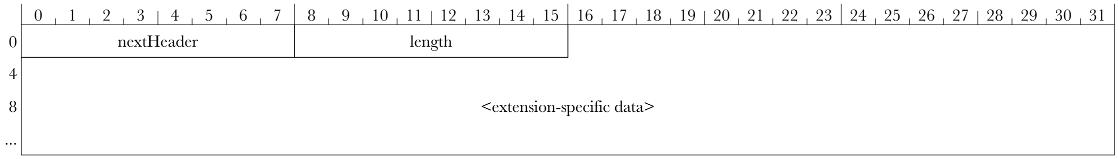


Figure 37 IPv6 Extension Header Structure Diagram

Table 16 IPv6 Extension Header Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`nextHeader` (Option Type)	8 bits	0.0	The protocol (or "type") of the next header. The next header may either be another IPv6 extension header or it may be an upper-layer header such as TCP.
`length` (Opt Data Len)	8 bits	0.8	The length of the current extension header. The length is expressed in units of 64-bit (8-byte) words, not counting the first 64-bit word. This means that an 8-byte extension header has a `length` value of `0` while a 24-byte extension header has a `length` value of `2`.
<extension-specific data> (Option Data)	multiple of 64 bits minus 16 bits	0.16	Extension-specific data.

There is an exception to the common extension header structure described above: the fragment header. The fragment header predates IETF RFC 6564 where the common format was codified.

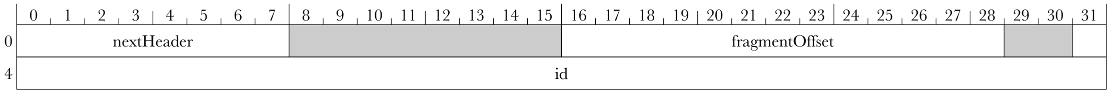


Figure 38 IPv6 Fragment Extension Header Structure Diagram

Table 17 IPv6 Fragment Extension Header Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`nextHeader`	8 bits	0.0	The protocol (or "type") of the next header. The next header may either be another IPv6 extension header or it may be an upper-layer header such as TCP.
`fragmentOffset`	13 bits	0.16	Fragment offset value in 8-byte units. This field is analogous to IPv4's `fragmentOffset`.
`moreFragments`	1 bit	0.31	Indicates that the current packet is not the last fragment. This field is analogous to IPv4's `moreFragments`.
id	32 bits	4.0	The fragmented packet's ID. All fragments of an original, unfragmented packet must have the same `id` value. Though wider, this field is analogous to `Ipv4.id`.

The only IPv6 extension header that an IPv6 router is expected to deal with is the Hop-by-Hop extension header, but support for the Hop-by-Hop extension header is optional and is not supported by most high-performance routers. This means that an IPv6 router must only walk the string of `nextHeader` and `length` fields until it reaches a non-IPv6 header such as TCP.

Time-to-Live

In IPv6, the time-to-live value is called a hop-limit value. This name is certainly much closer to the actual implementations in modern routers than IPv4's time-based value. Note also that it is a hop limit and not a hop count value, reinforcing the notion that the value starts out at some non-zero positive integer and is decremented by each forwarding hop until reaching zero. In this book, IPv6's hop-limit field is referred to as `ttl` simply for consistency and familiarity with the long-standing use in IPv4 and MPLS.

Next Header

IPv6's next-header value is analogous to IPv4's `nextHeader` value. Its purpose is to identify the type of the header that immediately follows the 40-byte IPv6 header.

Internet Control Message Protocol v6

The Internet Control Message Protocol version 6 (ICMPv6, IETF RFC 4443) is used to send information messages or to report errors from the point where the error occurred back to the source of the packet triggering the error. Informational messages include multicast listener queries and echo requests and replies. Examples of ICMPv6 error messages include time-to-live expired and packet too big.

ICMPv6 messages are encapsulated in IPv6 packets and use the `nextHeader` field value `0x3a`. Figure 39 depicts the structure of an ICMPv6 message header.

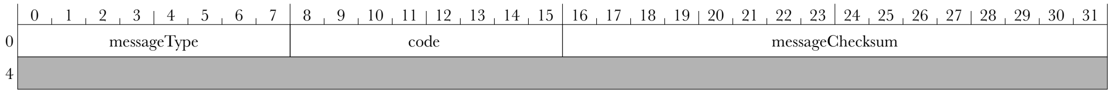


Figure 39 *ICMPv6 Message Header Structure Diagram*

Table 18 *ICMPv6 Message Header Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`messageType` (Type)	8 bits	0.0	The message's type. The `messageType` and `code` values are combined to determine the specific type of the message. `messageType` values `0-127` are error messages while values `128-255` are informational messages.
`code` (Code)	8 bits	0.8	The code for a particular message type. This value demultiplexes the `messageType` value to arrive at a specific message type.
`messageChecksum` (Checksum)	16 bits	0.16	A checksum computed over the entire message. This checksum is the same type of ones-complement checksum as `headerChecksum`. For `messageChecksum`, however, it is computed over all of the fields described in this table as well as certain fields from the encapsulating IPv6 header (known as a pseudo-header) and all of the 16-bit words that make up the data portion of the ICMP message that follows the ICMP message header.
reserved	32 bits	4.0	Unused. These bits must be set to zero by the originator and ignored by the receiver.

There is a data section that follows every ICMPv6 message header. The data section is the IPv6 header from the packet that spawned the ICMPv6 message followed by as much of the original IPv6 packet's payload as possible without exceeding MTU limits.

Virtual Router Redundancy Protocol (VRRP)

See page 87.

Multi-Protocol Label Switching (MPLS)

Why MPLS?

That isn't a facetious or rhetorical question. MPLS was developed long after IPv4 had proven itself to be very effective for building large, globe-spanning networks and IPv6 was under development as a refinement of the same fundamental concepts. So, what problem did MPLS promise to solve?

MPLS was developed in the mid 1990s during the ascent of the Internet. At that time, there was a general consensus that 32-bit IPv4 longest-prefix match lookups were too complex to be practical in silicon and must be performed in software. This condemned IPv4 routers to being slow, expensive systems while Ethernet bridges—which employed relatively simple exact-match lookups for their forwarding operations—were seeing significant performance benefits from implementing the forwarding algorithms in silicon.

A few simple observations about network characteristics led to MPLS. First, if a packet is being forwarded by a series of IPv4 routers in the core of a network, each router is performing an independent lookup on exactly the same `destinationAddress` value and arriving at a functionally equivalent result. Second, packet and bit rates tend to be higher in the core of a network where there is a high degree of convergence of packet flows. Finally, to restate the preceding paragraph, exact-match lookups (or indexed reads) are significantly simpler than longest-prefix lookups.

All this leads to a simple conclusion: perform the complex lookup just once at the edge of the network, represent that lookup result with a proxy value (a "label"), and have the routers in the core of the network forward the packet using a simple lookup on the label value. Specifically, a 20-bit, locally-scoped label value is used as a component of an exact-match search argument or as a direct index into a table to fetch the forwarding instructions to the packet's next hop.

Figure 40 and Table 19 define the format and field definitions of an MPLS header.

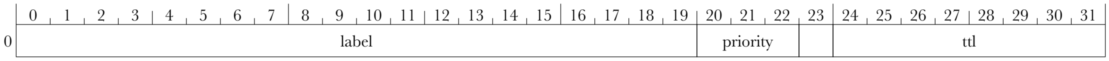


Figure 40 MPLS Header Structure Diagram

Table 19 *MPLS Header Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`label` (Label)	20 bits	0.0	MPLS forwarding information. It is generally locally scoped and it is without structure. There are a small number of reserved values.
`priority` (was EXP, now CoS)	3 bits	0.20	QoS priority. This priority code point is mapped to internal traffic class values in application-specific ways.
`bottomOfStack` (S)	1 bit	0.23	Identifies the last header in a "stack." When set to `1`, this bit marks the last header in a stack of MPLS headers. Typically, the next byte after an MPLS header whose `bottomOfStack` bit is set is the first byte of the MPLS payload. There are, of course, exceptions.
`ttl` (TTL)	8 bits	0.24	The packet's hop limit.

As you can see, the structure of an MPLS header is quite simple. Unfortunately, it may be a bit too simple for it is missing something that Ethernet, IPv4, and IPv6 have: some kind of "what to expect next" field. MPLS has no functional equivalent to Ethernet's `ethertype` field or IP's `nextHeader`. The type of the MPLS payload must be inferred from information that may only be meaningful at one hop in the network. This and other quirks of MPLS are explored as we tour the contents of its header structure.

Labels

The twenty-bit label width was chosen as a compromise. It is narrow enough to use directly as a memory address value or as a space-efficient component of an exact-match search argument, but wide enough to provide reasonable scale for a single router node. Unfortunately for MPLS, a significant development in networking condemned this major selling point of MPLS to irrelevance; Juniper Networks proved with the M40 router that it was, indeed, possible to perform high-speed longest-prefix matches in hardware.

So, why has MPLS persisted and flourished in the face of its major benefit being nullified? The answer is that it appeals to those who want absolute control over the forwarding decisions made for every packet rather than relinquishing that control to routing protocols and hash functions. This is known as traffic engineering and it is very effective at allowing software-defined networking (i.e., SDN) controllers to manage the flow of data through an network. One particular MPLS model that lends itself particularly well to management by an SDN controller is what's known as "segment routing." This is, essentially, a modernized version of source routing that was attempted and abandoned in the 1980s. The concept is simple: a packet carries with it all of the hop-by-hop forwarding decisions that each router in turn must carry out. In the case of MPLS segment routing, a stack of MPLS headers are used and each MPLS label switch router in the core of the network examines the outermost MPLS header, pops that header and forwards the packet based on that header. The next router uses the next MPLS header. Label switch routers and other details of MPLS forwarding behavior are described next.

MPLS Core Behavior

The narrowness of `Mpls.label` makes it pretty much useless as a global address value or even an end-to-end value in a large, closed network. Hence, in most MPLS networks, `label`'s value is only relevant to the single MPLS router in the network that examines it. The same numerical value may have a very different meaning to a different router.

The typical forwarding model of MPLS is label swapping—i.e., the result of a label lookup is a new `label` value as well as a transmit interface identifier. As an MPLS packet traverses the network, its `label` value is swapped for a new value at every hop; that swapped-in value only being relevant to the very next hop in the MPLS network. The path that an MPLS packet follows is known as a label switch path (LSP) and the routers that do this forwarding are known as label switch routers (LSRs) or transit routers.

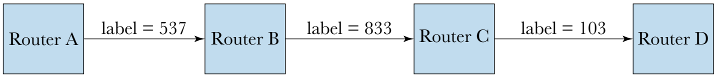


Figure 41 *MPLS Labels and Paths*

Unlike Ethernet and IPv4/IPv6, which have both source and destination address values in their headers, MPLS has just a single label value. The label value is not associated with either a source or a destination but, rather, with the network segment between a source and a destination. Referring to Figure 41, when Router A sends a packet to Router B, it does so by giving an MPLS header a `label` value of `537`. Router B, in turn, examines that `537` value in its own forwarding database and determines that a `label` value of `833` is required to identify the path between it and Router C. The process continues thusly to Router D and beyond.

As you learned in Chapter 4, what's happening here is that each segment in the MPLS network is a tunnel that originates at one router is then terminated at the very next router (i.e., the next hop). This all occurs at the same level of hierarchy. In other words, the depth of the tunnel encapsulations does not change with each hop. This behavior of exiting and entering tunnels at a single forwarding point is known as "label swapping" and is a fundamental behavior of MPLS. For an example of how label swapping and label-based forwarding enable traffic engineering, consider Figure 42.

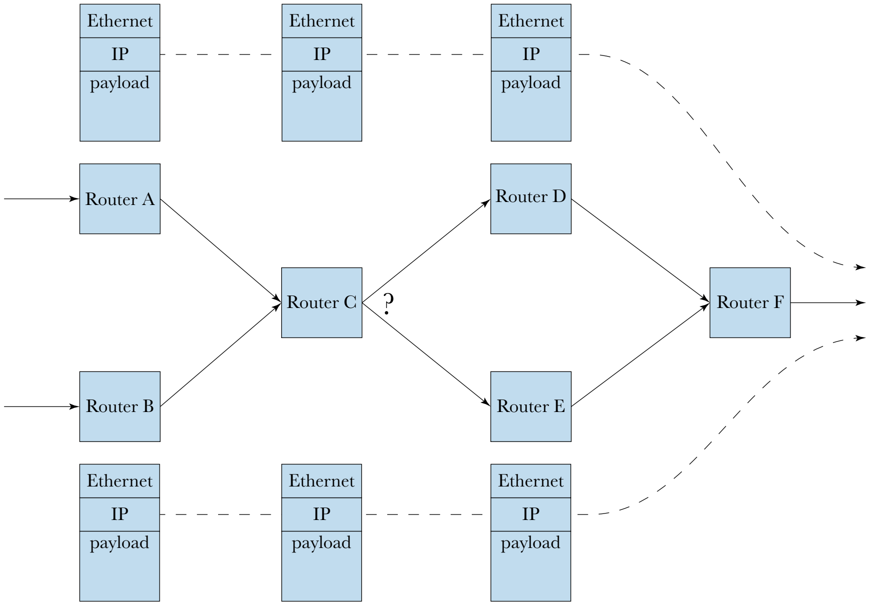


Figure 42 *IP Routing on Converging/Diverging Networks*

As shown in Figure 42, two IP packets are addressed to the same destination across a network that, at one point, converges on a common router (Router C) before diverging again. When the packets arrive at Router C via Router A and Router B, they both have identical IP destination addresses. If the desired behavior is for the upper packet to be forwarded to Router D and the lower to Router E, how is Router C supposed to make that determination?

Figure 43 shows the same network architecture but with the IP packets encapsulated inside MPLS packets.

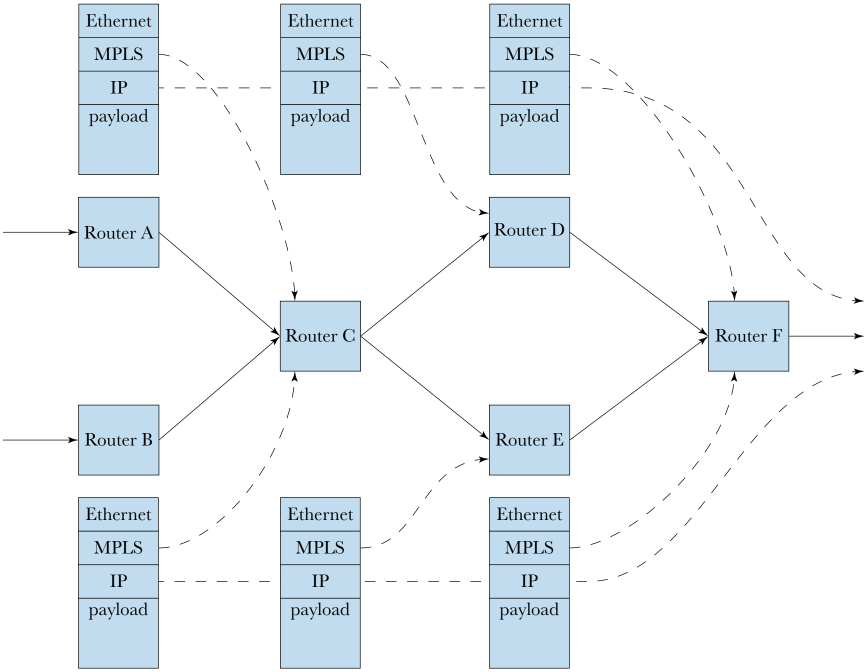


Figure 43 *MPLS Routing on Converging/Diverging Networks*

With the IP packets encapsulated inside MPLS packets, they arrive at Router C with different `label` values even though, ultimately, they are addressed to the same final destination. Their `label` values are different because the packets are following different label switch paths (LSPs). Because the label values differ for the packets that arrive at Router C via Router A versus those that arrive via Router B, Router C is able to easily distinguish between the two paths and forward the upper packet to Router D and the lower packet to Router E.

Tunnel encapsulations of arbitrary depth are permitted in MPLS. Just as in the other forwarding protocols, MPLS packets are allowed to enter other MPLS tunnels. When an MPLS packet enters a new MPLS tunnel, a new MPLS header is prepended to the packet and it becomes the new top-of-stack header. This is known as pushing[5] a label onto the stack. However, when entering a new MPLS tunnel from an existing MPLS tunnel without terminating the existing tunnel (i.e., the existing tunnel enters the new tunnel as the new tunnel's payload), it is unusual to simply push a new MPLS header onto the stack that represents the tunnel being entered. More commonly, a swap on the existing MPLS tunnel is performed prior to pushing a new MPLS header onto the stack. The reason for this is that tunnel

[5] The terms "push" and "pop" come from the stack concept in computer science. Indeed, a consecutive set of MPLS headers terminated by an MPLS header whose `bottomOfStack` bit is set to 1 is known as a "label stack." But really, trust me, they're headers.

that is being entered can be thought of as a single hop or segment as far as its payload is concerned. Therefore, the MPLS router at the exit of the tunnel that must pop the MPLS header that corresponds to the terminating outermost tunnel must also examine the next MPLS header in the stack and make a forwarding decision based on its `label` value. If that `label` value isn't swapped at the tunnel entrance, then it becomes necessary for two MPLS routers to correctly interpret the same `label` value: the router that preceded the new tunnel's entrance and the router at the new tunnel's exit. By swapping before pushing, the router at the tunnel's exit is provided with a `label` value that is relevant to it.

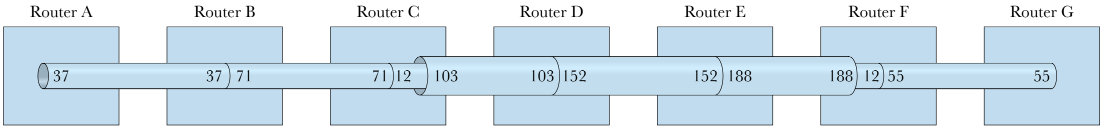


Figure 44 *MPLS Tunnels Visualized*

When an MPLS packet exits a tunnel that is terminated, the outermost MPLS header is stripped from the packet. This is known as popping a label from the stack. MPLS headers are "pushed" onto a packet to enter a tunnel. Exiting and entering consecutive MPLS tunnels at a single hop uses a technique known as "label swapping."

Though the term "label" is often used when describing MPLS header manipulations, it really is whole MPLS headers that are being manipulated. By building a stack of MPLS headers, a hierarchical network structure can be built.

MPLS Edge Behavior

The MPLS routers that sit at the edges of an MPLS network—i.e., the routers that form the boundary between an IP-based edge network and an MPLS-based core network—are known as label edge routers (LERs). Label edge routers perform fundamental tunneling operations just like any other tunneling scheme:

1. An Ethernet + IP packet is received, the Ethernet tunnel is terminated and the Ethernet header is stripped.
2. Destination processing reveals that the IP packet is addressed to a destination on the far side of an MPLS tunnel (i.e., a label switch path, or LSP).
3. The IP packet is encapsulated in one or more MPLS headers that tell the next-hop MPLS transit router how to forward the packet.
4. The MPLS + IP packet is encapsulated in an Ethernet packet and transmitted via an interface that delivers the packet to its next hop.

At the egress edge of the MPLS network, the label-edge router terminates the last MPLS tunnel and strips off the last MPLS header (i.e., the MPLS header whose `bottomOfStack` bit is set). The egress label-edge router then processes the encapsulated packet (IPv4 or IPv6 in this example) and forwards it as directed by its header. However, this egress edge behavior demands that a router perform two lookups on a single packet: one to terminate the MPLS tunnel and one to forward the encapsulated IPv4 or IPv6 packet (or some other kind of packet). Penultimate hop popping was developed to relieve the egress edge router of this burden (described further below).

Not all MPLS headers imply forwarding behaviors. Non-forwarding MPLS headers are commonly used to indicate the type of an MPLS tunnel. For example, a single label switch path may need to convey Ethernet, IPv4 and IPv6 packets. To overcome MPLS's lack of a next-header indicator value, a label value may be used to perform the same function. In this scenario, a next-header-identifying label is first pushed onto the MPLS stack and is marked as being the bottom-of-stack. Next, an MPLS header whose `label` value corresponds to the first hop of the label switch path is pushed onto the MPLS stack. As the packet is forwarded, the MPLS header at the top of the stack is interpreted and swapped as necessary while the bottom-of-stack header is left alone. At the egress edge of the MPLS network, the top-of-stack MPLS header is examined and its `label` value directs the router to pop that header from the stack and to treat the next MPLS header's `label` value as a next-header indicator. Finally, the bottom-of-stack header is popped and the newly-exposed payload header (whose type has been reliably determined by the bottom-of-stack MPLS header) is processed for forwarding.

Penultimate Hop Popping (PHP)

Ordinarily, an MPLS label switch path operates by swapping MPLS labels at each hop, with the last (i.e., ultimate) MPLS hop—the egress label edge router—popping the MPLS header from the packet, revealing a bare IP packet or some other type of packet. That MPLS egress edge router must then route the newly-exposed packet.

With penultimate hop popping, the MPLS transit router that immediately precedes the MPLS egress label edge router is configured to both forward the packet based on the MPLS header's label value (as is normal) and also popping that header from the packet prior to transmission instead of swapping in a new label value. If an MPLS control word is present, it is also stripped from the packet. These actions deliver a bare IP packet to the MPLS egress label edge router, which, in turn, forwards the packet according to its IP header. In this way, both the penultimate and ultimate MPLS hop each performs just one lookup operation.

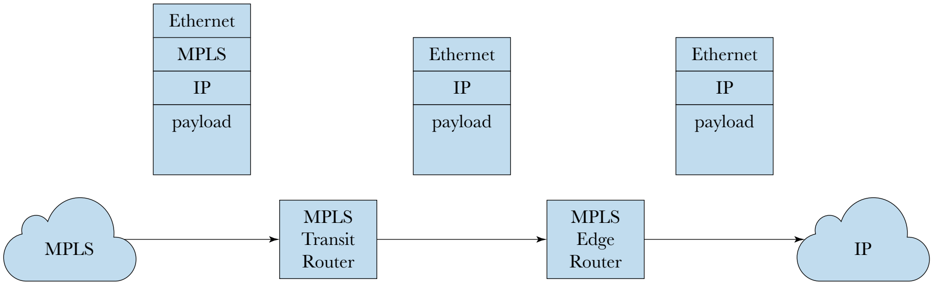


Figure 45 *MPLS Penultimate Hop Popping Illustrated*

In Figure 45, an IP packet encapsulated inside of MPLS and Ethernet arrives at an MPLS transit router (i.e., the penultimate hop). The `label` value in the MPLS header directs the MPLS transit router to forward the packet to the MPLS edge router and to also strip the MPLS header from the packet. This is the penultimate hop popping behavior. The MPLS edge router (i.e., the ultimate hop) forwards the IP packet in the normal manner into the IP portion of the overall network.

As is usual with MPLS, penultimate hop popping behavior is not initiated due to some op-code or bit that is set in an MPLS header. Instead, a seemingly normal `label` value is submitted to a lookup in the router's forwarding database and an implicit null label value is returned as a result. An implicit null label informs the router to pop the current top of the MPLS header stack and to *not* push on a replacement header. The use of an actual implicit null label value in the forwarding database is not strictly required. Any appropriate and system-specific value may be returned as a lookup result as long as the processing hardware knows to implement the correct behaviors in response.

Bottom-of-stack processing is a deep and complex issue with MPLS. Hence, it merits its own section, further below.

Reserved Label Values

MPLS label values in the range of `0x0_0000` through `0x0_000f` are reserved for special purposes.

Table 20 *MPLS Reserved Labels*

Value	Description	Reference
0	IPv4 explicit null label	RFC 3032
1	router alert label	RFC 3032
2	IPv6 explicit null label	RFC 3032
3	implicit null label	RFC 3032

Value	Description	Reference
4–6	reserved	
7	entropy label indicator (ELI)	RFC 6790
8–12	reserved	
13	GAL label	RFC 5586
14	OAM alert label	RFC 3429
15	extension label	RFC 7274

A number of the reserved `label` values are used to identify the MPLS payload or, in the case of the entropy label indicator, stand in for the MPLS payload. These various solutions to determining what follows the bottom-of-stack are explored in detail below.

Bottom of Stack (and Beyond)

In virtually all material referring to MPLS, the `bottomOfStack` bit is known as the `S` bit for the singularly irritating reason that standards body participants have a penchant for only assigning names to fields that happen to fit in the space available in a header structure diagram. Hence, one-bit fields often have one-character names. I think we can all agree that "`S`" is not very descriptive.

The purpose of the `bottomOfStack` bit is to mark the end of the MPLS header stack and the transition to the MPLS payload. As you will see, the bottom-of-stack is not always the end of MPLS-related header data. And, figuring out what follows an MPLS header stack is one of the more complex aspects of the protocol.

A variety of techniques for positively identifying the payload of a stack of MPLS headers have been developed. What follows is a survey of those techniques.

Label Switch Path Inferred Payload

Without the functional equivalent of an `ethertype` or `nextHeader` field to unambiguously identify the next header, how does an MPLS router figure out what kind of payload the MPLS header stack is conveying? The initial concept for MPLS was that each LSP (label switch path) would carry just one kind of payload. This works well enough for MPLS egress label edge routers that are intended to terminate the final MPLS tunnel and then forward the packet that was the payload of the MPLS tunnel. Such routers simply encode a protocol-type value into the MPLS label lookup result. Thus, the label switch path is effectively used to infer the payload type: one payload type per LSP.

Unfortunately, LSP-inferred payload determination leaves intermediate MPLS label switch path routers at a loss when the ultimate MPLS payload is buried within several further layers of MPLS tunnels. An MPLS router in the middle of an MPLS network will only have enough information in its forwarding database to terminate any tunnels addressed to it, and to forward MPLS packets based on the MPLS header that remains after popping those MPLS headers associated with terminated tunnels. Problems arise, however, when performing equal-cost, multi-path (ECMP) load balancing (discussed in Chapter 8). The identities of a series of terminated tunnels and a single forwarding label value are generally insufficient flow-identifying information for making egress path selections that achieve good load balancing across the available links. The inner MPLS headers of an MPLS header stack (i.e., those MPLS headers that follow the forwarding header) are easily parsed and their label values should be included in a load balancing hash computation. However, it is often beneficial to include flow-identifying fields from headers of the MPLS payload in the hash computation. This, of course, requires that these headers be parsed in order to extract the relevant fields. Unfortunately, an MPLS label switch router (LSR) at the core of an MPLS network doesn't have forwarding database entries for label values inside of the forwarding MPLS header, making it incapable of deterministically identifying the type of the MPLS payload. Entropy labels can go a long way toward supporting ECMP without having to parse beyond the bottom of the stack of MPLS headers, but entropy labels are not universally employed.

Payload Best Guess

The LSP-inferred approach to next-header determination leaves a transit router with having to make an educated guess regarding the type of the payload. Unfortunately, the multi-protocol aspect of MPLS is quite true; just about every protocol type can be conveyed by MPLS. IPv4 and IPv6 can be speculatively identified by examining the first four bits that follow the MPLS header whose `bottomOfStack` bit is set to `1`. If that 4-bit value is equal to `4`, then there's a good chance that an IPv4 packet is the payload. A value of `6` implies IPv6. Of course, it's also possible for `Mac.destinationAddress` values to have `4` or `6` in that location. The reliability of a guess of IPv4 can be vastly improved by confirming that the supposed IPv4 header's `headerChecksum` and `totalLength` values are correct. For IPv6, `payloadLength` can be checked. This isn't foolproof, but it helps a lot. The silver lining for this situation is that these payload headers are generally being parsed for load balancing information. If an MPLS router guesses wrong, the consequences are that non-flow identifying fields may be hashed and packets could arrive out of order. However, this is far from ideal.

Explicit Null Labels

Null labels are not used in MPLS forwarding decisions. They are used to direct other router behavior. In the case of the two explicit null label values, they are used to indicate that the payload of an MPLS header stack is either an IPv4 packet (`label == 0`) or an IPv6 packet (`label == 2`). Explicit null label values are only allowed at the bottom of the stack (i.e., bottomOfStack must be set to 1). Explicit null label values were originally intended for use in penultimate hop popping where the last forwarding label is popped off of the stack and is replaced by an explicit null label. This was done to preserve the MPLS quality of service paradigm to the edge of the MPLS network while providing a means for indicating the MPLS payload type. However, in modern MPLS networks, an explicit null may be pushed onto the MPLS label stack as the bottom-of-stack by the ingress MPLS edge router along with any number of additional MPLS headers.

Implicit Null Label

An implicit null label value is never actually assigned to label in an MPLS header. It is distributed to MPLS routers for use in their forwarding databases to affect router behavior. If a lookup on a label value returns an implicit null value as its result, the MPLS router is expected to "pop" the stack (i.e., strip off the outermost MPLS header) and not push on a replacement (i.e., the label value returned from a successful lookup operation). In summary, getting an implicit null label value as a lookup result means "pop" rather than "swap." This behavior is related to penultimate hop popping (PHP) as previously described.

Entropy Label

Transit routers may have difficulty reliably parsing any headers beyond the MPLS headers. This means that valuable flow-identifying information from payload headers (e.g., IPv4 or IPv6) goes unused, diminishing the quality of the computed load-balancing hash values. To address this problem, the entropy label was invented.

Entropy labels are simply MPLS headers whose label value contains flow-identifying entropy (i.e., information). Typically, the label value of an entropy label is computed by an ingress label-edge router using any label values that follow the entropy label as well as fields from headers contained in the MPLS payload (e.g., sourceAddress, destinationAddress, nextHeader, sourcePort, destinationPort). However, an entropy label value may also include information that precedes the entropy label in its hash computation.

When a transit router encounters an MPLS header stack that contains an entropy label, it may incorporate that label value into its load-balancing hash computation. It has the option of using just the entropy label, all label values in the stack including the entropy label, or all of the label values (including the entropy label) as well as any header fields that it is able to reliably parse from the MPLS payload.

By itself, an entropy label (EL) looks just like any other MPLS label, its label value is any non-reserved value, its bottomOfStack bit may be set to 1 or 0 and its priority value may be set to any arbitrary value. There are, though, two things that set entropy labels apart. First, its ttl value is always 0. This prevents an MPLS router from using an entropy label as a forwarding label. And, second, an entropy label is *always* preceded by an entropy label indicator (ELI) MPLS header. The entropy label indicator has a `label` value of `7` and its `bottomOfStack` value is always set to `0` (i.e., an entropy label indicator may never be the bottom of the stack since it must be followed by an entropy label).

Flow Label

Pseudowires defined by RFC 4447 have their own method for introducing entropy for load balancing purposes. IETF RFC 6391 specifies a method where a new MPLS header is added to the bottom of the MPLS stack, becoming the new bottom-of-stack (i.e., its `bottomOfStack` bit is set to `1`). This new label value—referred to as a “flow label”—is analogous to an entropy label and its value is computed in the same manner. The big difference between a pseudowire flow label and an entropy label is that a flow label is not preceded by any kind of special entropy label indicator equivalent. Since there is no reserved label value present in the MPLS stack indicating that a flow label is present, the transit routers’ forwarding database must be configured such that the pseudowire-identifying MPLS label value is associated with the use of a flow label. Another minor difference between entropy labels and flow labels is that RFC 6391 also recommends setting a flow label’s `ttl` value to `1`.

Control Word

As was previously described, it is common practice for MPLS routers to examine the first four bits following the bottom of the MPLS stack to determine the MPLS payload type. A value of `4` is assumed to be IPv4 and `6` is assumed to be IPv6. Taking advantage of this somewhat dubious practice, the MPLS control word was defined by RFC 4385 (and updated by others). An MPLS control word immediately follows the bottom of an MPLS header stack and its most significant four bits are always set to `0`.

The remainder of the control word in Figure 46 is defined in an application-specific manner. The pseudowire control word serves as a useful example.

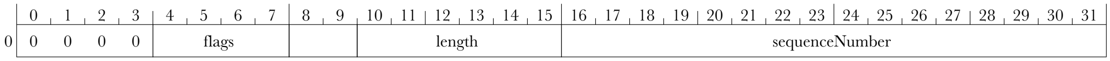


Figure 46 Pseudowire Control Word Structure Diagram

Table 21 Pseudowire Control Word Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`flags` (Flags)	4 bits	0.4	Used for per-payload signaling.
`fragmentInfo` (FRG)	2 bits	0.8	Used when fragmenting a pseudowire payload.
`length` (Length)	6 bits	0.10	Indicates the amount of Ethernet padding added to a packet. If the payload of the encapsulated Ethernet packet is less than 64 bytes, its length is defined by `length`. Otherwise, `length` is set to `0` and the length of the payload is inferred from the overall packet length.
`sequenceNumber` (Sequence Number)	16 bits	0.16	Used to re-order packets when equal-cost, multi-path (ECMP) forwarding is being applied to the pseudowire.

One of the useful purposes of the pseudowire control word is that it lets the MPLS router know that the MPLS payload is Ethernet. But, there may be many other types of MPLS payloads besides, Ethernet, IPv4, and IPv6. This is where the generic associated channel header comes in.

Generic Associated Channel Header (G-ACh)

The generic associated channel header (G-ACh or ACH) is never used to convey user packets[6]. For user packets, control word encapsulation is used. The generic associated channel header is meant to encapsulate a variety of messages that belong to a protocol that is carried by a pseudowire, label switch path or MPLS section associated control channel. So, yes, the control word is for user packets and the generic associated channel header is for control packets. (Don't get me started!)

Like a control word, a generic associated channel header immediately follows the bottom of the MPLS stack. And, also like a control word, it uses an unique constant in its most significant four bits to almost-reliably distinguish it from all other possible MPLS payloads. That value is `1`.

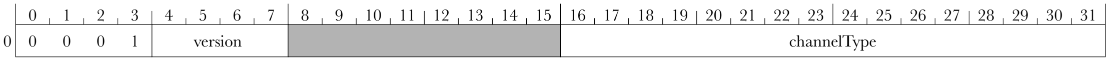


Figure 47 Generic Associated Channel Header Structure Diagram

[6] Despite this, there are channel-type code points for IPv4 and IPv6.

Table 22 Generic Associated Channel Header Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`version` (Version)	4 bits	0.4	Indicates the version of G-ACh. Currently, `version` is set to `0`.
`channelType` (Channel Type)	16 bits	0.16	The channel (or payload) type. This field is roughly analogous to `ethertype`, but its enumerations are different. Some of the interesting values are: `0x0007` = BFD control (RFC 5885) `0x000a` = MPLS direct loss measurement (RFC 6374) `0x000b` = MPLS inferred loss measurement (RFC 6374) `0x000c` = MPLS delay measurement (RFC 6374) `0x000d` = MPLS direct loss & delay measurement (RFC 6374) `0x000e` = MPLS inferred loss & delay measurement (RFC 6374) `0x0021` = IPv4 (RFC 4385) `0x0027` = pseudowire OAM message (RFC 7769) `0x0057` = IPv6 (RFC 4385) `0x0058` = fault OAM (RFC 6427) `0x8902` = G.8113.1 OAM (RFC 6671)

Generic Associated Channel Label (GAL)

To make the presence of a generic associate channel header explicit and unambiguous, a generic associate channel label is used. This MPLS header has a `label` value of `13` and must be the bottom of the MPLS stack (i.e., its `bottomOfStack` bit is set to `1`). A generic associated channel header always immediately follows a generic associated channel label. However, the use of a generic associated channel label to introduce a generic associated channel header is optional.

In Summary

Figure 48 summarizes the various methods for dealing with content beyond the bottom of the stack.

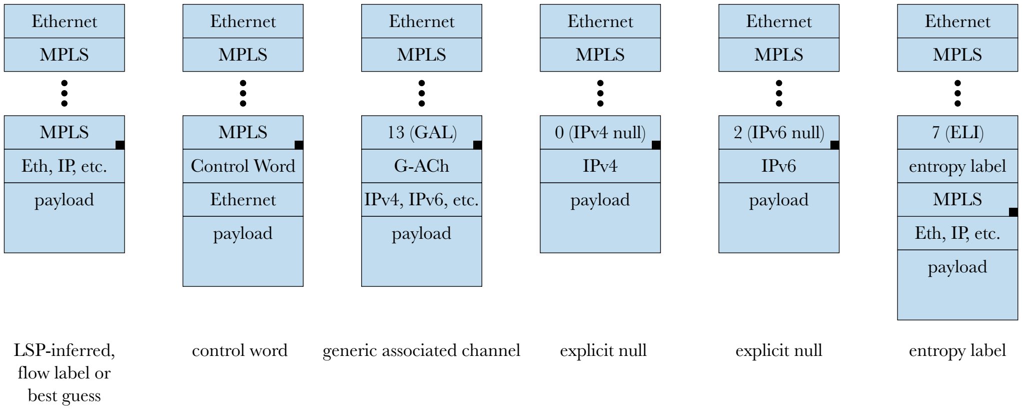


Figure 48 MPLS Beyond the Bottom of the Stack

Priority

The `priority` field (formerly known as the “EXP bits,” for “experimental bits”) is just three bits in length. Compared to Ethernet’s four bits and IP’s six bits, those in the MPLS camp are, understandably, left feeling somewhat inadequate. Despite its smaller size, it serves exactly the same function as the priority bits in the other forwarding protocols; it encodes information on how to handle a packet relative to other packets with different priority values during times of congestion.

Time to Live

Like the two variants of IP that have been discussed in this book, MPLS also has a time-to-live value carried in its header. In fact, a very deep stack of MPLS headers may have many time-to-live values. Despite this apparent complexity, the basic rules for dealing with `ttl` are quite simple:

> The outgoing `ttl` value must be one less than the incoming `ttl` value (but not less than `0`), regardless of the number of MPLS headers that may have been popped, swapped and pushed in between packet reception and transmission. If the outgoing `ttl` value is `0`, the packet must not be transmitted.

What this means in practice is that the `ttl` value of the outermost (i.e., topmost) MPLS header must be checked, decremented and set aside in temporary storage. As labels are popped, swapped and pushed as part of the packet’s normal processing by an MPLS router, that stored `ttl` value is then written to the packet’s new outermost MPLS header. The `ttl` values for any inner headers may be safely ignored (it is presumed that any forwarding labels were pushed onto the packet with non-zero `ttl` values and that the payload itself has non-zero time-to-live values).

Time to Live and Conveying IP in MPLS Tunnels

Transparent Tunnels

When an IP packet is first encapsulated within an MPLS packet, the outermost MPLS header's `ttl` value must be set to the `ttl` value of the encapsulated IP packet. It is presumed the IP time-to-live value has been decremented prior to this operation.

Upon exiting the MPLS tunnel, the encapsulated IP packet's time-to-live value may be replaced with the outgoing MPLS header's `ttl` value (i.e., post-decrement). What this means is that, in effect, the IP packet's time-to-live value is decremented for all of the hops made by the MPLS encapsulation. This makes the payload of the MPLS tunnel transparent—i.e., the payload is available and subject to actions.

Opaque Tunnels

Certain network architectures may prefer to treat the IP payload of the MPLS tunnel as opaque content; the IP payload consequently being immune to inspection or modification by the tunnel itself. In this case, the IP time-to-live value is preserved while in the tunnel and is only decremented when the packet is forwarded by the IP header itself (i.e., at the entrance and exit of the tunnel). It may also be a configurable policy that entering an MPLS tunnel be viewed as entering an IP router whose receive and transmit interfaces physically separated by a vast distance. In this case, it is appropriate to decrement the IP packet's time-to-live value just once: typically just when entering the MPLS tunnel.

Distinguishing Transparent from Opaque

Only the MPLS edge router at the entrance to an MPLS tunnel is configured to know whether to treat a particular MPLS tunnel as transparent or opaque. To signal transparent vs. opaque tunnel behavior to the MPLS egress edge router, the tunnel type is encoded in the `Mpls.ttl` value.

For a transparent MPLS tunnel, the `ttl` value of the first MPLS header pushed onto the IP packet is set equal to the post-decrement value of `Ipv4.ttl` or `Ipv6.ttl`. This initial time-to-live value must be, by convention, less than or equal to `64`. For an opaque MPLS tunnel, the `ttl` value of the first MPLS header pushed onto the IP packet is set to an arbitrarily large value (`255`, typically). In both cases, the `Mpls.ttl` value is decremented by one at each hop while the IP `ttl` value is left alone until reaching the MPLS egress edge router.

At the MPLS egress edge router, the `Mpls.ttl` value is compared to the IP `ttl` value. If the `Mpls.ttl` value is less than the IP `ttl` value, then the tunnel is transparent and the IP time-to-live value must be set to the decremented `Mpls.ttl` value. Otherwise, the tunnel is opaque and the IP time-to-live value is just decremented by one (or left alone for certain network policies).

Just to make it very clear how this works, in a transparent tunnel the initial MPLS time-to-live value is set to the post-decremented IP time-to-live value. The MPLS time-to-live value is then decremented by any subsequent MPLS hops (there will always be at least one). Hence, the MPLS time-to-live value is guaranteed to be less than the IP time-to-live value. For an opaque tunnel, its identification is always also unambiguous as long as the initial difference between the IP time-to-live value and the MPLS time-to-live value is greater than the number of hops across the MPLS network. If the number of hops exceeds this limit, the MPLS time-to-live value will be less than the IP time-to-live value and the supposedly opaque tunnel will be interpreted to be a transparent tunnel by the egress label edge router.

MPLS Time-to-Live Primitives

In summary, there are a number of operational primitives that may be applied when MPLS labels are pushed, swapped or popped, regardless of whether the encapsulated packet is an MPLS packet or an IP packet.

The *push* time-to-live primitives are:

- Set `ttl` of newly pushed MPLS header to time-to-live value of the encapsulated packet minus one (do not change time-to-live value of the encapsulated packet).
- Set `ttl` of newly pushed MPLS header to a constant (e.g., 255) and decrement the time-to-live value of the encapsulated packet.

The *swap* time-to-live primitive is:

- Decrement `ttl`.

The *pop* time-to-live primitives are:

- Decrement the `ttl` value from the popped MPLS header and then copy it to the encapsulated packet's time-to-live value.
- Discard the `ttl` value from the popped MPLS header and decrement the time-to-live value of the encapsulated packet.
- Discard the `ttl` value from the popped MPLS header and leave the time-to-live value of the encapsulated packet unchanged.

The *pop* time-to-live conditions that can be used to execute any of the preceding pop primitives are:

- Penultimate hop popping is in use.
- Penultimate hop popping is not in use.
- `ttl` of the popped MPLS header is less than the time-to-live value of the encapsulated packet.
- the time-to-live value of the encapsulated packet is less than `ttl` of the popped MPLS header.

These primitives and conditions may be employed in practically all possible combinations.

Cross Connect

In early telephone networks, a call was placed by establishing an electrical connection between a twisted-pair wire coming into the telephone company's central office (CO) from one customer to the wire associated with another customer. Whether that connection was established via old-timey switchboards or with automated mechanical relays, the effect was the same: all of the information arriving on a particular physical interface (i.e., twisted-pair wire) was forwarded to a single outbound interface with a minimum of delay and without examining the information flowing through the connection. Typically, a reciprocal connection was also established. Cross-connect networking operates in a similar manner.

If you recall, we've previously established that all packets arriving at a forwarding system are encapsulated by an imaginary header that represents the packet's receive interface. Hence, it is completely reasonable and practical to define a forwarding entity that only knows how to interpret the imaginary physical interface header. All this cross-connect forwarding entity has to do is map receive interfaces to transmit interfaces. Thus, all packets arriving on a particular physical port are forwarded to some other physical port without examining any of the encapsulated headers for forwarding purposes.

Of course, a physical port may be a member of a link aggregation group (LAG), so what a cross-connect forwarding entity really wants to work with is logical ports. And, as we've seen with other complex forwarding systems, the output of a forwarding entity may need to modify a packet's encapsulation when a forwarding entity's output interface is connected to an input interface of another type of forwarding entity. Hence, we could end up with a cross-connect forwarding entity forwarding packets to an MPLS forwarding entity within the same forwarding system. In practice, this means, for example, that all packets received by a particular logical interface may be encapsulated by an MPLS header whose `label` value is the same for all such packets. The MPLS forwarding entity may then forward the packet via an output interface that requires Ethernet encapsulation. The net result of this is that an Ethernet packet received via a logical port is encapsulated in a set of Ethernet and MPLS headers before being transmitted to the next hop.

The inverse of the scenario described in the preceding paragraph is also possible and is easy to imagine. A packet consisting of an Ethernet packet within an MPLS packet within an Ethernet packet can reach the tunnel termination points for the outer Ethernet header and the MPLS header, revealing the inner Ethernet packet. The MPLS forwarding entity in this case forwards the packet via a cross-connect forwarding entity instead of an Ethernet forwarding entity: meaning that the contents of the inner Ethernet packet are ignored and the packet's transmit interface is, effectively, selected by the innermost forwarding header of the MPLS header stack.

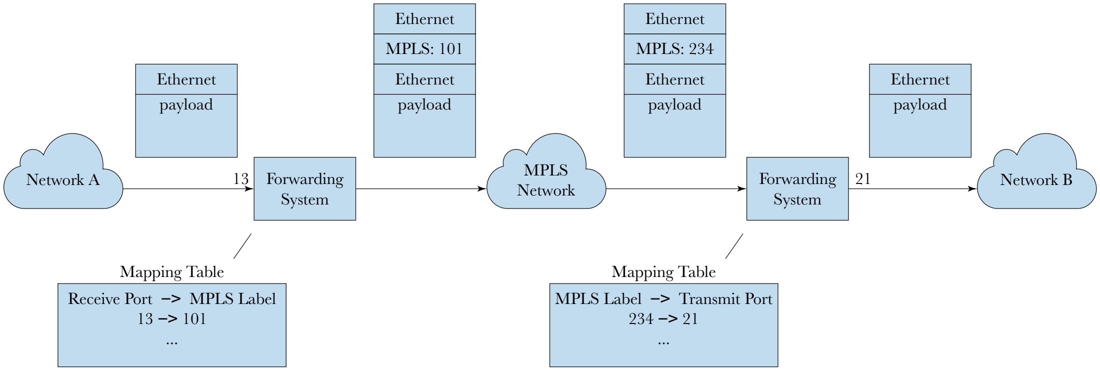


Figure 49 Cross Connect Using MPLS Transport

In the example shown in Figure 49, an Ethernet packet with some opaque payload is received by the forwarding system on the left. Without even examining the packet's Ethernet header, the packet's receive interface ID (`13`, in this case) is used to map the packet to an MPLS encapsulation with a `label` value of `101`. The packet is then forwarded according to that label by the leftmost forwarding system into an MPLS network. In the network, the MPLS header's `label` value is swapped with each hop in the usual manner. Upon exiting the MPLS network, the rightmost forwarding system interprets the `label` value of `234` to mean: "forward this packet via transmit port `21`." Hence, the packet's outer Ethernet and MPLS headers are stripped and the original Ethernet packet is delivered to Network B without the headers or fields of the original packet being examined by any of the forwarding systems between Network A and Network B.

In a similar vein as the use of logical interfaces for forwarding, virtual LANs may also be used. In this case, the combination of a receive interface ID and one or two VLAN IDs are used to make a forwarding decision, ignoring the `destinationAddress` field of the outermost Ethernet header.

All kinds of variations on this general concept are possible. Logically speaking, it's just a matter of arranging the appropriate forwarding entity types in the right order so as to accomplish the necessary encapsulations and decapsulations. The important thing to keep in mind here is that a physical or logical interface is just as useful a source of forwarding information as any header or protocol defined by the IEEE or IETF.

8 Load Balancing

Redundant paths through networks are very common. There are number of reasons why this is the case. First, as link speeds increase, their costs per bit per second can also tend to increase. (At least at the very high end.) Because of this, it is often far more cost effective to use multiple slower links instead of a single very high speed link. Next, the bandwidth demands of some applications may exceed the capacity of even the fastest available physical layer technology, necessitating the adoption of multiple, parallel links in the place of a single one. Finally, links can fail. If the physical infrastructure of a network only allows for a single path from any point to any other point, then, when one of the links in that path fails, that network will be severely degraded until a repair is made. Having redundant paths already extant in the network makes it easy to quickly react to failures. If those redundant paths are already available, it makes sense to make use of them all the time and not just during failures.

Load Balancing Essentials

Identifying Flows and Avoiding Mis-Ordering

Bridging requires in-order packet delivery and routing greatly benefits from it. This naturally leads to the question: what does it mean for packets to be in order? To adequately answer this question, we must take a closer look at the definition of a flow.

A flow is a series of packets sent from an originating endpoint to a terminating endpoint. These packets must be received by the terminating endpoint in exactly the same order in which they were transmitted by the originating endpoint. An endpoint, however, is not necessarily a monolithic thing. Packets transmitted by a physical endpoint (server, computer, etc.), may come from any number of internal processes. For example, a web browser may be downloading HTML content at the same time that an FTP client is downloading a file. Each of these processes expects to receive its packets in order, but is not affected in any way by the relative order of arrival of the packets between the two independent processes. In other words, it does not matter if the web browser receives, say, ten packets before the FTP client receives a burst of, say, seven packets, or if the web browser initially receives a run of just four packets before FTP packets start to arrive. As far as each process is concerned, it's receiving its packets in the correct order.

To take this example a bit further, each process—web browser and FTP client—may be downloading multiple, independent files from different servers. Again, the relative order of packet arrival between these subprocesses does not matter.

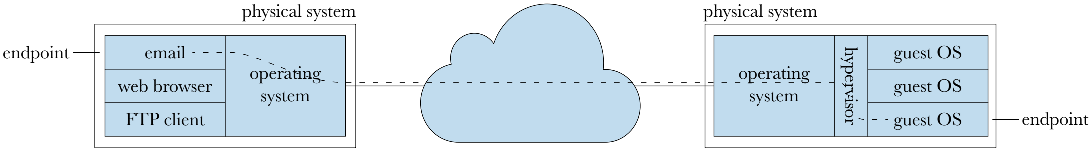


Figure 50 *Endpoints*

With this hypothetical scenario in mind, we can define a flow as being a combination of addressing, protocol, forwarding domain and priority information. If all of the available information of these types from a packet are somehow combined to form a unique numerical value, then that value serves as a reasonable proxy for a flow. If another packet yields a different proxy value because one or more of these flow-identifying values differs, then that packet can be safely said to belong to an independent flow.

Flow identification is allowed to be imperfect. False associations are allowed. What this means is that it is okay to map two separate flows to the same flow-identifying proxy value. But, it is not okay for two different packets from the same actual flow to map to two different flow-identifying proxy values. If this were to happen, they may be queued and scheduled separately and end up being forwarded in an order that differs the order in which they were transmitted by the originating system.

Imperfect mapping with false associations (i.e., collisions) and not false divisions exactly describes a hash function.

By hashing as many of the flow-identifying field values from a packet as possible, the resulting hash value serves as a very serviceable flow-identifying proxy value. This value may be used to assign packets to flow groups, the members of which must be forwarded in exactly the same order as the one in which they were received. Packets belonging to different flow groups may be forwarded without concern for their relative order.

The issue of packet order is important in the context of load balancing because forwarding packets to a common destination via two or more separate paths through the network is a perfect opportunity for packets to arrive out of order at their destination.

In the scenario depicted in Figure 51, there are two forwarding systems connected by parallel paths. The upstream forwarding system receives two packets in sequence: a long packet followed by a short packet. Both are addressed to the same ultimate destination, but take separate paths in an effort to maximize efficiency. The long packet is received first by the upstream forwarding system and, after a brief processing and queuing delay, is forwarded onto one of the two parallel paths. The second, shorter packet is received next and it, too, is forwarded after an equally brief processing and queuing delay. However, this shorter, second packet is transmitted onto the second path between the two forwarding systems because that second path was idle at the time. As shown in Figure 51, the short packet completes its transmission from the upstream forwarding system to the downstream forwarding system while the long packet is still in the midst of being transmitted. The downstream forwarding system dutifully forwards the short packet after a brief processing and queuing delay, and only then can it forward the long packet after it's been completely received. Thus, the order of the two packets is reversed.

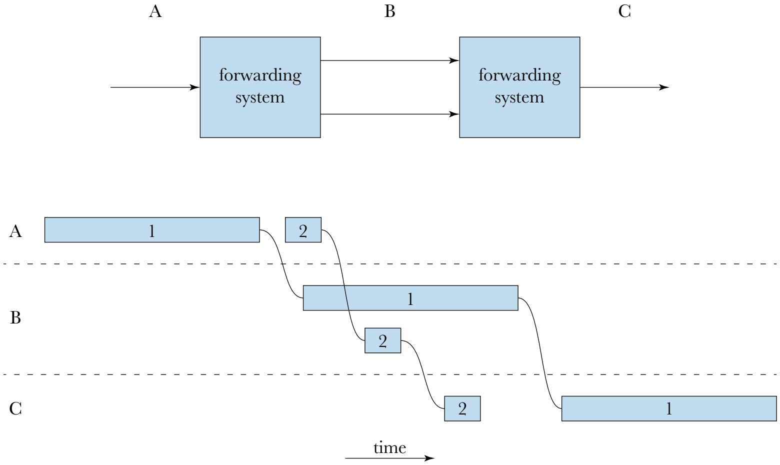


Figure 51 Parallel Paths Causing Packet Mis-Ordering

As was described previously, flows are identified by hashing packet header fields that pertain to flow membership. Examples of these fields include source and destination addresses and port numbers, ethertypes and protocol IDs, VLAN IDs and priority values, as well as metadata such as physical and logical receive port IDs. Headers both before and after a packet's forwarding header are useful for load balancing purposes. The headers before the forwarding header help identify where the packet came from. Headers after the forwarding header help identify the packet's ultimate destination and the process to which the packet belongs. Taken together, these values reliably map a packet to a flow. Fields that must be explicitly excluded from a load balancing hash computation include length fields, sequence numbers, checksums, opcodes and flags that may change from packet to packet within the same flow. Including this dynamic information in a load balancing hash will cause packets within a flow to take diverging paths to their destination—all of the packets will get where they're going, but their relative order cannot be assured.

Though it is generally true that the more fields that are included in a flow-identifying hash computation the better, there are situations where incorporating fewer fields yields better results. An excellent example of this scenario is the incompatibility between load balancing and IPv4 fragmentation. When an IPv4 packet is fragmented due to packet lengths exceeding an MTU somewhere along the packet's path, then all of the headers that may follow the IPv4 forwarding header now only exist in the first fragment; the remaining fragments only convey further payload data. If a load-balancing hash is computed as deeply as a possible, then it is conceivable that the first fragment may include fields from the following headers (in order): Ethernet, IPv4, UDP, VXLAN, Ethernet, IPv4. However, for the second through the last fragment of the packet, it is impossible to parse past the outer Ethernet and IPv4 headers. This means that the first fragment and all of the subsequent fragments will have different hash results, mapping those packets to different flows. This, ultimately, introduces the potential for those fragments to arrive at their destination out of order. If fragmentation cannot be avoided, the only practical solution is to limit the hash computation to just the outer Ethernet and IPv4 headers in this example (i.e., the encapsulating and forwarding headers).

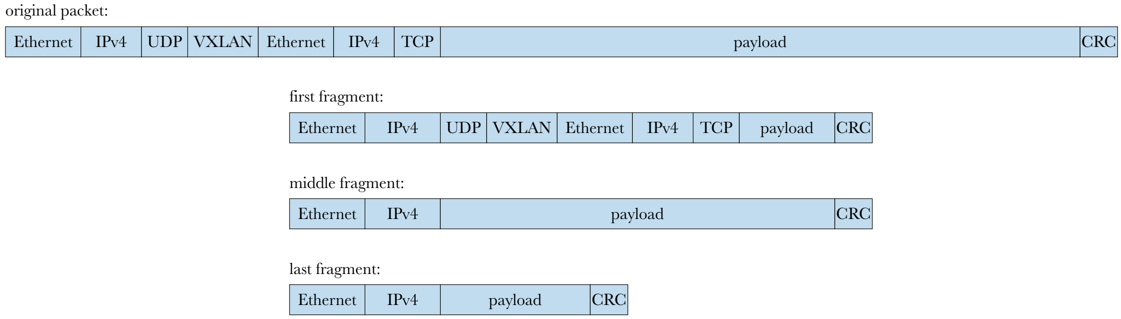


Figure 52 IPv4 Fragmentation and the Availability of Packet Headers

Even without the problems associated with IPv4 fragmentation shown in Figure 52, it is possible that certain header fields may tend to bias hash results such that some paths are favored over other paths. This can be difficult to predict and a certain amount of trial and error is required. Because of this, it is very helpful to have fine-grained controls over which fields are included in the load balancing hash computation.

Presuming that a good hash has been computed, how do we go about selecting a forwarding path for a packet based on that value? Let us presume that a list of *n* potential forwarding paths is available. It is simply a matter of scaling the hash value to be evenly distributed across the values 0 through *n*-1. This is most simply done by treating the hash value as a fraction that ranges from 0 <= *hashValue* < 1 and then multiplying *hashValue* by the number of entries in the transmit path options list. That product is then used as an index into the list.

Hierarchical Load Balancing

It is both possible and reasonable for there to be a hierarchy of load balancing operations applied to a packet. For a simple example, one or more of the transmit path options for an ECMP operation (ECMP is described in detail further below) to be a LAG (LAGs are also described in detail further below). This is not particularly challenging to implement in actual hardware. The challenge is to make proper use of the hash value to ensure that each stage of load balancing operates independently of all of the others.

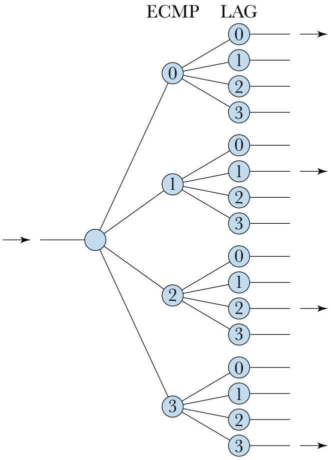


Figure 53 *Hierarchical Load Balancing—Accidental Correlation*

Let's consider a simple example. In this example, there is a 4-way ECMP where each ECMP option is a 4-way LAG. If we took a very simple approach to the use of our hash value, we'd essentially use the most significant two bits of the hash value as our index into the two consecutive transmit option lists. Now, if, for example, option number two is taken for the ECMP, then option number two is also taken for the subsequent LAG. This means that the LAG (the second of two load balancing operations in our example) will always choose just a single transmit path option, leaving the other three unused.

Ideally, the path choice at each level of hierarchy is uncorrelated with all of the other levels. To solve the accidental correlation shown above, it is necessary to use different bits of the hash value with each load balancing operation. Hence, after performing the ECMP load balancing operation, the hash value must be rotated by some number of bits or transformed in some other way so that new entropy may be used by a subsequent load balancing operation.

Dealing with Link Failures

The number of next-hop paths associated with a single forwarding destination may vary over time. Paths become unavailable due to link or equipment failures. Those failures are eventually repaired. Paths are added and removed in order to optimize physical resources. Ideally, these changes impact only the flows associated with the paths that are changing state. For example, if a path fails, it is reasonable to map all of the flows that were using that path to all of the remaining paths. The flows formerly associated with the failed path are certainly impacted because they are now following new paths and their destinations may see some packet loss and certainly some out-of-order packet arrivals around the time of the transition. All of the flows associated with the surviving paths, however, are only impacted to the extent that the addition of the re-mapped flows may increase the utilization of paths carrying those flows.

The problem is a bit more complex when a path is added to a load-balancing group. In this case, it is desired to take advantage of the new path by remapping some fraction of all of the flows on all of the previously existing paths to the new path. Thus, if p paths were active before a new path was added, then the number of flows that should be remapped is approximately $1/(p+1)$. The affected flows are likely to experience some packet mis-ordering during the transition.

The naïve approach to dealing with a path failure is to change the modulo of the hash value to match the new number of active paths. This certainly works, but it has the unfortunate side effect of potentially remapping every flow on every path and not just the ones associated with the failed path.

A preferred approach is to remap just those flows associated with the affected path. One way to do that is to perform the normal modulus of the hash value across the original number of paths, p. Then, for all of the packets that map to the failed path, compute a new hash value using p-1 as the modulo and then use that value to choose from among the remaining paths. For example, assume we have a 5-way load balancing group and one of the members fails. Start by load balancing as if all of the paths were still operational using a modulo of 5. Then, for every packet that is mapped by modulo 5 to the failed path, use the original hash value and apply a modulo of 4 and then use that value as an index into a list of paths that excludes the failed path. This method assures that only the flows associated with the failed path are affected. When the failed path is brought back on line, it is simply a matter of no longer performing the second hash modulus.

An alternative to the two-step hash modulus approach is to apply a modulus to the hash value that is significantly larger than the number of available paths, say four times the number of paths. The output of the hash modulo function is then used as an index into a table of path ID values. If the ratio of hash modulus values to the number of paths is, for example, 10:1, then the table has 10 copies of each path ID value. When a path fails, then all of the table's entries that point to the failed path are updated to point to various operational paths. If the number of table entries pointing to the failed path is sufficiently high, then the flows previously using the failed path can be distributed among the remaining operational paths.

There are many ways of achieving essentially the same results described above. The important point is to try to limit the impact of the loss of a path to just 1/*p* flows; specifically, only the flows assigned to the failed path should be impacted by the path failure.

The Limitations of Flow-Hash Load Balancing

It is important to point out that the load balancing techniques described here are imperfect. It is essentially impossible to guarantee that all of the transmit path options will be utilized fully and evenly. This is true because of two fundamental problems: hashing does not assure even distribution of outcomes, and not all flows are created equal.

The header field data for a complex and exhaustive flow-identifying hash function may span hundreds of bits. The hash function may, ultimately, map these multi-hundred bit values to just a few bits in order to make a one-of-*n* choice where *n* may be on the order of 16 (much smaller and larger *n*s are possible). There can be no assurance that, say, 1,024 flows are evenly distributed across our hypothetical 16 transmit path options at 64 flows per option. A certain imbalance must be expected.

These imbalances are compounded by the fact that there is no assurance that all flows are going to offer the same bandwidth demands on the network. Flows that are quite small (tens of Mbits per second) may be mixed with very large flows (tens of Gbits per second). If just a few of the large flows are mapped to a single transmit path option, then that path may suffer congestion and packet loss while other paths are lightly utilized.

One way to address the inherent probabilistic nature of hash-based load balancing is to attempt to be flow-state aware. With this approach, flows are detected and characterized and moved from path to path as necessary to optimize the equality of path utilization. This method of load balancing can be done either exhaustively or approximately.

The exhaustive approach requires that each forwarding entity detect the starts and ends of flows by monitoring, for example, the TCP messages that are being passed between endpoints. Whenever a TCP Sync[7] message is detected, the associated flow may be assigned to the least busy available path. All packets belonging to that flow continue to use that same path until a TCP Finish message is detected. This approach, of course, requires that the forwarding entity maintain the state of, potentially, millions of TCP connections. In most cases this is impractical.

The approximate approach uses hashing of flow-identifying header fields and applies a modulo that yields a series of hash buckets that is far greater than the number of available paths. Then, for each hash bucket, state is maintained that tracks the activity of the group of flows that correspond to each bucket. Whenever all of the flows associated with a hash bucket cease to be active for a period of time that is greater than the propagation delay to the convergence point of the multi-path network, those flows become candidates for being remapped from busy paths to less-busy paths. This technique depends on gaps between bursts of packet being large enough and frequent enough that there is a good probability that all of the flows associated with a hash bucket will have overlapping gaps in their bursts. This particular load balancing method is vulnerable to improvements in the efficiency of transport protocols. If those gaps go away, so does this method.

Link Aggregation Groups

Simple LAG

In its simplest form, a link aggregation group (LAG) is a narrowly-defined and limited form of load balancing. A LAG is intended to replace a single, point-to-point physical connection with multiple, parallel point-to-point connections. What this means in practice is that a LAG is considered an inseparable bundle of links that must all originate at a single forwarding system and terminate at a single other forwarding system. The links within a LAG are not permitted to take separate paths from the origin of the LAG to its termination.

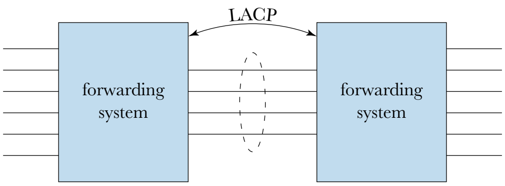


Figure 54 A Simple LAG

[7] TCP is explained in Chapter 12: Connections.

LAG Configuration Protocol (IEEE 802.1AX)

Point-to-point physical connections are Ethernet's realm. And, as bridging behavior is inextricably associated with Ethernet, it follows that the spanning tree protocol (STP) will detect multiple, parallel links between pairs of bridges and disable all but one of the links in the aggregation group. To prevent the pruning of redundant links by STP, the link aggregation control protocol (LACP) is used to automatically configure Ethernet bridging systems such that groups of physical interfaces (i.e., Ethernet ports) are presented to the Ethernet forwarding entities (i.e., bridges) as single ports. What this means in practice is that a list of physical interfaces are represented by a single interface ID for both packet reception and transmission. Because the group of links is perceived by the bridge as a single, indivisible link, then no loop exists to be detected and disabled. For example, a broadcast Ethernet packet received via one of the links in a LAG will not be flooded to the remaining links of that LAG.

LACP operates by periodically sending LACPDU messages via the link-level discovery protocol (LLDP) back and forth between directly-connected Ethernet bridges. LLDP messages are never forwarded and are assured to be terminated by the receiving bridge. The LACPDU messages convey link ID values. A bridge receiving these messages can determine which of its physical interfaces are collectively and directly connected to a single other Ethernet bridge. Physical interfaces that meet these criteria are candidates for link aggregation. Once a link aggregate has been defined and configured, the associated Ethernet bridge forwarding entity sees all of the interfaces in the aggregate as a single interface—i.e., a single interface ID value is passed to the bridge with each packet received via any of the interfaces in the aggregate and, in the transmit direction, the link aggregation function is free to choose any of the member interfaces for a particular aggregate ID based on a packet's flow association.

Multi-Chassis LAG and Shortest-Path Bridging (IEEE 802.1aq)

Simple link aggregation between pairs of Ethernet bridges addresses one particular aspect of network reliability: link failure. But it does not address another: system failure. If a big chunk of a network's traffic is conveyed via an aggregated link through a single Ethernet bridge, that network is protected in the event of a link failure (i.e., the traffic can be distributed across the remaining links) but is not protected in the event of that single bridge's failure. When that bridge goes down, it's going to take every member of the link aggregate with it. To address this, multi-chassis LAG (commonly, MC-LAG) was developed, illustrated in Figure 55.

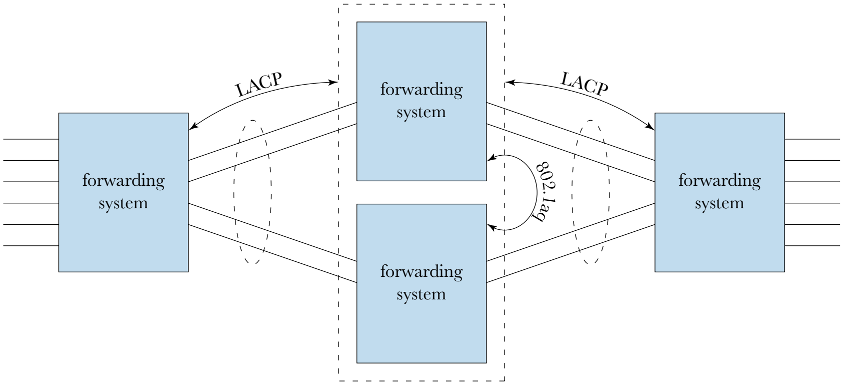


Figure 55 *Multi-Chassis LAG*

> NOTE Even though the word “chassis” is on the tin, it doesn’t mean that the forwarding system must adopt a chassis form factor. The forwarding system may be a fixed configuration system or even a single-chip system.

Like a conventional LAG, MC-LAGs must originate and terminate all of the aggregated links at common systems. However, unlike a conventional LAG, the links of an MC-LAG may pass through two or more independent Ethernet bridges between a common source and a common destination.

The trick to making MC-LAG work is to have the multiple chassis communicate with one another such that LACP operates in a normal manner. In other words, the LACP messages received by a single bridge from a group of two or more bridges all appear to be coming from a single bridge. Hence, the bridge receiving these messages freely configures multiple links connected to multiple bridges together as a single aggregated link.

The shortest-path bridging standard (IEEE 802.1aq) applies concepts and protocols from the routing world to the bridging world in order to make efficient use of multiple paths and the use of protocol-based forwarding database updates where, previously, spanning tree vigorously disabled redundant links and automatic address learning dictated packet forwarding patterns without an awareness of the overall network topology or the shortest and most efficient path through that topology. One of the fringe benefits of 802.1aq is that it brings multi-chassis LAG along for the ride.

Equal-Cost, Multi-Path Forwarding

Equal-cost, multi-path (ECMP) forwarding is analogous to LAGs, except that, by default, the multiple paths that run from a divergence point to a convergence point are managed by routing protocols such as IS-IS and BGP (see Chapter 19 on page 321) protocols associated with Layer 3 forwarding using IP and MPLS. Though the protocols associated with LAGs vs. ECMP and the protocols used to set up and manage multiple paths through a network may differ, the packet-by-packet behavior of the two is the same. Specifically, the flow-identifying fields from as many headers as possible are run through a hash function. The output of that function is a flow-identifying proxy value that may aggregate some number of flows together. A packet's flow-identifying proxy value is then mapped in an equal or weighted manner to one of the available paths that can convey the packet toward its destination. And, any of those paths chosen by an ECMP function may be a collection of paths where a further ECMP (or LAG) function is required to make a choice from among the options. There is no predefined limit to the number of levels of hierarchy of ECMPs and LAGs that may be configured. If a packet is facing multiple potential paths to its destination and each of those paths entails entering, say, two tunnels and each of those tunnels also face multi-way ECMP choices, then a cascade of three ECMP (or ECMP-like) functions is required to help ensure that the flows are distributed across all of those available paths.

It is, perhaps, tempting to visualize (and depict) ECMP options as all corresponding to physical interfaces and cables emanating from a forwarding system. However, in practice this need not be the case. There is nothing about ECMP that compels those paths to follow several independent physical paths. Indeed, several ECMP path options may map to a common physical transmit interface and medium. What distinguishes these ECMP path options is not their physical path but their virtual path—i.e., the values that identify the tunnels into which the packets have entered. A single physical path conveying a multitude of tunnels may diverge into several separate physical paths at a point in the network further downstream.

9 Overlay Protocols

At the time of the writing of this book there was a tremendous amount of activity related to massively-scaled data centers and the virtualization of their resources. These data centers are often hosts to many thousands of independent tenants or clients who rent resources from the data center owner/operator. The resources available include compute, storage and network access.

It is inevitable that tenant activity ebbs and flows over daily, weekly and seasonal timescales. Also, tenants themselves come and go with the rising and falling fortunes of business on the Internet. To be cost effective, the owner/operators of these data centers must be able to quickly respond to their tenants' demands while not allowing expensive resources to sit idle or be underutilized.

If hundreds of tenants are seeing randomly distributed bursts of activity, it is inefficient to allocate dedicated physical resources (i.e., servers) to each tenant. It is far more efficient to have a single physical server run several copies of the tenants' operating system of choice using virtual machine technology. With a virtual machine, a hypervisor function runs on top of the host machine's native operating system. The hypervisor, in turn, presents an operating environment that appears to the several copies of guest operating systems as if each were running on its own dedicated hardware. The hypervisor manages access to physical resources so as to avoid conflicts and maintain secure isolation between the guest operating systems.

With data centers having the ability to dynamically "spin up" virtual servers as needed, it also makes sense to provide data centers with the means to move virtual machines from one physical machine to another in order to balance loads and ensure that, instead of having thousands of servers operating at an average load of, say, 50%, 60% of the servers should operate at 80%, while the remaining 40% are powered down to save electricity and reduce wear and tear.

With virtual machine mobility within a data center the need for large-scale network virtualization also arises. As we've previously seen, IEEE 802.1Q VLANs provide the means for isolating Layer 2 bridged traffic into virtual networks. However, with its 12-bit `vlanId` value, it is limited to just 4,094 virtual networks. This is far from adequate for modern data centers. In response to this need for network virtualization at a large scale, a number of so-call "overlay" protocols were developed. These protocols all share common behavioral characteristics (i.e., they behave a lot like IEEE 802.1ah Provider Backbone Bridging) and only differ in the details. Examples of these protocols include:

- UDP + VXLAN
- NVGRE
- UDP + GENEVE

All three of these overlay protocols fall under a general framework known as Network Virtualization Over Layer 3 (NVO3) as described by IETF RFC 7365. In a nutshell, Layer 2 (Ethernet) or Layer 3 (IPv4 or IPv6) services are provided to a (potentially) large number of independent tenants in a data center by tunneling through (overlying on top of) a Layer 3 network infrastructure (IPv4, typically).

With these overlay protocols, the data center's network is divided into two levels of abstraction. At the lower "underlay" level, IPv4 or IPv6 over Ethernet is used as a foundation upon which the tenants' virtual networks are built. The underlay level uses routing protocols such as BGP (or IS-IS) and ECMP to take advantage of all of the redundant pathways that exist in the physical infrastructure.

The overlay protocol header that's interposed between the outer headers (e.g., Ethernet/IP) and the inner payload headers (e.g., Ethernet/IP or just IP) provides, most essentially, a tenant or virtual network identifier value that's wide enough to allow not just thousands of such overlays, but millions. This virtual network identifier is used to provide a forwarding domain for the payload packet; preventing traffic from one forwarding domain from accidentally being received by another. Thus, multiple tenants may have overlapping MAC address, VLAN ID or IP address numbering spaces without concerns about conflicts or confusion.

Each virtual machine in a physical machine (server) has its own MAC address and coordinates access to shared resources such as a network connection through the hypervisor function. If two virtual machines need to exchange packet data, they can do so using Ethernet packets (with IP payloads, if desired). The hypervisor maintains a table that associates each virtual machine with a virtual network identifier. This association is used to build the overlay header and to identify the IP address of the physical system that hosts the hypervisor of the packets' intended destination virtual machine. The resulting final packet is shown in Figure 56.

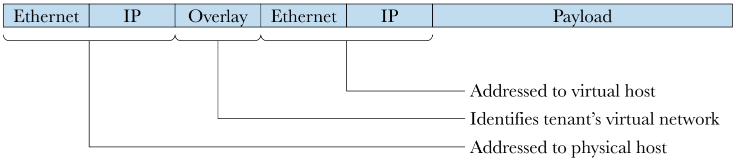

Ethernet
IP
Overlay
Ethernet
IP
Payload
Addressed to virtual host
Identifies tenant's virtual network
Addressed to physical host

Figure 56 *Typical Overlay Application*

When the packet shown above is received by the physical system that hosts the intended virtual machine destination, that physical system's hypervisor checks the overlay header's virtual network identifier to validate that at least one of the configured virtual machines is associated with that virtual network. Finally, the inner Ethernet header is used to forward the packet to the addressed virtual machine within the specified virtual network.

It is important to point out that the overlay encapsulation and decapsulation both occurred within the hypervisors at either end of the connection. So, in this example, the underlay network need not be aware of nor even natively support the chosen overlay network protocol. This is, of course, yet another example of fundamental tunneling behavior.

There are, however, a number of scenarios where the origination and termination of an overlay tunnel may need to reside within the data center network itself. For example, if a data center network that uses overlays must communicate with another network or with endpoints that do not support such overlays, then the entry/exit point for the overlay must reside within one or more of the forwarding systems that make up the data center network.

As with any tunnel entry/exit point, the data center overlay entry/exit points must have forwarding databases of sufficient scale to map virtual machine addresses to virtual network identifiers and to physical machine addresses. Depending on the location of the overlay network entry/exit point in the overall network topology, that forwarding database may need to support hundreds of thousands of addresses.

If an overlay protocol is providing Ethernet tunneling services (i.e., an Ethernet packet is the payload of the underlay network and the overlay header), then the tunnel origination and termination points must provide the usual Ethernet bridging services: address learning and aging; flooding; etc. When an overlay tunnel exit point receives and decapsulates a payload packet, it must submit the inner Ethernet header's `sourceAddress` value to a learning/aging lookup operation. If that lookup fails (i.e., the `sourceAddress` value is unknown by the tunnel termination point in the context of the virtual network identifier), then that `sourceAddress` value must be added to the forwarding system's forwarding database. The "receive interface" identifier that must be associated with the newly-learned `sourceAddress` value is the outer IP header's `sourceAddress` value. When this is done, a reciprocal packet entering a tunnel in the opposite direction at the same point will benefit from a successful destination lookup that maps the packet's `Mac.destinationAddress` value to an IP `destinationAddress` value for use in the new, outer encapsulating header. The packet is then forwarded based on that new IP destination address.

Ethernet, of course, must also support the flooding of tenant packets with unknown unicast, multicast, or broadcast `Mac.destinationAddress` values. The flooding of these packets within a tenant's virtual network is emulated at Layer 3 using IP multicast. If the underlay network does not support multicast distribution, then ingress replication must be used. In other words, if a virtual machine (server) must send a multicast packet to a thousand other virtual machines, then its hypervisor must create and transmit one thousand copies of the packet and transmit them in a unicast fashion to each of the intended recipients.

The bottom line is that these overlay protocols are, essentially, tunnels that are, conceptually, not significantly different from any other tunnel type. There's an outer set of headers (one or more) that defines the path through the network where the tunnel is instantiated. There's a tunnel payload that is opaque to the tunneled network infrastructure. Then, of course, there's the payload packet that has its own forwarding header(s) for getting each packet to its intended destination once the payload emerges from the tunnel through which it was being conveyed. So, aside from the presence of the overlay header itself, this describes pretty fundamental tunneling behavior.

Then why does the overlay header exist? It exists to support network virtualization at a scale that cannot be achieved using standard VLANs. Simply tunneling through a data center network from one hypervisor to another does not, by itself, provide any meaningful isolation between tenants of the data center. IEEE standard VLANs could be pressed into service here, but there are two problems here: one minor and one major. The minor problem is that adding a VLAN header to a tenant's packet involves modifying the tenant's packet. This may not sit well with some tenants. The major problem, though, is that the 12-bit `vlanId` value only supports 4,094 virtual LANs (remember, values `0x000` and `0xfff` cannot be used to identify VLANs). In a modern data center, this is far too limiting. That's where the overlay header comes in. With its typically 24-bit virtual network identifier, up to 16 million virtual networks may be defined. In practice, the virtual network identifier value is used in exactly the same manner as a `vlanId` value; it is mapped to a forwarding domain identifying value which is then combined with address values when forming search arguments for forwarding lookups.

With that overview of how overlay networks are used in data center networks out of the way, the next few sections delve into the details of a few of the more prominent and popular standards.

User Datagram Protocol (UDP)

Where TCP (see Chapter 12 on page 167) provides reliable data transport—in order, intact—UDP provides a simple portal between two (or more) processes. TCP is commonly used for things like file transfers and other unicast data transfer applications, while UDP is used for streaming applications (unicast or multicast) where data loss may be tolerated such as IP telephony or streaming video.[8]

[8] At the time of this writing, the vast majority of streaming video on the Internet is unicast video on demand (e.g., Netflix, *et al*) that relies on a TCP connection rather than UDP.

A header diagram is shown in Figure 57 and the field definitions of a UDP header are listed in Table 23.

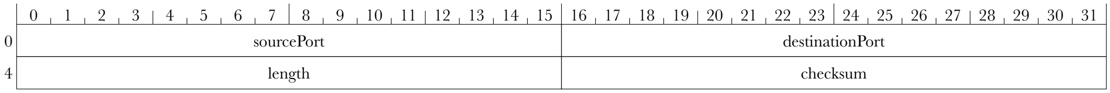


Figure 57 *UDP Header Format Diagram*

Table 23 *UDP Header Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`sourcePort` (Source Port)	16 bits	0.0	The origin process of a UDP packet. This field generally identifies the originating endpoint of communication within an operating system. This value is optional as it is not always meaningful. `sourcePort` is used to convey load-balancing entropy in a number of applications. If the `sourcePort` value should be ignored, a value of `0` is used.
`destinationPort` (Destination Port)	16 bits	0.16	The terminating process of a UDP packet. This field generally identifies the terminating endpoint of communication within an operating system.
`length` (Length)	16 bits	4.0	The length, in bytes, of the UDP header and its payload. The minimum value for this field is 8 (i.e., the length of a UDP header).
`checksum` (Checksum)	16 bits	4.16	The checksum of the IP pseudo-header, UDP header, and the UDP payload data. An IP pseudo-header isn't a real header. It's a collection of fields from an IP header that are included in the checksum computation. The included fields are: source and destination addresses, protocol, and length. Refer to IETF RFC 768 for details. The use of `checksum` is optional for IPv4. When not used, `checksum` must be set to `0`. The use of `checksum` is mandatory for IPv6. Fortunately, the authors of the VXLAN specification have given us a reprieve by recommending against computing a checksum when encapsulating a VXLAN packet and ignoring a non-zero checksum when decapsulating a VXLAN packet.

While not generally thought of as an overlay technology, UDP does serve as a useful tunneling glue layer between forwarding protocol header and the overlay header. For example, an IP/UDP packet may have an MPLS/IP/TCP packet as its payload. In these types of applications, the `destinationPort` field is used as a next-header identifier instead of a process identifier in the destination endpoint. When used as an overlay glue layer, the `sourcePort` value is often used for load balancing entropy.

UDP + VXLAN

The VXLAN, or Virtual Extensible Local Area Network, protocol is described by IETF RFC 7348. A VXLAN header never stands alone in the midst of a packet. It is always preceded by a user datagram protocol (UDP, IETF RFC 768) header, as shown in Figure 58.

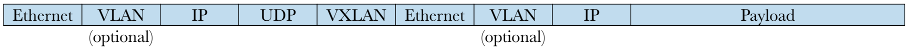


Figure 58 *Typical VXLAN Header Stack*

A UDP header is universally preceded by an IPv4 or IPv6 header. An IP `nextHeader` value of `0x11` is used to indicate that a UDP header follows an IP header. UDP headers are typically used to channelize an IP connection between two endpoints. These channels are identified by the UDP header's `sourcePort` and `destinationPort` fields. The port numbers (not to be confused with physical ports) identify the processes or protocols that are the originators or terminators of messages. In the case of VXLAN, a `destinationPort` value of `4789` is used to indicate that a VXLAN header immediately follows the UDP header.

There's no need to identify the process or protocol that is the source of a VXLAN packet, so `Udp.sourcePort` is used instead to convey some load balancing entropy in the form of a hash of the inner headers' flow-identifying fields. This is helpful because the data center underlay network may be built with forwarding systems that don't understand what to do with a VXLAN header, but they'll likely know how to parse a UDP header, extracting the `sourcePort` and `destinationPort` values for use in its ECMP load balancing hash computations. The entropy hash value should be mapped into the TCP/UDP dynamic/private port number range of 49,152–65,535.

UDP headers may be followed by a wide variety of header types. It is, after all, a generalized transport method. A VXLAN header, on the other hand, must always be preceded by a UDP header.

The VXLAN overlay header's primary role is to convey the virtual network identifier value. The format of a VXLAN header and the definitions of its fields are shown in Figure 59.

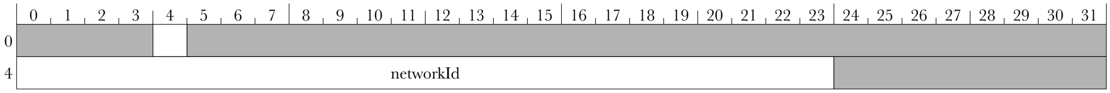


Figure 59 *VXLAN Header Structure Diagram*

Table 24 *VXLAN Header Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`networkIdValid` (I)	1 bit	0.4	Indicates that the VXLAN header conveys a valid `networkId` value. For normal operation—where the overlay traffic is being segregated into separate virtual networks—this bit must be set to 1 to indicate that a valid `networkId` value is present. If this bit is set to 0, then no valid virtual network identifier is present.
`networkId` (VXLAN Network Identifier)	24 bits	4.0	The virtual network ID value. When entering a VXLAN overlay network, the original packet's destination address (typically, `Mac.destinationAddress` and, if present, `vlanId`) is looked up in a forwarding database. A successful match returns both the forwarding instructions (i.e., the transmit port number) and a new forwarding domain value. This new forwarding domain value is then mapped to the `networkId` value for use during packet encapsulation prior to transmission.

Overall, the VXLAN header and concept are pretty straightforward. At the tunnel ingress point, the packet's destination address is used to determine both the packet's transmit port and the `networkId` value to use. At the tunnel egress point, the `networkId` value (and possibly a `vlanId` value from the original packet) is used to define the packet's forwarding domain. The forwarding domain is combined with the original packet's `Mac.destinationAddress` value for a normal lookup in a forwarding database. Like all Ethernet bridging applications, the `Mac.sourceAddress` from the original packet (i.e., the encapsulated packet) must be submitted to a lookup for address learning and aging purposes. The `networkId` value is used to define the packet's forwarding domain (just as was done for the destination lookup). If the lookup is successful (i.e., a matching key is found), then the packet's source interface is checked in order to detect a MAC move situation. In the case of a VXLAN tunnel exit point, the encapsulating (i.e., outer) IP header's `sourceAddress` is used as the packet's source interface. This is sensible since it is to that IP address that a reply packet must be forwarded. At a VXLAN tunnel entrance the packet's physical receive ports are associated with the inner Ethernet header's `sourceAddress` value in the normal bridging manner.

Just about the only area where one might pick some nits with the VXLAN standard has to do with its VLAN tag handling in the payload packet. The VXLAN RFC essentially says that the payload packets should be devoid of VLAN headers unless the VTEP (i.e., the VXLAN tunnel endpoints) support those VLAN headers. Pretty weak sauce, that! At least the RFC is unambiguous in declaring that VXLAN tunnel endpoints must ignore any VLAN tag and solely use the `networkId` value to determine the identity of the overlayed virtual network. Essentially, there is an expectation that the `vlanId` value from the payload's VLAN header has already been factored into the choice of `networkId` values. In other words, if a particular tenant has a need for VLAN behavior and inserts VLAN headers into their Ethernet packets, then that tenant must be allocated multiple `networkId` values; one for each `vlanId` value.

Sharp-eyed readers will notice that the VXLAN header is missing a field that all well-designed headers should have: some kind of next-header field. When VXLAN was first defined, it was presumed that Ethernet+IP would be all that it would convey and that, therefore, the only header that would immediately follow a VXLAN header would be an Ethernet header. This was not an unreasonable assumption given that the dominant virtual machine technology at the time employed hypervisors that addressed the various virtual machines using Ethernet MAC addresses, even though the messages passed between virtual machines were invariably contained within IP packets.

To address this oversight and open up VXLAN to a broader array of applications, work is underway to extend VXLAN to include a next-header-like field. This effort is known as the generic protocol extension, giving rise to VXLAN GPE. The VXLAN GPE header is shown in Figure 60 with its field definitions listed in Table 25.

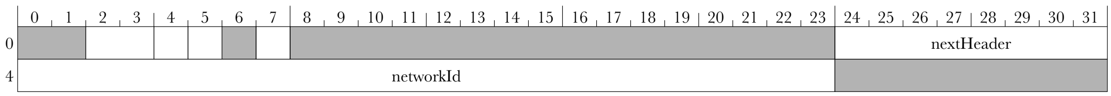


Figure 60 VXLAN GPE Header Structure Diagram

Table 25 *VXLAN GPE Header Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`version` (Ver)	2 bits	0.2	The VLAN version. This field indicates the version of the VXLAN header/ protocol. The current version number is `0`.
`networkIdValid` (I, for "Instance Bit")	1 bit	0.4	Indicates that the VXLAN header conveys a valid `networkId` value. For normal operation—where the overlay traffic is being segregated into separate virtual networks—this bit must be set to `1` to indicate that a valid `networkId` value is present. If this bit is set to `0`, then no valid virtual network identifier is present.
`nextHeaderValid` (P)	1 bit	0.5	Indicates that the `nextHeader` field is present and contains a valid value.
`oam` (O)	1 bit	0.7	Indicates that the payload packet is an OAM packet. If this bit is set to `1`, then the packet that is the payload behind the VXLAN GPE header is an OAM packet and must be processed as such by the receiving VXLAN tunnel endpoint.
`nextHeader` (Next Protocol)	8 bits	0.24	Identifies the type of the payload packet. If `nextHeaderValid` is set to `1`, then this field identifies the type of the payload conveyed behind the VXLAN GPE header. Setting this bit to `1` also requires that `destinationPort` of the preceding UDP header be set to `4790`. If `nextHeaderValid` is set to `0`, then the payload packet type must be Ethernet. This field is enumerated as follows: 0 = Ethernet 1 = IPv4 2 = IPv6 3 = Network Service Header
`networkId` (VXLAN Network Identifier)	24 bits	4.0	The virtual network ID value.

VXLAN GPE isn't, as it appears to be, a simple extension of VXLAN where previously-reserved bits are given a definition. No, VXLAN GPE is actually a distinct protocol that stands separate from VXLAN. Even though, with all of the new options set to zero, a VXLAN GPE header is indistinguishable from VXLAN, the two are distinct. The most striking evidence of this is that the `Udp.destinationPort` value that identifies VXLAN is `4789` while it's `4790` for VXLAN GPE. The new `Udp.destinationPort` value (`4790`) must be used if `nextHeaderValid` is set to 1. Otherwise either `4790` or `4789` may be used.

The most remarkable attribute of VXLAN GPE is its `nextHeader` field which allows a variety of payload types, freeing VXLAN from being limited to conveying just Ethernet-encapsulated packets. If the preceding UPD header's `destinationPort` value is set to `4790` and `nextHeaderValid` is set to `1`, then `nextHeader` identifies the type of the header that follows the VXLAN GPE header.

The `oam` bit is set to `1` to indicate that the payload packet is an OAM packet and must be processed accordingly. The current specification of VXLAN GPE is silent on what exactly that means.

NVGRE

Network virtualization using generic routing encapsulation (i.e., NVGRE, IETF RFC 7637) is pretty much defined by its own name. Conceptually nearly identical to VXLAN, NVGRE provides a network virtualization value in a header that is interposed between a tunnel layer (i.e., the underlay) and the payload packet (i.e., the overlay).

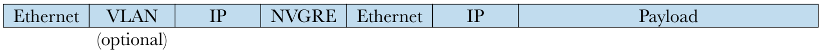


Figure 61 Typical NVGRE Header Stack

The structure and field definitions for NVGRE are shown in Figure 62 and listed in Table 26.

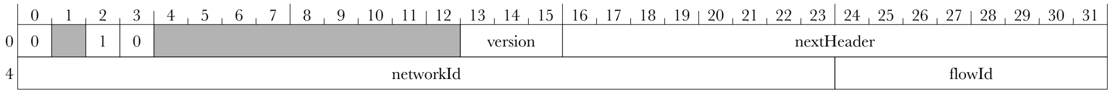


Figure 62 NVGRE Header Structure Diagram

Table 26 *NVGRE Header Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`version` (Ver)	3 bits	0.13	The NVGRE version number.
`nextHeader` (Protocol Type)	16 bits	0.16	Identifies the next header's type. This value is analogous to an ethertype value, but uses different encodings. For NVGRE's typical application of tunneling Ethernet packets over an IP infrastructure, the value `0x6558` is used to indicate transparent Ethernet bridging.
`networkId` (Virtual Subnet ID (VSID))	24 bits	4.0	The virtual network ID value. This value is used to distinguish one tenant from another in a data center network and is analogous to a VLAN ID.
`flowId` (FlowID)	8 bits	4.24	Per-flow entropy. This field is used for load balancing purposes with ECMP. Typically, a hash of the flow-identifying header fields of the payload packet are used to populate this field. This obviates the need for transit devices to parse beyond the NVGRE header. If a valid flow ID value is not computed, `flowId` must be set to `0`.

NVGRE is, for all intents and purposes, simply an application of GRE. In fact, both GRE and NVGRE are identified by the same IP `nextHeader` value: `47`. The only difference between a GRE header and an NVGRE header is that a few of the flag bits are set to constant values—`checksumPresent = 0`, `keyPresent = 1`, `sequenceNumberPresent = 0`—and the 32-bit `greKey` field is split into the 24-bit `networkId` and 8-bit `flowId` fields.

One interesting difference between VXLAN and NVGRE is that, while VXLAN allows for an optional VLAN header in the payload Ethernet packet, NVGRE specifically prohibits including a VLAN header in the payload Ethernet packet. An NVGRE tunnel entrance must strip any such VLAN header from the payload Ethernet packet (the `vlanId` value may be incorporated into the determination of the `networkId` value) and a NVGRE tunnel exit must discard any packet whose Ethernet payload packet includes an VLAN header.

GENEVE

Generic network virtualization encapsulation (or, GENEVE) can be described as VXLAN GPE with options. At the time of this writing, it is in draft form at the IETF.

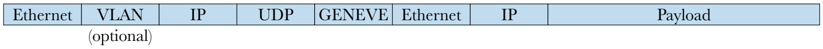


Figure 63 *Typical GENEVE Header Stack*

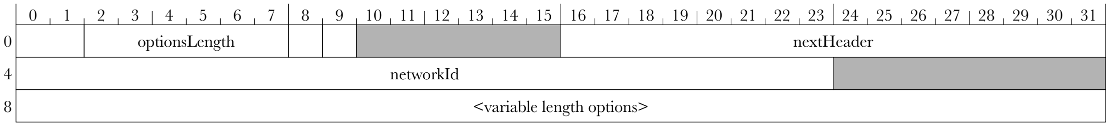


Figure 64 *GENEVE Header Structure Diagram*

Table 27 *GENEVE Header Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`version` (Ver)	2 bits	0.0	The GENEVE version number. The GENEVE header version is currently `0`.
`optionsLength` (Opt Len)	6 bits	0.2	Indicates the length of the option fields. This length value is expressed in multiples of 32-bit words; excluding the two 32-bit words that make up the baseline GENEVE header. If this value is set to zero, then no options are present. The offset of the start of the payload packet can be determined using this value.
`controlPacket` (O)	1 bit	0.8	Indicates that the packet contains a control message. If this bit is set to `1`, then a the payload packet is conveying a control message. Control messages are always sent between GENEVE tunnel endpoints. Therefore, such messages must never be forwarded by a tunnel exit endpoint. If this bit is set to `0` then the payload packet is a normal data packet.
`criticalOptionsPresent` (C)	1 bits	0.9	Indicates that critical options are present in the header's options section. If this bit is set to `1`, then one or more options have their own `criticalOption` bit set to `1`. Terminating endpoints that detect this condition must process the critical options. If said endpoint is unable to process the critical options, it must discard the packet. If this bit is set to `0`, then an endpoint may strip off all of the options and forward the encapsulated tunnel payload packet in a normal fashion.
`nextHeader` (Protocol Type)	16 bits	0.16	Identifies the next header's type. This field is, essentially, an ethertype field. For the typical GENEVE application, a value of `0x6558` is used to indicate transparent Ethernet bridging. Other values representing other payload types are allowed.
`networkId` (Virtual Network Identifier (VNI))	24 bits	4.0	Identifies a data center tenant's virtual network. This value is used to distinguish one tenant from another in a data center network and is analogous to a VLAN ID.

The structure and field definitions of the GENEVE tunnel options header are shown in Figure 65 and listed in Table 28.

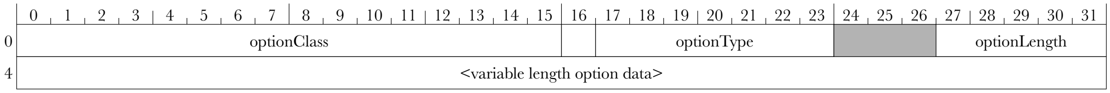


Figure 65 *GENEVE Tunnel Options Header Structure Diagram*

Table 28 *GENEVE Tunnel Options Header Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`optionClass` (Option Class)	16 bits	0.0	The option's "name space." Individual tenants, vendors, etc. may have a unique ID value assigned that is used to establish a private name space. Thus, an entity that controls a `optionClass` value may assign `optionType` values without fear of colliding with any other entity's `optionType` values.
`criticalOption` (C)	1 bit	0.16	The critical-option indicator. If this bit is set to `1`, then the option is critical and must be processed by a GENEVE receiving tunnel endpoint and not forwarded. If a GENEVE tunnel endpoint is unable to process a critical option, it must discard the packet. If this bit is set to `0`, then the option is not critical and may be ignored by a GENEVE endpoint.
`optionType` (Type)	7 bits	0.17	Indicates the option type.
`length` (Opt Len)	5 bits	0.27	Indicates the length of the option data field. This length value is expressed in multiples of 32-bit words; excluding the 32-bit word that makes up the baseline GENEVE tunnel option header. If this value is set to zero, then no option data is present. The offset of the start of the next option header (if any) can be determined using this value.

Like VXLAN GPE, GENEVE rides on top of UDP, utilizes `sourcePort` for flow entropy and features a 24-bit virtual network identifier. Unlike VXLAN GPE, GENEVE stipulates that an Ethernet payload packet must not include a VLAN header, the `vlanId` value of any previously existing VLAN header having been incorporated into the derivation of the `networkId` value.

GENEVE differs from VXLAN GPE in the arrangement of its fields and, most significantly, its support for a variable number of options appended to the header. The base GENEVE header includes a `optionsLength` field which, if non-zero, indicates that one or more options are present. The individual options are type-length-value (TLV) variety, meaning that each option header has a type value (`optionType`), a length value (`length`) that is used to either interpret or skip over individual options, and an option value itself (the option data that immediately follows the GENEVE option header).

The motivation for supporting the conveyance of more information than a simple 24-bit virtual network identifier is based on the observation that large, chassis-type routers with embedded, proprietary fabrics often include a prodigious amount of metadata with each packet as they are forwarded from an ingress packet forwarding engine (PFE) to an egress PFE. By supporting a potentially large amount of application-specific metadata, GENEVE could be used to make a data center's underlay network act very much like a single router of tremendous scale.

10 Virtual Private Networks

Virtual private networks (generically, VPNs) are essentially tunneling applications that are intended to emulate either wires (e.g., PWE3), Ethernet bridges (e.g., VPLS), or Ethernet bridges and routers (e.g., EVPN) across a network such as the Internet. When these tunneling methods are considered with respect to what it is that they're emulating—wire, bridge, router—it becomes a fairly straightforward task to rationalize and understand their behavior.

Wires don't concern themselves with what they convey. They are, for the most part, dumb pipes. When emulating a wire, the intent is to provide a point-to-point private connection across a packet-oriented network (e.g., IP, MPLS) that can support any kind of communication. As originally envisioned, the emulation of wires was a means to allow telecom companies to build packet-based networks that could support their valuable legacy circuit-switched businesses.

When moving from emulating wires to emulating bridges, things get quite a bit more complicated. Wires are point-to-point (shared media wires such as the original coax Ethernet standard are not emulated) whereas bridges are multi-point to multi-point. Bridges also must support the replication of packets in the case of broadcast, unknown-destination, and multicast traffic. And, of course, transparent bridges must support the learning and aging of MAC source addresses.

At the highest level of sophistication, VPNs support the emulation of sophisticated routers. It's not that forwarding packets based on IP or MPLS header information automatically make a router function sophisticated. Rather, it is the support for ECMP, multi-homing and split horizon checks (and other things) that make a router function sophisticated. Briefly, ECMP and multi-homing (akin to a multi-chassis LAG) provide load balancing across multiple paths (see Chapter 8 on page 115) while split horizon checks prevent flooding packets back to their source. With these capabilities, the emulation of a complex network function by a geographically disperse network becomes practical.

Like all tunneling methods, VPNs require an encapsulation and, like any kind of multi-tenant or multi-customer service, a means for identifying a specific instance of a virtual network so as to isolate tenants or customers from one another.

The following sections provide an overview of three representative VPN technologies. There are, of course, many more VPN types since inventing new tunneling methods is a favorite pastime for networking engineers. The three chosen for this book are: PWE3, VPLS, and EVPN.

Pseudowire Emulation Edge-to-Edge (PWE3)

PWE3 (specified by IETF RFCs 3916, 3985 and others) describes a basic architecture for emulating a circuit or wire tunnel across a packet-based network such as IPv4, IPv6 or MPLS. These RFCs don't go into the specifics of the on-the-wire formats and encapsulations, but, instead, describe the operational requirements and the means for establishing and maintaining a pseudowire connection. For a specific example of a pseudowire, IETF RFC 4448 serves as a useful example.

RFC 4448, entitled "*Encapsulation of Ethernet over MPLS*" does pretty much exactly what the title says. The intent is to provide a wire-like connection between two geographically separate Ethernet networks such that the two networks behave as if they were connected via a simple point-to-point Ethernet segment.

Like all PWE3-compliant pseudowire methods, the Ethernet over MPLS pseudowire follows the prescribed encapsulation architecture as depicted in Figure 66.

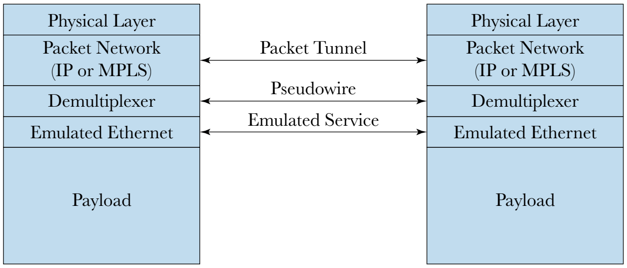


Figure 66 *PWE3 Protocol Stack Reference Model*

In the reference model shown in Figure 66, the physical layer may be anything that can convey IP or MPLS packets. In modern networks, this is invariably Ethernet. The tunnel itself is the underlying IP or MPLS network represented by either an IP header (IPv4 or IPv6) or one or more MPLS headers. Several pseudowires may be configured to operate simultaneously over the top of the underlying IP or MPLS packet network. Hence, this layer is associated with some form of demultiplexer—i.e., a value that identifies a particular pseudowire. Finally, the contents of the pseudowire are layered on top in the form of Ethernet packets or any other suitable data transport.

For Ethernet over MPLS (RFC 4448), these generic layer types are replaced with specific types. Following an Ethernet header for the physical layer, an MPLS header is used to specify the label switch path of the packet. This label is swapped at each label switch router hop across the underlying tunnel network. An MPLS header at the bottom of the MPLS header stack is used to demultiplex the pseudowire; the `label` value is interpreted as a pseudowire ID. An MPLS control word (see "*Control Word*" *on page 107*) or associated channel header (see "*Generic*

Associated Channel Header (G-ACh)" on page 108) may optionally follow the MPLS header stack. The Ethernet packet then follows the MPLS header stack and optional control word or generic associated channel header. This header structure is diagrammed in Figure 67.

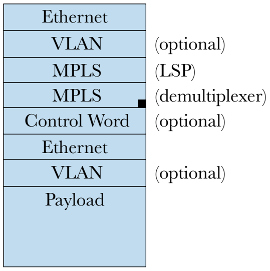


Figure 67 Ethernet Over MPLS Header Stack

The essential elements for Ethernet over MPLS are all present in the Figure 67. The outer Ethernet header provides router-to-router transport while the MPLS header at the top of the MPLS header stack is examined and swapped as needed to forward the packet from the tunnel entrance to its exit. The MPLS header at the bottom of the stack contains the pseudowire ID value (analogous to a virtual network ID used in the overlays discussed in the Chapter 9).

The Pseudowire Control Word and ECMP

The optional control word (which could, instead, be a pseudowire associated channel header) is there to support ECMP. Control words and pseudowire associated channel headers offer two different solutions to properly supporting ECMP. A control word does not indicate the type of the MPLS payload. This means that a transit MPLS router (an LSR) likely does not have information in its forwarding database to reliably determine the MPLS payload's type. Consequently, any attempt to parse beyond the MPLS header stack is merely a best guess and it is entirely possible to erroneously include non-flow header fields into a flow hash, sending packets belonging to a particular flow along disparate paths. To remedy the inevitable mis-ordering of packets that this causes, the `sequenceNumber` field is used by the router at the MPLS tunnel exit termination point to re-order the packets into the correct order prior to transmission.

Re-ordering packets when a large number of pseudowires are in use represents a very challenging design problem. Hundreds of thousands of pseudowires may require reordering services at a single point. This implies that hundreds of thousands of sequence number state values must be maintained and that many packets from each of the hundreds of thousands of pseudowire may need to be quarantined by the router while the arrival of packets missing from the correct sequence are awaited. A more sensible solution is to use a pseudowire-associated channel header.

The pseudowire associate channel header includes a `channelType` field that is roughly analogous to a `ethertype` field, positively identifying the MPLS payload type. With the payload type reliably identified, a greater range of flow-identifying packet headers may be parsed and hashed, yielding good ECMP load balancing and obviating the need for sequence numbers and reordering.

Another option for supporting pseudowire ECMP is to insert an entropy label (EL) and an entropy label indicator (ELI) somewhere in the stack of MPLS headers.

Pseudowires and VLANs

The Ethernet over MPLS pseudowire standard defines two modes for handling VLAN headers in the customers' (i.e., payload) packets: raw mode and tagged mode. These modes are configured by the service provider on the network edge equipment that mark the pseudowire origination and termination points.

VLAN headers in the payload Ethernet packet may originate either with the customer or with the service provider. A service provider adds a VLAN header to the payload packet if multiple customers are sharing a Layer 2 path to the pseudowire. These service provider VLAN headers are known as "service delimiting" VLAN headers.

In raw mode, service delimiting VLAN headers must be stripped from the packet prior to entering the pseudowire tunnel. It is presumed that the `vlanId` value from the stripped header had previously been used to map to a particular customer-specific pseudowire. If a packet contains a non-service delimiting VLAN header (i.e., the VLAN header originated with the customer), then the VLAN header is preserved. In other words, the only VLAN headers that may be carried across the pseudowire are customer VLAN headers.

In the tagged mode (which is the default mode for Ethernet over MPLS pseudowires) every packet must have a service-delimiting VLAN header. If a packet arrives at the pseudowire entrance without such a VLAN header, then a dummy VLAN header is added to the payload Ethernet packet ahead of any customer VLAN headers.

In both modes of operation, customer VLAN headers are transmitted as is and without modification by the pseudowire exit point.

In the tagged mode of operation, service-delimiting VLAN headers may be stripped or have their `vlanId` values remapped by the pseudowire exit points.

Pseudowires and Ethernet Flow Control

Although a pseudowire is supposed to emulate a physical wire (or fiber, if you prefer), it's not a perfect emulation. Though a number of link level (e.g., LLDP) and bridge control protocols (e.g., RSTP) work just fine, there's one in particular that's just not compatible with a pseudowire: IEEE 802.3x link flow control (and 802.1Qbb priority flow control). The reason that link-level flow control breaks down across a pseudowire is that the delays across the pseudowire far exceed those of an actual physical wire. As a consequence, if a receiving endpoint is being overwhelmed by a transmitting endpoint on the far end of a pseudowire, the reaction time due to the round trip delay of the flow control message going one way and the packets already on the wire continuing to arrive in the opposite direction means that the amount of data that can continue to arrive after transmitting a pause packet can be quite significant (and certainly far larger than the designer of the receiving endpoint was anticipating).

To remedy this, 802.3x and 802.1Qbb flow control packets must be terminated by the service provider's pseudowire edge equipment: pausing and buffering packets heading toward the customer's endpoint for as long as the flow control state remains in effect. Conversely, a service provider's pseudowire edge router may issue flow control message packets toward customer endpoint if its packet buffers are becoming congested due to a speed mismatch between the customer's Ethernet connection and the pseudowire or if a congestion indication has been received from the service provider's MPLS network.

Virtual Private LAN Service (VPLS)

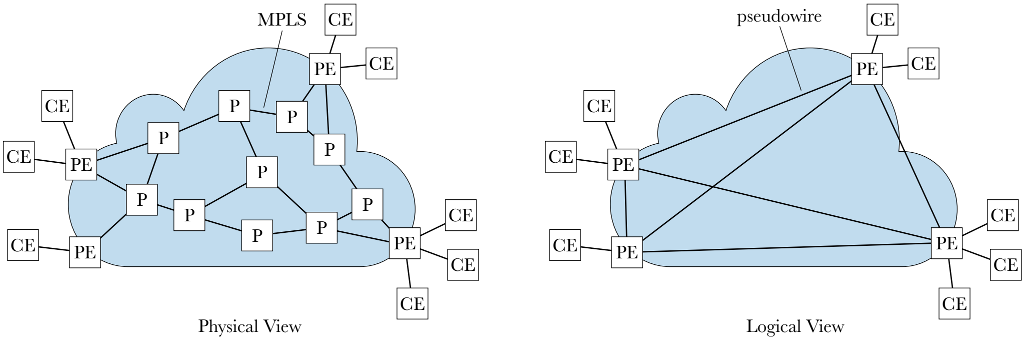
The virtual private LAN service takes the point-to-point concept of pseudowires and expands it to support multipoint-to-multipoint service. Instead of emulating a wire, the underlay network emulates an Ethernet bridge.

VPLS is specified by two separate and incompatible standards: IETF RFC 4761 and RFC 4762. The difference between them—and the cause of their incompatibility—is the signaling methods used to manage the network. RFC 4761 uses the border gateway protocol (BGP) for auto-discovery and signaling whereas RFC 4762 uses label distribution protocol (LDP) signaling. Fortunately for those of us who are mostly concerned with the data part of a network versus its control part, the two versions of VPLS are essentially identical.

VPLS leverages RFC 4448 Ethernet over MPLS pseudowires to interconnect a series of provider-edge (PE) systems. A representative model of a VPLS network is shown in Figure 68.

The left side of the Figure 68 shows the physical view of a hypothetical VPLS network. Provider edge routers (PE) are distributed around the perimeter of the service provider's network. A number of provider network core routers (P) are used to interconnect the PE systems. Customer edge systems (CE) are connected to nearby PE systems. The CE systems serve as the customer networks' interface to the service provider's network.

The right side of Figure 68 shows the logical view of the same network. In this view, a number of pseudowires have been configured through the service provider's network such that every PE is connected through a pseudowire to every other PE in a full mesh network.



Physical View

Logical View

Figure 68 *VPLS Network Model*

VPLS is intended to make a service provider's MPLS network look and act like a geographically dispersed Ethernet bridge (from the customer's perspective). This means that all of the usual Ethernet bridging behaviors are expected: forwarding based on `destinationAddress`; learning and aging `sourceAddress` values; and the flooding of broadcast, unknown-destination and multicast (BUM) traffic.

Forwarding customer Ethernet packets across a VPLS network is fairly straightforward. A CE sends a packet to its locally-attached PE router. The PE router receives the packet and uses the interface via which the packet was received to determine to which VPLS instance the packet belongs. If the packet has a VLAN header, that may also be considered in associating the packet with a VPLS instance.

The receiving PE examines the Ethernet packet's `destinationAddress` via a lookup into a forwarding database. This lookup returns the identity of the egress PE to which the packet must be forwarded. This information is used to determine the correct pseudowire to use to convey the packet to the egress PE. The Ethernet packet is encapsulated into an MPLS packet according to RFC 4448 (Ethernet over MPLS) and is transmitted via an interface that gets the packet to the pseudowire's next hop. At the egress PE, the MPLS label switch path is terminated and its payload Ethernet packet is exposed. The bottom-of-stack demultiplexing MPLS label provides the Ethernet packet payload with its forwarding context. The Ethernet packet's `destinationAddress` is again submitted to a forwarding lookup to determine to which, of possibly several attached CEs, the packet must be forwarded.

Of course, for a PE to successfully look up a MAC destination address, its forwarding database must be populated with relevant MAC address values. This is done through the usual Ethernet bridging process of detecting and learning `Mac.sourceAddress` values and the interfaces associated with them. What makes the VPLS implementation of bridging behavior a bit unique is that each `Mac.sourceAddress` value must be learned twice. When an Ethernet packet arrives at an ingress

PE from a CE, its `sourceAddress` value is submitted to a source address lookup. If no match is found, a new entry is created for that PE's forwarding database that associates the just-learned `sourceAddress` value with the interface via which the packet was received. At the egress PE, the payload Ethernet packet is decapsulated from the MPLS pseudowire and its `sourceAddress` value is submitted to a source address lookup. If the lookup fails to find a match, the `sourceAddress` value is added to the egress PE's forwarding database that associates the just-learned `sourceAddress` value with the *pseudowire* via which the packet was received.

Aging is, of course, simply a lack of learning. Every time a PE successfully looks up a MAC source address, its timestamp is reset to the current time. If a forwarding database entry's most recent timestamp exceeds some amount of time (five minutes, by default) from the current time, that entry is removed from the database.

BUM traffic (broadcast, unknown-unicast and multicast) must be flooded within the VPLS instance. This means that an ingress PE (one that receives a packet from a CE) must send a copy of the packet to each PE that belongs to the same VPLS instance. Then, all of the egress PEs that receive a copy of that packet must make further copies for all of the attached CEs that belong to the same VPLS instance as the packet.

If the PEs in a VPLS network behaved like normal Ethernet bridges, then there would be one of two problems: either spanning tree would prune the network of all of its redundant pseudowires (severely impacting the efficiency and performance of the VPLS network), or the BUM packets received from a pseudowire would be replicated back to all of the other pseudowires (leading to an endless storm of BUM packets). Fortunately, VPLS PE routers do not act like normal Ethernet bridges; they treat their MPLS-facing pseudowire interfaces as being in a separate and distinct category from the CE-facing Ethernet interfaces. This split between these two categories of interfaces is known as split horizon.

Split horizon forwarding is used to prevent loops in networks with redundant paths. One of the requirements of VPLS is that all of the PEs that are associated with a particular VPLS instance must be fully connected with one another with a mesh network (i.e., point-to-point connections are formed for every possible pair of PEs in a VPLS instance). If you look at a diagram of a mesh network between *n* points, what you're seeing is *n* separate tree structures; each with *n*-1 leaf nodes. When an ingress PE must flood a BUM packet to all of the egress PEs in the same VPLS instance, it is taking advantage of that single tree network instance associated with the ingress PE and is counting on the egress PEs to not then forward any of the BUM packets onto their own trees; for, if they did so, egress PEs would receive multiple copies of the BUM packet, which is certainly not allowed. Split horizon forwarding is diagrammed in Figure 69.

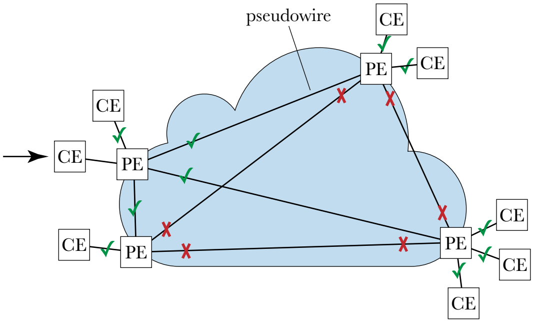


Figure 69 *VPLS Split Horizon Forwarding*

In the split horizon example shown in Figure 69, there is a single VPLS instance that includes all of the CEs. A BUM packet is received by a CE device which forwards it to its associated ingress PE device. The ingress PE floods the packet to all of the interfaces associated with the VPLS instance. In this example, those interfaces include the one that leads to the second CE associated with the ingress PE as well as the other three PEs (i.e., the egress PEs). The egress PEs receive their copies of the BUM packet (suitably encapsulated as per RFC 4448) and then flood the decapsulated Ethernet packet *only* to the attached CEs and *never* to any of the pseudowires that lead to the other PEs. Thus, the egress PEs are making a distinction between the two categories of interfaces—pseudowires to PEs and attachment circuits to CEs—and only allowing packets received from a pseudowire to flow in a single direction: toward the attached CEs. This means that each BUM packet is distributed by a simple tree carved out of a full mesh.

Scaling VPLS with H-VPLS

PEs in a VPLS network must learn all of the MAC addresses of all of the customer Ethernet-based endpoints for every VPLS instance associated with each PE. In large networks this can lead to forwarding database sizes that are in the millions of entries. Another scaling challenge is that each VPLS instance requires a full mesh of pseudowires between each of the VPLS instance's PEs. With n PEs, $n(n - 1)/2$ pseudowires are required. Forwarding database space and encapsulation parameter storage space must be allocated within the PEs to all of these pseudowires. To address these scaling concerns, hierarchical VPLS (H-VPLS) was developed.

In hierarchical VPLS, each PE system is connected to one or more aggregation systems instead of directly to the CE systems. This aggregation system can be thought of as a remote PE and is often installed in the customers' premises. The connection between the aggregation system and a PE is known as a "spoke."

The primary benefit of hierarchical VPLS is that an additional layer of tunneling is used to simplify the scale and complexity of the full mesh that exists at the core of the VPLS network and to reduce the table scale burden of the PE systems. The aggregation systems encapsulate the customer's packet in whatever Layer 2 tunneling scheme is desired where the new encapsulation (i.e., the new outer header) is addressed to the egress aggregation system instead of to the actual customer endpoint. Thus the customer's addresses are hidden from the core of the VPLS network and do not need to be learned. The VPLS network only sees the packets coming from and going to the aggregation systems, significantly reducing the scale of the VPLS network.

Hierarchical VPLS is yet another example of how the proper application of tunneling can be used to build larger and larger networks without having to increase its complexity as perceived by a single point in the network.

Ethernet Virtual Private Network (EVPN)

Just as VPLS represents the natural evolution of Ethernet over MPLS pseudowire service—introducing multipoint-to-multipoint forwarding and bridge emulation—EVPN represents the natural evolution of VPLS.

Though VPLS is quite capable and, at the time of this writing, in fairly widespread use, it is missing a few functions and capabilities that would enable it to make much more efficient use of the underlying MPLS infrastructure. In particular, VPLS does not support all-active multihoming (I'll explain what multihoming is in just a bit) and it does not take advantage of MPLS's ability to support multicast replication,[9] relying, instead, on the ingress PE to replicate packets for all intended egress PEs. EVPN (IETF RFCs 7209 and 7432) was developed specifically to address these shortcomings and make MPLS VPNs much more scalable, efficient and reliable.

Multihoming

Multihoming is, quite simply, the practice of providing a customer with multiple, redundant service provider connections. This is illustrated in Figure 70.

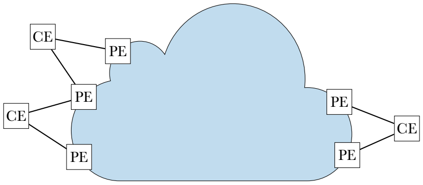


Figure 70 *Multihoming*

It is very common for a single provider edge (PE) system to be connected to the customer edge (CE) systems of multiple customers. This is simply how a network

[9] For those of you who are uncertain about what multicast is all about, the next chapter is entirely devoted to that very topic.

scales. However, since an organization's external network connections to the Internet and to remote facilities are essential for that organization's daily operations, it is too risky for such a vital resource to rely on just a single connection to the provider's network: a single fiber, a single pair of optics, or a single PE router. If any of those components fail, then the organization is cut off until the fault can be corrected.

The obvious solution is to provide two or more redundant connections from a CE to two or more PEs as illustrated in Figure 70. Unfortunately, since VPLS emulates a bridge, the spanning tree protocol will disable all but one of the links in order to prevent loops. Of course, if that remaining active link were to fail, spanning tree would activate one of the other links. However, during normal operation, the disabled links sit idle, representing wasted resources.

Allowing a CE to be connected to multiple PEs with multiple active links is known as all-active, or active-active multihoming, as opposed to single-active or standby-active. The obvious benefits of this approach are that the customer gets both a nice increase in attachment circuit bandwidth and the resiliency of redundant connections.

After a brief detour into how EVPN obviates the need for full-mesh pseudowires, we'll return to the topic of taking advantage of all-active multihoming.

Eliminating the Full Mesh

A full-mesh interconnect between all of the PE systems participating in a particular VPLS instance is a hallmark of VPLS networking. By taking advantage of the logical one-hop connectivity from any point to any other and simple split-horizon rules, emulating bridging behavior is fairly simple.

The problem with the full mesh is that, from any particular point to any other particular point, only one path is allowed, even though the underlying physical network may have a large number of redundant paths. Significant opportunities for bandwidth efficiency and resiliency are lost by not taking advantage of those redundant paths. One of the key attributes of EVPN is that it takes full advantage of all of the physical connections in a service provider's network rather than paring down those connections to a series of pseudowires that form a full mesh.

The implication of the move from a full mesh to multi-path interconnect—from the perspective of the provider edge and provider core routers—is that ECMP load balancing must be performed on each packet in order to distribute flows of packets among the available paths such that those paths are as fully and fairly used as possible. EVPN uses a multi-protocol variant of the border gateway protocol (MP-BGP) to distribute routing information among the service provider's routers. This routing information serves two purposes: it defines all of the possible paths from any point to any other point for each EVPN instance, and it distributes customer Ethernet MAC addresses and reachability information.

That's right: rather than rely on the independent learning and aging of customer Ethernet MAC addresses by the various PE routers, EVPN relies on MP-BGP to distribute this information. This process is described in the Bridge Emulation section, further below.

Another benefit of moving from a full mesh of pseudowires to multi-path interconnect is that multicast forwarding—i.e., the receipt of a single packet and the transmission of multiple copies of that packet to specific destinations—can be pushed from the ingress PE router to a point deeper in the heart of the service provider's network. With this move, it is possible to configure a multicast replication tree that is distributed across the service provider's network. Each node in the tree represents a point where two or more copies of a packet are transmitted for each one received. Ideally, no link in a network ever carries more than one copy of any packet.

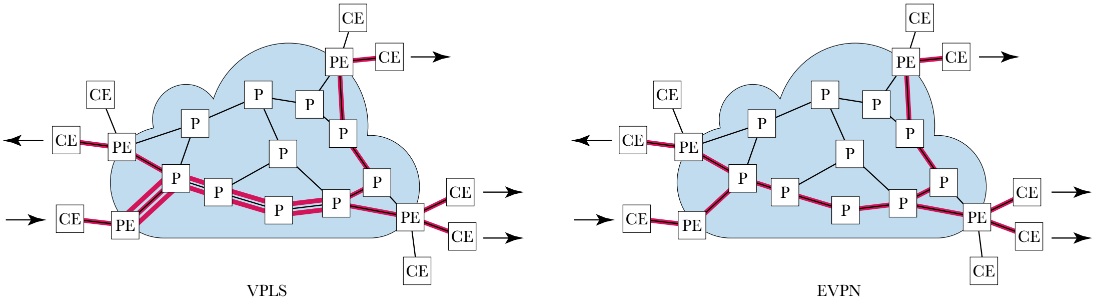


Figure 71 *EVPN Multicast*

The VPLS network illustrated in Figure 71 relies on ingress replication to distribute copies of a packet to multiple transmit interfaces. Note that the ingress PE at the lower left receives one packet and transmits three copies to the first P router; each destined for a separate egress PE router. None of the P routers make any further copies, but notice that several P routers must forward two copies of the packet along a common path until a divergence point is reached.

Now, compare that behavior to the EVPN example on the right. Here, a single copy of the ingress packet is forwarded by the ingress PE at the lower left to the first P router on the path. That P router then transmits one copy for each of the two diverging paths through the service provider's network. This means that a single copy of the packet proceeds to the right toward the next divergence point where, again, two copies of the packet are transmitted: one each for the two diverging paths that lead to the two egress PE routers on the right side of the cloud.

The EVPN multicast model makes much more efficient use of the service provider's physical network infrastructure by delaying replication as long as possible.

The example shown above isn't terribly dramatic. But, just imagine if an ingress PE router had to flood a packet to 100 (or 1,000) egress PE routers. You can begin to see the benefits of the EVPN approach.

The flooding (or selective distribution) of BUM (broadcast, unknown-unicast, multicast) packets is just one aspect of Ethernet bridging. Another important aspect is the management of the forwarding databases. That is covered immediately below.

Bridge Emulation

EVPN has a very different forwarding database management method from VPLS. In VPLS, each PE is responsible for independently detecting and learning MAC source addresses. This can, unfortunately, lead to some performance and stability problems when MAC addresses move from PE to PE or when PEs go offline or online. To address this, EVPN relies on MP-BGP to distribute MAC addresses to the PE routers.

When an ingress PE receives a packet from a CE that has a `Mac.sourceAddress` value that is unknown to that PE, it learns the address locally (and associates the packet's receive interface with that MAC address) and then passes the addressing information off to the MP-BGP controller which, in turn, distributes it to all of the PEs that participate in the same EVPN instance. The other PE routers add the MAC address and next-hop forwarding information to their forwarding databases as static entries (i.e., not subject to aging). The original ingress PE is responsible for updating timestamp information for the newly learned MAC address to determine when a sufficient amount of time has passed since last observing the MAC address so that it may be aged out of the forwarding database. MP-BGP is, again, pressed into service to inform the various PE routers of any MAC addresses that have aged out or that have moved from one PE to another. This active distribution of MAC addresses makes EVPN much more responsive to changes and much more stable and reliable than VPLS.

All-Active Multihoming

A customer edge (CE) system's use of multiple connections to multiple provider edge (PE) routers is analogous to multi-chassis link aggregate groups (MC-LAGs). Specifically, a single source of packets (a CE) is connected via multiple Ethernet links to separate and largely independent forwarding systems (PEs) and flow-identifying hash computations are used to assign packets to each of the available parallel paths. The key difference between simple MC-LAG and EVPN is that the two or more PEs that are connected to a common CE also have connections to each other, meaning that BUM packets may be forwarded from the initial receiving PE to another PE to which the packet's source is also attached. That second PE then dutifully forwards the packet back to its origin. This, of course, must be prevented.

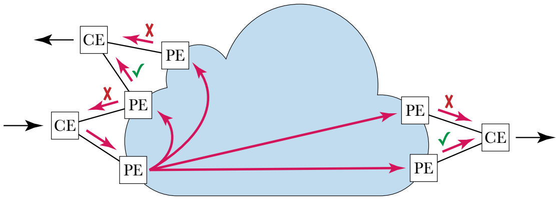


Figure 72 All-Active Multihoming

Figure 72 illustrates two classes of problems. The first is that one of the PEs forwards the packet back to its origin. The other is that two PEs independently forward the packet to a shared, multihomed CE. Each problem requires its own solution.

The solution to the first problem—that of a BUM packet being echoed back to its origin—is a classic split-horizon problem and is solved in EVPN using what's known as an Ethernet segment identifier (ESI). This is simply a value that uniquely identifies an Ethernet segment within an EVPN instance. This value is encoded into a non-forwarding MPLS label. An Ethernet segment consists of all of the Ethernet links that are connected to a particular CE system. So, the two links that extend from the CE that is the origin of the packet in Figure 72 both belong to the same Ethernet segment and packets received by any PE attached to those links are assigned the same Ethernet segment identifier. When a PE router receives a packet from another PE, it checks the packet's Ethernet segment identifier and blocks transmission onto any interface that is associated with that identifier, solving the first accidental replication problem.

For the second problem—two egress PEs independently forwarding copies of a BUM packet to a common CE—a "designated forwarder" is used. The idea is fairly straightforward. The designated forwarder concept is based on the observation that traffic between PEs and CEs contain VLAN headers in the customer packets. It is then simply a matter of allocating VLAN IDs to each of the PEs associated with a particular Ethernet segment. When an egress PE receives a multi-destination packet (broadcast, unknown-unicast, multicast), it only forwards those packets whose VLAN IDs have been configured as part of its designated forwarder set. Therefore, it is guaranteed that only one copy of each packet is received by each CE. Unicast packets with known unicast destination addresses are unaffected by designated forwarder tests.

Both of these solutions to accidental multihoming replication problems involve discarding packets instead of forwarding them. This, of course, means that unnecessary copies of those packets were created in the first place. It would be, of course, very desirable for a network to only spawn exactly as many copies of a multi-destination packet as necessary. Achieving this means that the extra effort involved in creating those copies and in forwarding them can be saved.

Unfortunately, creating exactly the right number of copies turns out to often be quite impractical.

To accomplish this kind of precision packet replication would require the replication and distribution of a massive amount of forwarding database and system state information. For example, in a large EVPN network, a replication P node could be located fairly close to an ingress PE. This replication P node would have to have a copy of the designated forwarder information for every PE and Ethernet segment to which it has any kind of access. Also, to react quickly to state changes (e.g., a PE router going offline), the replication P node would have to have both primary and secondary forwarding databases. Making matters worse, if the replication P node is far from the egress PEs, then the signaling time increases, making the reaction time to failures much worse.

So, even though it seems to be (and is) inefficient to go through the effort to copy and transport packets that will just be discarded, it is the best option when all of the implications of doing otherwise are considered.

EVPN Over VXLAN

EVPN is most closely associated with MPLS as its transport tunnel encapsulation. This is reasonable since EVPN is viewed as an evolution of MPLS pseudowires and VPLS. In recent years, however, EVPN over VXLAN has emerged as a popular solution for building efficient, scalable data center networks. With just one exception, VXLAN serves as just as effective a tunneling technology as MPLS. That one exception is split horizon.

Split horizon checks in EVPN over MPLS makes use of an MPLS header whose `label` value is used to encode an Ethernet segment identifier. When forwarding BUM packets, the Ethernet segment identifier carried by that MPLS header is compared with the Ethernet segment identifier associated with an intended transmit interface. If the two values match, the packet is discarded.

VXLAN does not have a field that can be used to encode an Ethernet segment identifier, so another means must be devised. The recommended solution is for each of the VXLAN tunnel endpoints (the equivalent of a provider edge system in EVPN) to keep track of the IP addresses of all of the Ethernet segments of the other VXLAN tunnel endpoints with which it has shared, multi-homed Ethernet segments (i.e., multiple IP addresses per VXLAN tunnel endpoint). Each IP address in the egress VXLAN tunnel endpoint's database is then associated with an Ethernet segment that terminates at the current VXLAN tunnel endpoint. When a packet is received from the VXLAN overlay network, its source IP address from the outer IP header is examined and compared to the IP addresses stored as just described. When a match is found, the packet is filtered from being replicated onto those transmit interfaces associated with a matching IP address from the local database, thus preventing a packet from being received from and transmitted onto the same Ethernet segment.

11 Multicast

Multicast has been referred to a number of times in preceding chapters. In this chapter, we'll dive into the details of the concepts, implications, addressing and protocols associated with multicast traffic. But first, a quick definition of the term "multicast." Multicast simply means the selective forwarding of multiple copies of a packet to multiple destinations. Multicast is not broadcast or flooding, which are both the copying of a packet to *all* possible destinations. The word "selective" is a very important part of multicast's definition.

BUM Packets

Not all of the packets that require multicast-like treatment are multicast packets. Broadcast packets—which are a degenerate form of multicast—are addressed to all possible destinations instead of all interested destinations. The distinction is important. Multicast strives to only send copies of a packet to those destinations that are interested in receiving such a packet. Broadcast is not so polite. It'll send a copy to every endpoint in a VLAN, for example, regardless of their interest or even compatibility with said packets. In Ethernet bridging, packets with an unknown destination address must be flooded to all endpoints in a VLAN so that: a) the packet is assured of getting to its destination, and b) all of the bridges in the network will have a chance to observe the packet's MAC source address for learning purposes. Flooding and broadcast have identical behaviors that are triggered by two different events—i.e., the matching of the broadcast destination address versus the failure to match the packet's destination address at all.

Collectively, broadcast, unknown-unicast and multicast packets are referred to as BUM packets. For a forwarding system's data path, there's no need to distinguish between the three. In all three cases, a single packet is received and one or more copies are transmitted. With regard to packet processing, however, broadcast and unknown-unicast packets require very different handling than do multicast packets. It is the handling of multicast packets, specifically, that is the primary focus of this chapter.

Multicast Representations

In packet processing, the arcane multicast addressing definitions make things challenging. Practically speaking, IP multicast is the most interesting case because it represents the largest type of traffic and the largest number of applications. IP packets are always conveyed by some kind of lower-layer tunnel in the form of Ethernet or MPLS, for example. Ethernet multicast (with a non-IP payload) is also, of course, a real thing and has real applications.

IP Multicast Over Ethernet

IP multicast packets are, by convention, conveyed as payloads of multicast Ethernet packets. Each version of IP has its own multicast addressing architecture. For IPv4, a multicast address falls into the range: `224.0.0.0-239.255.255.255` (`0xe000_000-0xefff_ffff`). This means that the most significant four bits indicate that an IPv4 address is multicast if they are set to `0xe` while the remaining 28 bits indicate the multicast destination itself (i.e., the so-called "group address").

For IPv6, RFCs 2464, 3306 and 3956 pertain. In a nutshell, the most significant 16 bits of the 128-bit address identify an IPv6 address as being multicast (as well as the scope of the multicast and some flags) while the next 80 bits contain either unicast prefix (RFC 3306) or rendezvous point (RFC 3956) information. What's pertinent to Ethernet is that the least significant 32 bits are used to identify the multicast destination.

For both IPv4 and IPv6, the Ethernet destination address value is derived from the Ethernet packet's IP payload multicast destination address. For IPv4, that Ethernet destination address value has the form: `01–00–5e–xx–xx–xx` (the most significant bit of `xx–xx–xx` must be set to `0`). The least significant 23 bits of a multicast IPv4 destination address are mapped to the least significant 23 bits of the Ethernet destination address value. This, of course, means that five bits of the IPv4 multicast destination address value that define the packet's multicast group are missing from the Ethernet destination address (remember, IPv4's multicast group address value is 28 bits wide).

To eliminate the ambiguity from the multicast Ethernet address where the Ethernet packet's payload is a multicast IPv4 packet, the Ethernet bridge function can look into the IPv4 header and grab the missing bits (i.e., `destinationAddress[4...8]`) and append those bits to `Mac.destinationAddress`, yielding an unambiguous 53-bit MAC address that can be used as a forwarding database entry's key value.

Things are a bit more sensible for IPv6. Here, the Ethernet destination address value has the form: `33–33–xx–xx–xx–xx`. The least significant 32 bits of the IPv6 multicast destination address (all of the relevant group address bits) are mapped to the lower 32 bits of the Ethernet destination address.

Though multicast Ethernet addresses are called for when the Ethernet packet's payload is a multicast IP packet, it is not essential that this be done. If a multicast IP packet is conveyed as the payload of a unicast Ethernet packet, then the Ethernet packet is addressed to the next IP router to forward the packet while the IP packet is addressed to the appropriate multicast group. The use of unicast Ethernet to encapsulate multicast IP is generally not done.

IP Multicast Over MPLS

IP multicast packets may be conveyed via multicast MPLS LSPs. This means that the label values extracted from the MPLS headers are looked up and interpreted as multicast values if appropriate (i.e., the lookup results must return a list of destinations instead of a single destination).

It is not possible to glance at an MPLS packet and determine whether it is unicast or multicast. The original MPLS RFC (3032) defined two ethertype values for MPLS: `0x8847` for unicast and `0x8848` for multicast. That unnecessary distinction was deprecated by RFC 5332 which stipulates that `0x8847` is for downstream-assigned labels and `0x8848` is for upstream-assigned labels, regardless of their unicast or multicast nature. Multicast MPLS packets may be conveyed either by a unicast Ethernet packet or a multicast Ethernet packet. If unicast is used, the Ethernet packet is addressed to the next MPLS LSR router's MAC address. If multicast is used, the `destinationAddress` value may be set to either `01-00-5e-80-00-00` or `01-00-53-8v-wx-yz`, where `v-wx-yz` are copied from the 20-bit `label` value of either the first (i.e., top) or second MPLS header.

Ethernet Multicast

A purely Ethernet multicast packet is one where the Ethernet packet's payload is opaque (i.e., is not either IP or MPLS and the only forwarding information that is available is in the Ethernet header) or the packet is being forwarded by a forwarding system that is only capable of Ethernet bridging. For these packets, the `Mac.destinationAddress` value is looked up and either flooded within the packet's VLAN (for a lookup miss) or distributed to a subset of the transmit interfaces within the packet's VLAN (for a lookup hit).

Bit Index Explicit Replication (BIER)

At the time of this writing, a new multicast method is wending its way through the standards process in the IETF. Known as Bit Index Explicit Replication (BIER), its purported benefits over the use of traditional multicast addresses are that it avoids having to build multicast distribution trees and minimizes the requirement for intermediate nodes to maintain multicast-related state. This forwarding method falls into the broad category of source-based forwarding. Source-based forwarding was experimented with in the early days of networking, but never really caught on. In recent years, the idea has been revived and is being pursued with vigor on a number of fronts (e.g., MPLS segment routing where a large stack of labels is pushed onto a packet at ingress and each hop interprets and pops the topmost label).

In a nutshell, a BIER header is imposed upon a packet between an underlay and the payload packet, roughly akin to a VXLAN header. However, unlike a VXLAN header, a BIER header provides destination identifying information instead of just a virtual network identifier. The destination identification information in BIER is in the form of a bit string where each bit corresponds to an egress BIER router at the edge of a BIER network domain. As the a BIER packet is forwarded across the BIER network domain, bits in the bit string are cleared such that the remaining bits in the string that are set to 1 are those that correspond to the egress BIER routers to which the current copy of the packet is addressed.

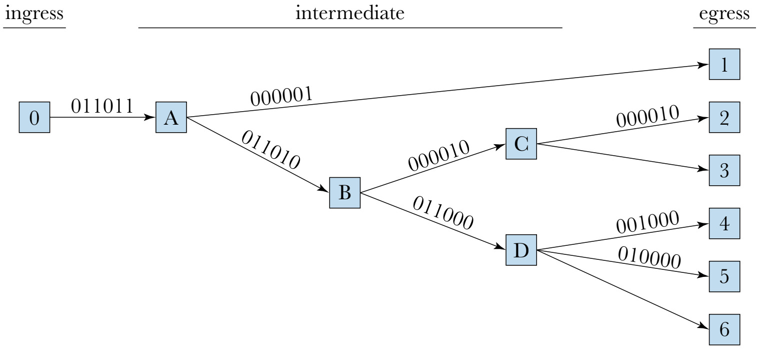


Figure 73 *BIER Replication and Forwarding Method*

An IP packet is receive by a BIER ingress router (0) and is encapsulated in a BIER header as shown in Figure 73. The packet is addressed to multiple BIER egress routers. The BIER bit string is numbered from `1` starting at the rightmost (least significant) bit. So, the bit string `011011` implies that the packet is destined to the BIER egress routers with the IDs `1`, `2`, `4,` and `5`.

By convention (though it really doesn't matter) bit string bits are processed from bit 1. The core BIER router (A) sees that bit 1 is set to `1`, so it examines its forwarding database to identify the next-hop router that corresponds to that bit. As it turns out, the next-hop router in this case is a BIER egress router, so it clears all of the other bits in the bit string to `0` and forwards the packet to router 1.

Router (A) continues to process the bit string. Examining bit 2, it finds that it is also set to 1. The forwarding database indicates that router B is a common next-hop for BIER egress routers 2, 3, 4, 5 and 6. Router A first applies a mask to the received bit string that clears all of the bit string bits that do *not* correspond to BIER egress routers that are reachable via the next-hop for bit 2 (i.e., router B). In this example, that clears bit 1 (bit 1 had previously been cleared, so this action has no effect). This version of the bit string is placed in the BIER header of the packet that is forwarded to router B. Router A then clears all of the bit string bits that *do* correspond to BIER egress routers that are reachable via router B. This clears bits 2 through 6. At this point, all of the bits in the bit string being processed by router A have been cleared, so it is done with its replication work for the packet.

The same process is carried out by routers B, C and D: the bit string is scanned for bits that are set to 1, next-hop data associated with each such bit is fetched and used to mask the bit string two different ways (one to eliminate destinations and one to eliminate non-destinations), and the packet is replicated and forwarded as necessary.

At the BIER egress routers (1–6), the BIER header should have just one bit asserted in the bit string: the bit associated with that BIER egress node itself. The BIER egress node strips off the BIER header and then forwards the multicast payload packet in the normal manner.

The BIER encapsulation header is depicted in Figure 74 and defined in Table 29.

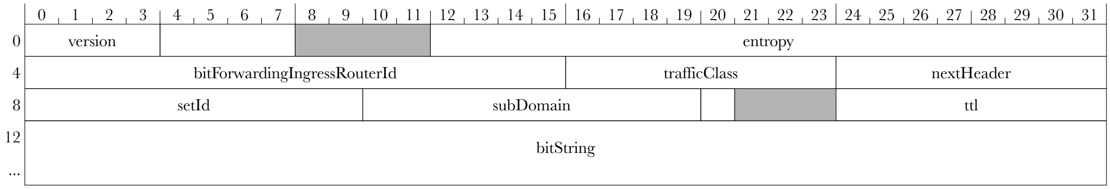


Figure 74 *Transport-Independent BIER Encapsulation Header Structure Diagram*

Table 29 *Transport-Independent BIER Encapsulation Header Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`version` (Ver)	4	0.0	Identifies the BIER header version. Currently, this value must be set to `0`.
`bitStringLength` (BSL)	4	0.4	Indicates the length of `bitString`. This value is enumerated as follows: 1 = 64 bits 2 = 128 bits 3 = 256 bits 4 = 512 bits 5 = 1,024 bits 6 = 2,048 bits 7 = 4,096 bits All other values are reserved.
`entropy` (Entropy)	20	0.12	Load balancing entropy. Typically, this value is set to a hash of the flow-identifying fields of the payload packet headers.
`bitForwardingIngressRouterId` (BFIR-ID)	16	4.0	Identifies the BIER ingress router. This identifying value is set within the context defined by `subDomain`. In other words, the same `bitForwardingIngressRouterId` value may be used in a different sub-domain and refer to a separate BIER ingress router.
`trafficClass` (DS)	8	4.16	Specifies the packet's traffic class. This value is interpreted the in the same manner as, say `Ipv4.trafficClass`, as specified by RFC 2474.
`nextHeader` (Protocol)	8	4.24	Identifies the BIER payload packet type. This value is interpreted in the same manner as, say `Ipv4.nextHeader`, in that the same enumerations are used.

Field Name (std. name)	Width	Offset (B.b)	Definition
`setId` (SI)	10	8.0	Provides an offset to `bitString`. The number of potential multicast destinations for a packet may exceed the number of bits available in `bitString`. When this is the case, the BIER ingress router sends one or more copies of the original packet into the BIER network, each with a different `setId` value. The actual index number of a bit in `bitString` is the raw bit number of the bit in the bit string offset by the product of `setId` and the expanded form of `bitStringLength`. This value is only interpreted as described above when `opaqueNumber` is set to `0`.
`subDomain` (Sub-domain)	10	8.10	The BIER forwarding sub-domain. This value acts as a forwarding context in which to interpret `bitString`. This value is only interpreted as described above when `opaqueNumber` is set to `0`.
`opaqueNumber` (O-flag)	1	8.20	Indicates that `setId` and `subDomain` should be interpreted as a single 20-bit opaque value. If this bit is set to `1`, then the 20-bit concatenation of `setId` and `subDomain` is used as an index or a key for fetching the `setId`, `subDomain` and `bitStringLength`. This use of a 20-bit proxy for these three values is an adaptation of the normal behavior for the MPLS encapsulation form of the BIER header.
`ttl` (TTL)	8	8.24	A time-to-live value. This field operates in exactly the expected manner (see, for example, `Ipv4.ttl`).
`bitString` (BitString)	64-4K	12.0	Variable-length bit string. This bit string is used in conjunction with `setId` (to establish a base offset) and `subDomain` (a forwarding context) to identify the BIER egress nodes to which copies of the packet must be sent.

When MPLS is used as the transport underlay for BIER, a separate, MPLS-specific BIER header format is used. This header format is depicted in Figure 75 and defined in Table 30.

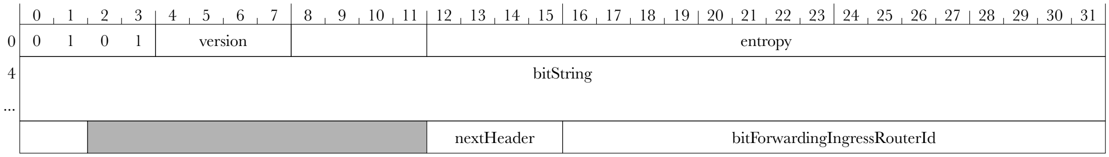


Figure 75 *BIER Over MPLS Encapsulation Header Structure Diagram*

Table 30 *BIER Over MPLS Encapsulation Header Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
5	4	0.0	Disambiguates the BIER header. These bits are intended to prevent confusion between a control word, associated channel header, IPv4, IPv6 and BIER when blindly parsing beyond the MPLS bottom of stack.
`version` (Ver)	4	0.4	Identifies the BIER header version. Currently, this value must be set to `0`.
`bitStringLength` (BSL)	4	0.8	Indicates the length of `bitString`. This value is enumerated as follows: 1 = 64 bits 2 = 128 bits 3 = 256 bits 4 = 512 bits 5 = 1,024 bits 6 = 2,048 bits 7 = 4,096 bits All other values are reserved.
`entropy` (Entropy)	20	0.12	Load balancing entropy. Typically, this value is set to a hash of the flow-identifying fields of the payload packet headers.
`bitString` (BitString)	64-4K	4.0	Variable-length bit string. This bit string is used in conjunction with `setId` (to establish a base offset) and `subDomain` (a forwarding context) to identify the BIER egress nodes to which copies of the packet must be sent.
oam	2	_.0	Performance measurement bits. These bits are used for passive performance measurement marking.

Field Name (std. name)	Width	Offset (B.b)	Definition
`nextHeader`	4	_.12	Identifies the BIER payload packet type. This value is enumerated as follows: 1 = MPLS with downstream-assigned label at the top of stack 2 = MPLS with upstream-assigned label at the top of stack 3 = Ethernet 4 = IPv4 5 = BIER OAM 6 = IPv6 All other values are reserved.
`bitForwardingIngressRouterId` (BFIR-ID)	16	_.16	Identifies the BIER ingress router. This identifying value is set within the context defined by `subDomain`. In other words, the same `bitForwardingIngressRouterId` value may be used in a different sub-domain and refer to a separate BIER ingress router. This value provides context for interpreting `bitString` bits.

In comparing the IP overlay and MPLS overlay versions of the BIER header, you'll notice that the MPLS version is missing quite a few fields. In particular, the MPLS version is missing `trafficClass`, `setId`, `subDomain` and `ttl`. All of these missing fields are expected to be encoded in the MPLS header that precedes the BIER header. This is the so-called "BIER-MPLS" header.

`Bier.trafficClass` and `Bier.ttl` use the native traffic class and time-to-live fields of the MPLS header. The `Bier.setId` and `Bier.subDomain` values are encoded into locally-relevant, 20-bit `Mpls.label` values and are distributed to the intermediate BIER routers that will encounter those label values in received packets. These intermediate BIER routers look up the BIER-MPLS header's label value in order to fetch the necessary `setId` and `subDomain` values. Once that is done, the BIER header is processed normally.

Multi-Layer Multicast

In the previous section, a couple of examples of multicast addressing at multiple layers of encapsulation—specifically, multicast IP and MPLS packets as payloads of multicast Ethernet packets— were presented.

A reasonable question to ask at this juncture is: If the IP or MPLS addressing directs the packets to their destination, why does the Ethernet header's destination address also point to the same set of destinations?

The reason that this is done is to support networks that are a combination of Ethernet bridges and IP or MPLS routers. With unicast packets, the Ethernet header is addressed to the next router along the path toward the packet's destination. Any Ethernet bridge (i.e., a forwarding system that is not capable of IP or MPLS routing) can successfully forward such a packet toward its single destination by interpreting the Ethernet header. However, for multicast, it may be the case that multiple IP or MPLS routers serve as the next hop for a multicast packet's replication tree within the network. If any bridges are interposed between a particular router and the set of next-hop routers, then a unicast Ethernet packet will be insufficient to ensure that the packet is replicated and forwarded to its several next-hop routers.

Things get really interesting when you consider the behavior of forwarding systems that support both Ethernet bridging and IP or MPLS routing.

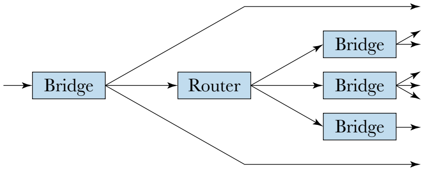


Figure 76 *Combined Multicast Bridging and Routing*

Figure 76 shows several forwarding entities—an ingress bridge, a router and three egress bridges—of a single forwarding system. Let us presume that a multicast Ethernet packet with a multicast IP payload is received by the ingress bridge. Based on the packet's multicast `Mac.destinationAddress` value[10], three destinations are identified: two bridged transmit interfaces and an IP router instance. The ingress bridge duplicates and forwards the packet unmodified (except for possible VLAN translations) to the appropriate transmit interfaces. For the third copy of the packet, the ingress bridge strips off the Ethernet header, updates the packet's forwarding domain and forwards the IP packet to the router forwarding entity. The router identifies three destinations based on the IP destination address value and forwards the packet to three separate bridging forwarding entities encapsulated with new Ethernet headers and with time-to-live values decremented and header checksums updated (if appropriate). The new Ethernet headers imposed by the IP router have `Mac.sourceAddress` values that reflect the current IP router and `Mac.destinationAddress` values that are derived from the IP packet's destination address value. Finally, the three egress bridges interpret the new `Mac.destinationAddress` value in order to identify the one or more transmit interfaces to which the packet must be forwarded.

[10] For IPv4, the additional five bits from `Ipv4.destinationAddress` may be appended to `Mac.destinationAddress` in order to eliminate any forwarding ambiguity.

It is important to note that, in the simple example shown above, the originally-received packet is forwarded to several transmit interfaces by multiple means and onto multiple VLANs. It is bridged to some interfaces and routed to others. This multi-layer multicast behavior adds significant complexity to a forwarding system compared to relatively simple unicast behavior.

Source-Specific Multicast

What one ordinarily thinks of as multicast is any-source multicast; meaning, any endpoint may transmit a packet addressed to a particular multicast address and reasonably expect all of the endpoints that are subscribing to that multicast group to receive a copy of the packet. This is not at all unlike normal unicast behavior; any packet, regardless of its source, that is addressed to a particular destination is expected to be delivered to that destination. This permissiveness can be problematic for certain applications. For example, imagine that you and a large fraction of the world's population are watching the finals of a singing competition via an IP multicast video stream. It would be very unfortunate if some merry prankster started sending their own multicast packets to the same multicast destination address just before the announcement of the winner.

Another weakness of the any-source multicast model is that multiple, independent multicast sources have no practical means for preventing the accidental overlap of the assignment of multicast destination addresses to multicast streams.

To address these problems—and others dealing with setup protocols and scaling—source-specific multicast was developed. A multicast destination or group is designated by G. A source of multicast traffic is designated by S. In the any-source multicast model, a multicast listener simply subscribes to a channel defined by (*, G). For source-specific multicast, a multicast listener subscribes to (S, G) where both the source address and destination address must match the specified values before a packet may be forwarded to the listener.

Ranges of IP addresses have been allocated to source-specific multicast. For IPv4, it's `232.0.0.0/8` and for IPv6 its `ff3x::/96`. If a multicast IP router detects an IP destination in one of these ranges, it can invoke the source-specific multicast behavior wherein the IP source address is combined with the IP destination address to form a lookup search argument. However, there is a variant of the any-source multicast method that, in many ways, behaves the same as source-specific multicast; it is known as source-filtered multicast.

With source-filtered multicast, packets are only forwarded if, for a particular destination address, the source address is a member of a specific set of address values, or the source is *not* a member of a specific set of address values. This is essentially a superset of the source-specific multicast behavior. Source filtered multicast uses the normal IP multicast address ranges, and so cannot be distinguished from any-source multicast based on the characteristics of the IP destination address value.

What this implies is:

- The IP destination address must be submitted to a lookup.
- This lookup returns a token and provisional next-hop data.
- The token value is then combined with the IP source address value for a second lookup.
- The result of the second lookup then qualifies the result of the first.
- If the second lookup is a miss, then the provisional next-hop data from the destination lookup is used.
- If the second lookup is a hit, then the next-hop data from the source address lookup is used.

Bear in mind, that the next-hop data may specify that the packet not be forwarded.

IGMP Snooping

The default behavior for an Ethernet bridge is to flood packets with multicast destination addresses to all of the interfaces that belong to the packet's VLAN (i.e., all of the interfaces of a particular Ethernet forwarding entity instance). In other words, the multicast destination address is treated as if it were an unknown unicast destination address. And, if there are no multicast address values in the bridge's forwarding database, this is the behavior that should be expected.

But what if there were multicast destination address values in the forwarding database? Such entries could specify a list of transmit interfaces to which copies of the packet must be forwarded. This has the beneficial effect of only forwarding multicast packets toward endpoints that are interested in receiving particular multicast packets.

If an Ethernet bridge is expected to build the contents of its forwarding database by observing the `Mac.sourceAddress` values in the packets that it forwards, how does it add multicast addresses? Automatic learning clearly won't work since multicast address values are never used as Ethernet source addresses.

An Ethernet bridge can automatically add multicast addresses to its forwarding database by using a technique known as IGMP snooping. In a nutshell, what a bridge is doing is listening in on the Internet group message protocol (IGMP) join and leave messages sent between IP endpoints and routers and using that information to determine which interfaces are associated with the multicast distribution tree for each multicast address. This technique, while largely effective, is a bit of a hack (kind of like how NAT can be a hack). There are a lot of exception conditions and new versions of protocols can break existing implementations. And, since IGMP snooping entails adding behavior to a data link (Layer 2) protocol forwarding entity that makes it operate on packet headers that belong to the network layer (Layer 3), neither standards entity (IEEE and IETF, respectively) have clear jurisdiction. There is, however, an informational (i.e., not normative) RFC (RFC 4541) that surveys the requirements for implementing IGMP snooping on an Ethernet bridge.

From a forwarding hardware perspective, the essential requirement is that IGMP message packets be detected, intercepted and redirected to the control plane (i.e., the host CPU of the forwarding system) for processing. The control plane, in turn, makes the necessary updates to the bridge's forwarding database. Since IGMP is IPv4-specific, RFC 4551 covers multicast listener discovery (MLD) snooping for IPv6.

12 Connections

If you recall, Ethernet is unique among all of the forwarding protocols described so far in that it has both a header and a trailer. Now granted, the trailer is very simple, containing just the 32-bit CRC value that validates the correctness of the bits of the Ethernet packet. But think about the important implications of that trailer: it is possible for packets to become corrupted during transmission. It is, of course, possible for packets to become lost or otherwise problematic in a number of other ways. Packets may be dropped due to congestion, delivered out of order, duplicated or delivered to the wrong destination. What's important here is that Ethernet—and all of the forwarding protocols so-far discussed—lacks any means for correcting these problems. In short, all of these protocols are unreliable at delivering user data. This kind of unreliable, packet-by-packet data delivery is known as "connectionless networking."

The loss of a small number of packets is fine for certain applications. If you're making a FaceTime call, a lost packet may cause some distortion or noise in the audio or video. That's annoying, but survivable. On the other hand, a multi-megabyte document or application with a missing 1,000-byte string of data from its center is pretty much useless. If a network loses one out of a thousand packets (it's a really crappy network) and a file requires thousands of packets to transport it from one place to another across this network, the file may never successfully arrive.

Fortunately, it is possible to provide reliable delivery services over unreliable forwarding protocols. To express it a different way: a connectionless networking technology can serve as the foundation for a connection-oriented network.

Transmission Control Protocol (TCP)

What we now know as TCP/IP was initially described as a single, monolithic, packet-based networking technology that provided reliable data delivery in the face of occasional packet loss. Wisely, the internetworking functions (i.e., Layer 3 packet forwarding) and the transport functions (i.e., Layer 4 reliable connections) were split into two separate but closely associated protocols. TCP is responsible for abstracting away all of the details of ensuring that packets are delivered in order and intact.

In addition to reliably delivering packets across an unreliable network, TCP has three other responsibilities:

- service demultiplexing
- connection setup and tear down
- congestion mitigation

We will delve into all of these. First, though, let's take a look a the TCP header (defined by IETF RFC 793) and shown in Figure 77. For all intents and purposes, TCP headers are always preceded by an IP header (IPv4 or IPv6) using an IP `nextHeader` value of `6`.

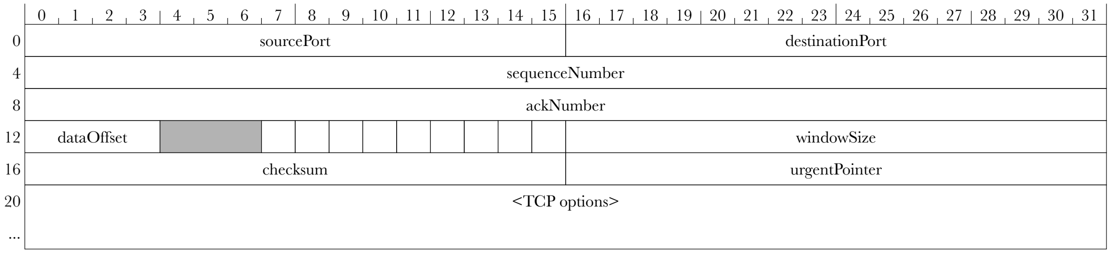


Figure 77 TCP Header Structure Diagram

Table 31 TCP Header Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`sourcePort` (Source Port)	16	0.0	Source and destination TCP process identifiers. These values are used to identify one of, potentially, several services as the origin or termination of a TCP packet. When responding to a received TCP packet, it is expected that the `sourcePort` value from the received TCP packet be used as the transmit TCP packet's `destinationPort` value, and vice versa. As a simple example, these port numbers permit an endpoint to receive both email- and web browser-related packets and ensure that each is delivered to the appropriate process.
`destinationPort` (Destination Port)	16	0.16	
`sequenceNumber` (Sequence Number)	32	4.0	The sequence number of the first byte of the TCP packet's data payload (except when it's not). If `sync` is false (`0`), then `sequenceNumber` behaves as described just above. Otherwise (i.e., `sync == 1`), `sequenceNumber` is the initial sequence number and the first data byte is at `sequenceNumber + 1`. Initial sequence numbers are chosen at random to prevent malicious replay attacks. Separate, independent sequence numbers are maintained for each TCP connection.

Field Name (std. name)	Width	Offset (B.b)	Definition
`ackNumber` (Acknowledgment Number)	32	8.0	Identifies the last successfully received data byte. If `ack` is true (`1`), then this field contains the next `sequenceNumber` value that the sender of the `ackNumber` is expecting to receive. This has the effect of acknowledging all of the data bytes that precede the byte referenced by `ackNumber`. Put simply: “I am expecting to receive bytes starting with byte 100. Hence, bytes 99 and earlier have all been successfully received.”
`dataOffset` (Data Offset)	4	12.0	Indicates the presence of TCP options. This field is analogous to `Ipv4.headerLength`. The nominal value of `5` indicates that the TCP header is 20 bytes long and, hence, has no additional TCP options appended to the basic TCP header. Each value greater than `5` indicates that 4 additional bytes have been appended to the TCP header. Multiplying `dataOffset` by 4 yields the byte distance from the start of the TCP header to the start of its data payload.
`ecnNonceSum` (NS)	1	12.7	A 1-bit sum of random ECN (explicit congestion notification) nonce-encoding code points. This value is used to prevent a TCP data transfer destination endpoint from lying about whether or not packets experienced congestion during their transport from their source. By lying and claiming that no congestion has been experienced, a TCP destination can prevent a source from slowing its offered data rate in the face of congestion. Actual congestion marking by forwarding systems between the source and destination overwrites the source-generated nonce values, making it impossible for a TCP destination whose received packets experienced congestion from computing a correct sum. This function is experimental and is described in detail in IETF RFC 3540.
`congestionWindowReduced` (CWR)	1	12.8	An acknowledgment by a TCP source that its transmitted packets have experienced congestion en route to their destination. This flag bit is set to `1` by a TCP source in response to receiving one or more TCP packets from the destination with the `ecnEcho` flag set to `1`. (See IETF RFC 3168 for details.)

Field Name (std. name)	Width	Offset (B.b)	Definition
ecnEcho (ECE)	1	12.9	Reflects the receipt of an IP packet indicating that congestion has been experienced. This flag bit serves two ECN-related functions. First, if sync is set to 1, it indicates that the TCP peer is ECN-capable. If sync is set to 0, it indicates that an IPv4 or IPv6 packet was received with ecn set to 3 (i.e., congestion experienced). See IETF RFC 3168 for details.
urgentPointerValid (URG)	1	12.10	Indicates that the value in urgentPointer is valid. See urgentPointer, below, for details.
ack (ACK)	1	12.11	Indicates that ackNumber is valid. See ackNumber, above, for details.
push (PSH)	1	12.12	Forces immediate transmission and delivery of TCP data. TCP generally buffers up data until it has accumulated enough bytes for an efficient data transfer (i.e., the overhead of the headers becomes small compared to the payload data). However, for certain applications (e.g., Telnet), it is necessary to send small amounts of data immediately (e.g., individual keystrokes) instead of attempting to buffer up the data. On the receiving end, TCP packets whose push flag is set to 1 have their data payload immediately delivered to the destination process.
reset (RST)	1	12.13	Resets a TCP connection. A TCP Reset is sent either during connection establishment to reject a connection or during data transfer to reject further communication.
sync (SYN)	1	12.14	Starts a TCP connection by synchronizing sequence numbers. A TCP Sync is used to initiate a connection between TCP processes on originating and target endpoints. Only the first packet from each end of a connection may be sent with sync set to 1 for the duration of the connection.
finish (FIN)	1	12.15	Ends a TCP connection. A TCP packet with finish set to 1 is sent to indicate that there are no more data to transfer and that the TCP connection must be torn down.

Field Name (std. name)	Width	Offset (B.b)	Definition
`windowSize` (Window)	16	12.16	The size of the receive data buffer (i.e., window). This value indicates how many additional data bytes the sender of the current TCP packet is able to receive without exceeding its receive buffer capacity.
`checksum` (Checksum)	16	16.0	Protects the TCP header, its data payload and a pseudo-header. This 16-bit, ones-complement checksum is used to protect the TCP header and the data payload of the TCP header. In addition, `checksum` is also computed over a pseudo-header that grabs actual fields from the preceding IPv4 or IPv6 header as well as a synthesized header value. For IPv4, the following fields are concatenated to form a 12-byte pseudo-header: • `sourceAddress` • `destinationAddress` • <one byte set to `0`> • `Ipv4.nextHeader` • TCP (or UDP) length: The combined length of the TCP or UDP header and its data payload, as measured in bytes. For IPv6, the following fields are concatenated to form a 40-byte pseudo-header: • `sourceAddress` • `destinationAddress` • TCP (or UDP) length: The combined length of the TCP or UDP header and its data payload, as measured in bytes. • <three bytes set to `0`> • `nextHeader`
`urgentPointer` (Urgent Pointer)	16	16.16	Identifies the last byte of urgent data. The `urgentPointerValid` and `urgentPointer` values are used to mark bytes within a TCP data stream as being urgent (i.e., must be delivered to their destination process in an expedited manner). All bytes from the current value of `sequenceNumber` up to and including `urgentPointer` must be considered urgent data. The value of `urgentPointer` is held constant across several TCP packets if an urgent TCP data stream requires several packets to convey. The use of `urgentPointer` does not affect the behavior of `sequenceNumber`. RFC 793 (the TCP standard) is updated by RFC 6093 to clarify the semantics of `urgentPointer`.

Field Name (std. name)	Width	Offset (B.b)	Definition
<TCP options>	variable	20.0	Zero or more TCP options. From zero to ten 32-bit words may follow the base 20-byte TCP header as TCP options. All options start with a single-byte type value. Some options' second byte is an option-length value. Some of the more interesting options include (listed by type value): 0: end of options list 1: padding byte 2: maximum segment size 3: window scale 4: selective acknowledgment permitted 5: selective acknowledgment 8: timestamp and timestamp echo

Service Multiplexing

The `Tcp.sourcePort` and `Tcp.destinationPort` values identify the two ends of a reliable connection between two processes running on endpoints addressed by the encapsulating IP header. Any particular endpoint, whether it's considered a server (web, email, file, etc.) or a client (notebook, desktop, mobile, etc.) may—and usually does—have multiple services or processes running simultaneously. Each endpoint system or device typically uses just a single IP address to make itself accessible across an IP network such as the Internet. The TCP port numbers are used to address the several and separate processes running on each endpoint. Thus, it is possible to browse web sites while streaming music, downloading an application, and sending and receiving text messages.

Common services often operate behind so-called "well known" port numbers. For example, a web server accepts HTTP (hypertext transfer protocol) messages via port 80 while FTP (file transfer protocol) uses ports 20 and 21. The initiators of the TCP connections to these ports typically use ephemeral TCP source port numbers in the range of 49,152–65,535 to which the target of the TCP connection is expected to reply. The use of ephemeral source port numbers makes it possible for a single web browser to make multiple, simultaneous connections to a variety of web servers and have the served-up contents demultiplexed, delivered and rendered to the appropriate part of each web page.

Establishing and Ending Connections

A TCP endpoint cannot simply start sending data packets to another TCP endpoint. It must first establish a connection with that other endpoint. Connection establishment is accomplished through a three-way handshake. This process is initiated through the actions taken by an endpoint referred to here as an initiator. For an initiator to succeed at establishing a connection with a target, the target must be ready to accept such a connection. A target makes itself ready to establish a connection by “listening” on a particular TCP port number. This is known as a passive open. The three-way handshake that’s described in detail below is the active open portion of the process.

The steps in the TCP three-way handshake are shown in Figure 78.

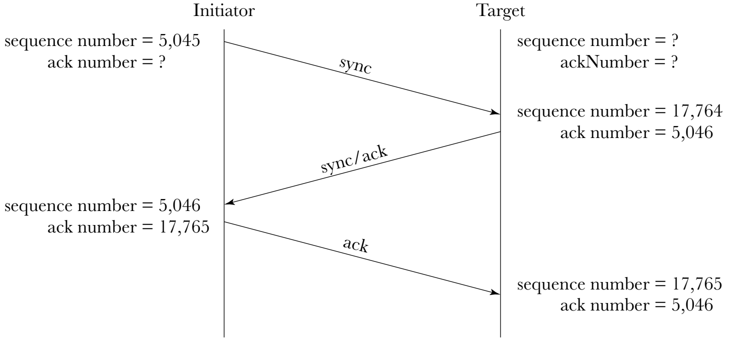


Figure 78 *TCP Connection Establishment Handshake*

The steps illustrated in Figure 78 are thus:

1. Sync

   The initiator that is establishing a connection with a target generates a random sequence number (5,045 for this example) and sends a TCP packet with its `sync` flag set to `1` and its `sequenceNumber` set to the just-defined sequence number.

2. Sync/Ack

   Upon receipt of the TCP Sync packet from the initiator, the target sets its ack number value to the received `sequenceNumber` + 1 (5,046 in this example). The target responds by setting its own sequence number to a random value (17,764 in this example) and sending a TCP packet whose `sync` and `ack` flags are both set to `1` and whose `sequenceNumber` is set to the just-defined sequence number value and whose `ackNumber` is set to the target’s just-set ack number.

3. Ack

   Upon receipt of the TCP Sync/Ack packet, the initiator sets its ack number to the received `sequenceNumber` + 1. The initiator then sends a TCP Ack packet to the target whose `ack` flag is set to `1` and whose `sequenceNumber` and `ackNumber` are set to the initiator’s corresponding internal values.

At the completion of this sequence of packet exchanges, the client (initiator) and server (target) have sequence and ack numbers that are in sync (i.e., one system’s sequence number is equal to the other system's ack number, and vice versa) and a two-way connection has been established.

To terminate the connection, either side may initiate the process and the two halves of the connection are torn down separately. The `finish` flag bit is used to convey the intent to terminate a connection. This four-way handshake is shown in Figure 79.

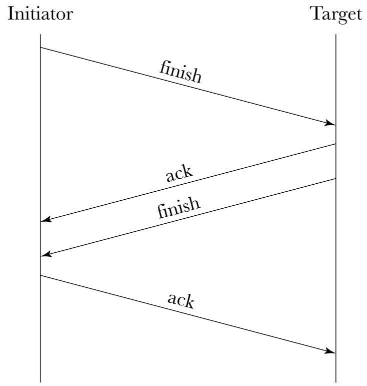


Figure 79 *TCP Connection Termination Handshake (Four-Way)*

The steps in a four-way TCP connection termination handshake are thus.

1. Initiator Finish

   One side of the connection (the initiator) sends a TCP packet to its connection partner whose `finish` bit is set to `1`.

2. Target Ack or Finish/Ack

   The target of the initial termination message responds by sending a TCP packet to the initiator whose `ack` bit is set to `1`. This acknowledges that the initiator-to-target data connection is closed and that the initiator will no longer send TCP data packets to the target. The target responds by sending two TCP packets to the initiator: one whose `ack` bit is set to `1` and one whose `finish` bit is set to `1`. If the target has no more data to send to the initiator, it may combine these two TCP packets into a single TCP packet where both `ack` and `finish` are set to `1`. If the target does have more data to send, the connection may remain "half open" until the data transmissions are complete.

3. Initiator Ack

   Finally the initiator sends a TCP packet whose `ack` bit is set to `1` to complete the connection termination process.

The preceding description of the TCP connection setup/tear down process is, of course, significantly abridged. The process must be tolerant of lost packets and other errors, so the actual state machine has a number of timers and error recovery paths.

Reliable Data Delivery

For a data connection to be reliable, three types of problems must be detected and corrected. These are:

- out-of-order data
- missing data
- duplicated data

TCP's sequence numbers address all three of these problems. Figure 80 illustrates the behavior of TCP sequence numbers.

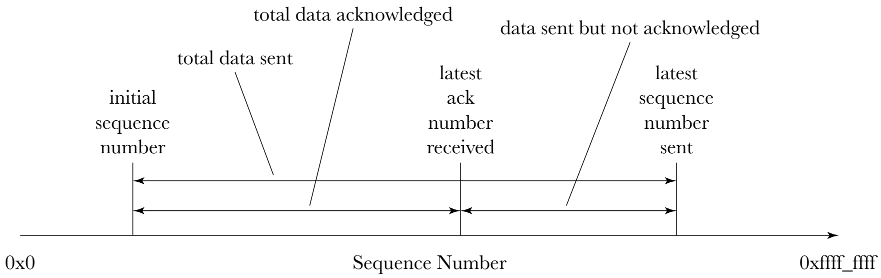


Figure 80 *TCP Sequence Number Behavior*

Briefly: The initial sequence number is chosen randomly. As the TCP sender transmits bytes to the receiver, each TCP packet includes a `sequenceNumber` value that indicates the cumulative sequence number of the last byte of each packet. The TCP receiver responds by transmitting TCP Ack packets whose `ackNumber` value identifies the highest-numbered byte received without any gaps from the first byte of the TCP session. Sequence numbers roll over through zero upon exceeding the maximum value that a 32-bit number can hold.

The sections below explain how the sequence number mechanism described above solves the three fundamental data reliability problems.

Data Reordering

Figure 81 illustrates the process.

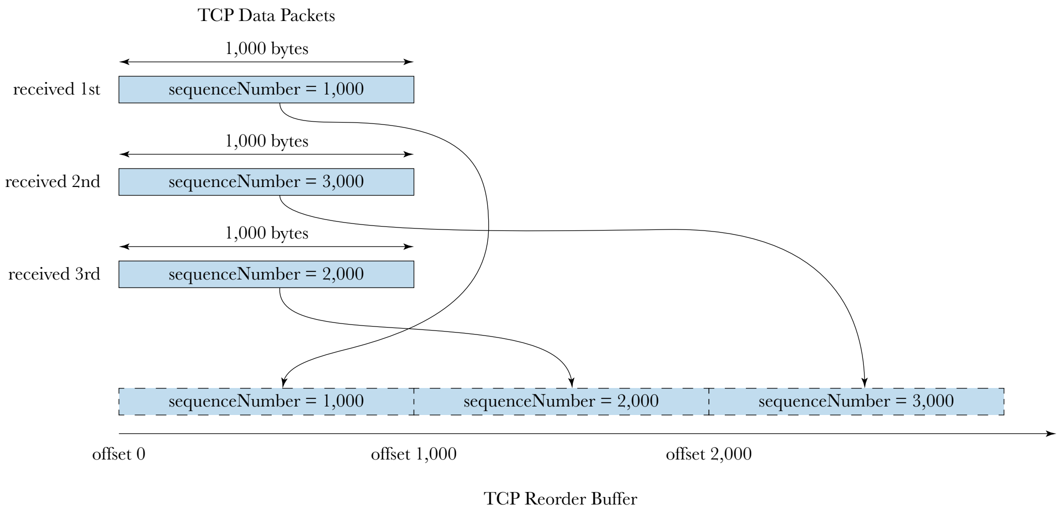


Figure 81 TCP Data Reorder

You can see the presence of `sequenceNumber` in each packet means that a TCP receiver can position each received packet within a TCP reorder buffer where the original sequence of data bytes may be reassembled. Let's consider a simple scenario where the initial sequence number is 0. If the first packet received by the TCP receiver has a `sequenceNumber` value of 1,000 and the packet has a TCP data payload of 1,000 bytes, then this packet neatly fits within the buffer at offset 0. The next packet received by the TCP receiver has a `sequenceNumber` value of 3,000 and carries 1,000 bytes of TCP data. The receiver subtracts the data byte count from the received `sequenceNumber` value to arrive at a sequence number of 2,000 for this packet's first data byte. Thus, the data is written to the buffer starting at offset 2,000, leaving a 1,000-byte gap between the first and second received packets. Finally, a TCP packet with 1,000 data bytes and a `sequenceNumber` of 2,000 is received, meaning the sequence number of the first byte of this packet is 1,000. This third packet's data is written into the 1,000-byte gap between the first two packets' data, completing the in-order data transfer.

Missing Data Retransmission

Using the data reordering example from above as a starting point, let's assume that the TCP packet whose `sequenceNumber` value is 2,000 wasn't simply delivered out of order, but was lost due to any of a variety of the usual things that happen on networks. From the TCP receiver's perspective, the order in which the TCP data packets are received and its responses to the data packets are the same regardless of whether a packet was lost in transmission (and later retransmitted) or simply delivered out of order by the network. Figure 82 takes Figure 81 and augments it with `ackNumber` values.

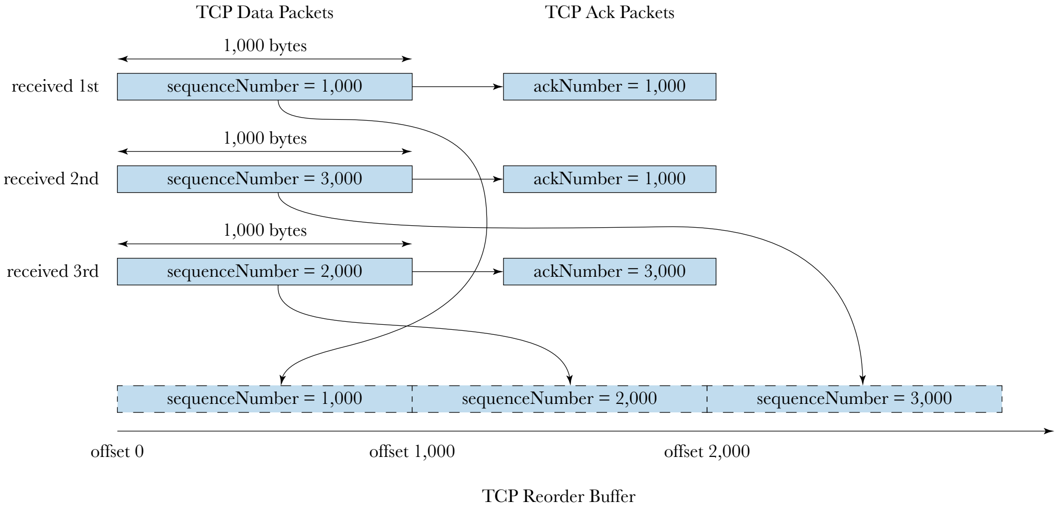


Figure 82 *TCP Missing Data Behavior*

In response to the receipt of the first TCP data packet, the TCP receiver transmits a TCP Ack packet back to the sender with an `ackNumber` value of 1,000, indicating that is has successfully received bytes 0 through 999. The second TCP data packet received by the TCP receiver indicates that a gap exists from byte 1,000 through 1,999. To inform the TCP sender of this, the TCP receiver transmits another TCP Ack packet with the exact same `ackNumber` value of 1,000. This informs the TCP sender that, while the first 1,000 bytes have been successfully received, the next packet (and possibly more) is currently missing. This duplicate acknowledge (the sender has already received an acknowledge for the first 1,000 bytes) motivates the sender to retransmit the first unacknowledged packet (i.e., the packet whose `sequenceNumber` is 2,000).

The TCP sender, in turn, keeps track of the TCP Ack messages that have been received and maintains an awareness of the highest-numbered data byte that's been successfully transmitted. Every unacknowledged byte is subject to retransmission.

Re-transmissions and acknowledges may also get lost. Hence, TCP maintains timers to detect these cases and automatically retransmit either data or Ack packets as necessary to keep the process moving along.

Duplicate Data Discard

Duplicate data may be received by a TCP receiver if a packet that is assumed missing is simply delayed and delivered out of order. The TCP receiver may request another copy of the packet through the use of a TCP Ack message before the delayed data packet is received. If both copies of the data packet are eventually successfully received, then the receiver must deal with redundant data.

If a TCP receiver receives a TCP data packet whose `sequenceNumber` corresponds to a data packet that had previously been received (regardless of whether or not it had been acknowledged), the TCP receiver may either discard the packet or simply overwrite the data in its receive buffer with the newly received data. Since the data is the same, the data in the buffer does not change.

Selective Acknowledge

If a TCP data packet is lost before arriving at a receiver, but that missing packet is followed by several successfully received TCP data packets, the receiver has no effective means for alerting the sender to just retransmit the single missing packet. Selective acknowledgment (defined by RFC 2018 and known colloquially as SACK) is an optional extension to TCP that addresses this specific, common scenario.

If both the TCP sender and TCP receiver agree to participate in selective acknowledgment, then an optional header is appended to TCP Ack packets that list the beginning and ending sequence number of ranges of packets that have been successfully received and queued by the receiver. If two or more consecutive blocks of data have been received, they may be represented by a single start/end entry pair in a selective acknowledgment message. Thus, the sender is provided with a detailed picture of which strings of bytes are missing from the receiver, allowing it to retransmit only those ranges of data bytes that have gone missing.

TCP data packets arriving out of order at a receiver can be briefly interpreted as missing data, prompting two actions by the sender: the retransmission of the missing data and, significantly, a reduction of the rate of data transmission due to the reasonable assumption that packet losses were caused by network congestion. To mitigate against the unnecessary slow-down of the network, the selective acknowledgment function was updated by RFC 2883 to support so-called “duplicate SACK” functionality. Essentially, this extension allows a receiver to claim “never mind about my recent report of a missing packet; it was received after all.” Though this doesn’t necessarily prevent a sender from sending an unnecessary copy of data that’s not actually missing, it does allow the sender to quickly return to normal transmission rates.

Congestion Management

Congestion management in connection-oriented networks has been an area of continuous and active research since the development of TCP in the 1970s and it will probably remain a topic of interest for decades to come. Congestion, its causes, and remedies, is a deep and complex topic. Chapter "13 Quality of Service" on page 185, delves into congestion as it relates to quality of service.

The mechanics of TCP's congestion management scheme is quite simple. Essentially, the TCP receiver transmits TCP Ack messages to the sender that indicate how many bytes the receiver is willing to accept. This limit is known as the "window size" and is conveyed by `Tcp.windowSize`.

If the sender transmits a greater number of bytes than indicated by `windowSize`, then those excess packets are subject to being dropped due to buffer exhaustion on the receiver. Of course, dropped packets must be retransmitted, placing more stress on the network. Hence, senders are obliged to respect the limits advertised by the receiver. The receiver may advertise a smaller window size than it can accommodate in actuality. And, the sender may, of course, transmit fewer bytes than the `windowSize` value dictates and may, further, throttle its transmission to less than a maximum rate. It is the receiver's setting of `windowSize` and the sender's reaction to its values where the subtlety and complexity of TCP congestion control lie.

If a receiver sets its window size parameter too small, then only one packet may be received prior to transmitting a TCP Ack to the sender, establishing a one packet at a time two-way handshake: data/ack, data/ack, etc. If there is a long transmission propagation delay relative to the transmit bit rate and packet length (e.g., a packet that can pass a particular point in the network in 1 µs but where the round-trip transit time is 1 ms), then network utilization is going to be very low since at least 1 ms will pass between each 1 µs packet. Larger window sizes mean more data packets received for each acknowledge transmitted.

The sender determines how many bytes to transmit based on a parameter that it maintains called the congestion window. The congestion window is a limit of the number of unacknowledged bytes that a TCP sender may transmit to a receiver. When a TCP connection is first established, the sender sets its congestion window parameter to a small multiple of the maximum TCP data payload size. If data transmissions result in the timely reception of acknowledges, then the sender increases its congestion window parameter roughly exponentially until reaching the "slow-start threshold." The slow-start threshold doesn't mark the beginning of the slow start portion of window size growth. No, that would be a bit too intuitive. Instead, slow-start is the exponential-growth initial period that is followed by a slower linear growth period. The linear growth period is maintained until packet loss is detected and congestion is presumed.

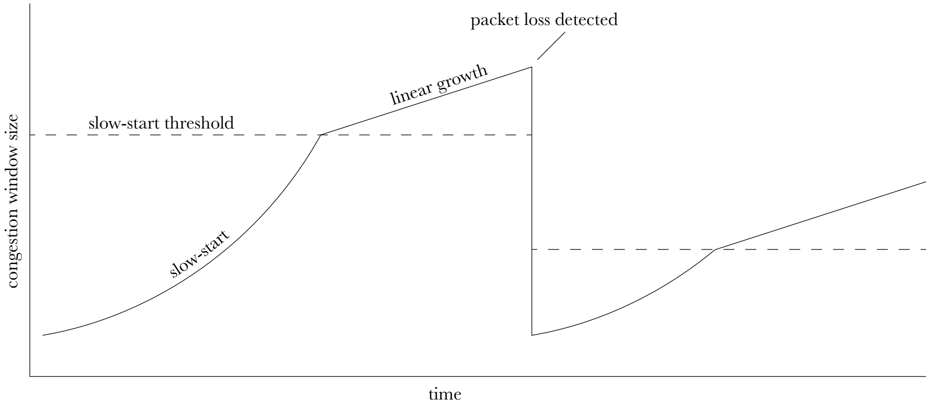


Figure 83 *TCP Congestion Window Size*

When packet loss is detected the slow-start threshold is reduced by about half and the slow-start phase of congestion window growth is reentered.

The behaviors described above take place entirely within the sending and receiving endpoints and not in the forwarding systems in between the two; though, of course, the endpoints are reacting to things that are happening within those forwarding systems: i.e., packet loss due to congestion. Stated another way, the TCP endpoints use packet loss to detect congestion and then adjust their behavior so as to avoid future congestion.

The weaknesses with this approach are two-fold. First, it is possible for all of the TCP endpoints to react the same way at the same time, causing the network to oscillate between periods of congestion and periods of inactivity, even in the face of continuous high demand by the sending endpoints. Second, the endpoints are compelled to make inferences about the congestion state of the network based on what's implied by packet loss.

Random Early Detection (RED)

To address the first weakness, random early detection (RED) and later weighted-RED (WRED) were developed. The concept is quite simple. To avoid the accidentally-synchronized behavior of TCP senders that leads to congestion/idle oscillations, the forwarding systems that forward TCP packets measure the average depths of their queues and use that value to assign a drop probability to each packet. A representative probability curve is shown in Figure 84.

When a queue within a forwarding system is running near empty or at some moderate value, then the drop probability assigned to each packet being enqueued into that queue is zero. As the average queue depth increases and eventually exceeds a particular threshold, the enqueued packets are assigned larger and larger drop probabilities values with continued queue depth increases, leading ultimately to a drop probability of 1—i.e., all packets are dropped.

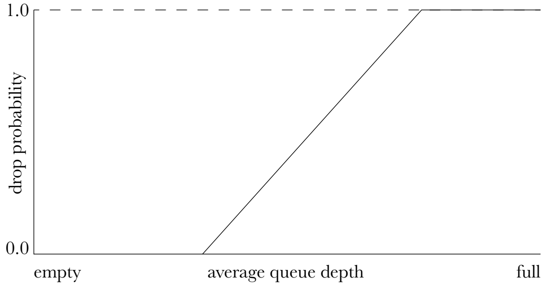


Figure 84 *Random Early Detection Probability Curve*

By randomly dropping a few packets here and there during the early stages of congestion without regard for which connection the packets belong to, a few TCP connections are made to slow down in service of the greater good (the greater good being forestalling the collapse of the network). If network congestion continues to grow, then more and more TCP connections are affected. Over an extended period of congestion—as packets are randomly dropped—some connections are able to recover back to their normal speed while others are compelled to slow down. This has the effect of evenly spreading the impact of the congestion events across all of the active endpoints over a sufficiently long time frame.

For random early detection to be effective, a forwarding system must be forwarding packets from a large number of simultaneous, long-lived TCP connections. Smaller-scale forwarding systems that forward the packets from a small number of TCP endpoints will not see enough connection diversity to be able to gradually modulate demand on the network.

Weighted random early detection (WRED) is a variant of RED where different drop probability curves are applied to packets of different priorities within the same queue. This has the effect of assigning higher drop probabilities to packets of lower priority.

Explicit Congestion Notification (ECN)

To address the second weakness of TCP's congestion management mechanism—implicit congestion notification—a mechanism based on explicit congestion notification (ECN) has been developed.

There are two observations that are relevant to explicit congestion notifications:

1. Explicit notification demands that information dedicated to the task of alerting TCP senders of congestion must be carried in messages from either the receiving endpoint or from forwarding systems along the path from the sender to the receiver.
2. TCP sender and receiver endpoints cannot directly measure the congestion

state of the forwarding systems that lie on the path between them. Thus, the forwarding systems must be involved in the process.

Explicit congestion notification relies on a three-way handshake. The stages are:

1. Detect congestion and mark the affected data packet accordingly to inform the receiver.
2. Feed back to the sender that one or more of its transmitted packets experienced congestion.
3. Inform the receiver that the sender has reduced the size of its congestion window.

Congestion detection is performed by the intermediate forwarding systems that forward packets from a TCP sender to a TCP receiver. The marking of a packet as having experience congestion actually takes place in the IP header, not the TCP header. The IP `ecn` field is used for this purpose.

If the path from a TCP sender to a TCP receiver is capable of explicit congestion notification (i.e., the forwarding systems have RED or RED-like mechanisms that can be used to detect impending congestion and it can update the `ecn` code point) the packets are initially transmitted with `ecn` set to either `ecnCapableTransport0` or `ecnCapableTransport1`. If a forwarding system detects congestion while forwarding a packet, that packet's IP header is updated by changing its `ecn` field to `congestionExperienced`. Once `ecn` is set to `congestionExperienced`, subsequent forwarding systems are not allowed to change it to another value, regardless of whether or not the packet experienced congestion while being forwarded by those subsequent systems.

The receiving TCP endpoint reacts to a `congestionExperienced` notification by transmitting TCP Ack or data packets whose `ecnEcho` bit is set to `1` to the TCP sender. Packets thus marked continue to be transmitted by the receiver until it receives a TCP data or Ack packet from the sender whose `congestionWindowReduced` bit is set to `1`, indicating that the sender's congestion window size has been reduced, effectively reducing the demand for bandwidth that the sender is placing on the network.

In summary, the forwarding systems detect and report congestion to the receiving endpoint via IP and the two endpoints communicate about experiencing and reacting to congestion via TCP.

Implications for Forwarding Systems

TCP connections do not, as a general rule, terminate at forwarding systems[11]. Rather, they terminate at endpoints such as mobile phones, computers, servers and the like. Thus, forwarding systems can operate just fine while being blissfully unaware of TCP. There are, of course, some caveats to that claim.

[11] Management consoles do, indeed, connect to forwarding systems via TCP, but user data forwarding paths do not.

First, the TCP port numbers—`sourcePort` and `destinationPort`—provide rich load balancing information and are often useful when crafting access control lists (ACLs). So, while these applications don't necessarily use TCP information for forwarding purposes, it is still beneficial for a forwarding system to be able to recognize and parse a TCP header in order to extract this flow-identifying information.

Second, for flow-based forwarding, it is generally beneficial for a forwarding system to keep track of the connection state of individual or aggregated TCP connections. Here, too, detecting and parsing TCP headers is required. But, flow-based forwarding must go even further in that it is necessary to detect the start of a flow—`sync`—and the end of a flow—`finish` or `reset`—and keep track that a flow has been established so that TCP data packets that belong to the flow may use a flow-based forwarding method instead of a conventional layer-by-layer forwarding method. When a new connection is detected, a flow-describing, multi-header key is established in a flow forwarding database for that new flow. Upon detection of the termination of a flow, or the termination of the final flow in an aggregation of flows, the flow table entry that supported the just-terminated flow(s) may be removed from the table to make room for new flows.

TCP Variants

There are a couple of interesting variations on TCP that are worth exploring. They are described in the following sections.

Data Center TCP (DCTCP)

The explicit congestion notification mechanism of TCP is binary. Meaning, when congestion is experienced, the sender is informed of the fact that congestion has been experienced, but not the extent to which congestion has been experienced, i.e., no information about the severity of the congestion is fed back to the sender.

DCTCP (currently in draft form at the IETF) remedies this by providing shades of gray. If the congestion is modest, then the sender should reduce its congestion window size a little bit. If the congestion is severe, then the sender should reduce its congestion window size a lot. The behavior of the forwarding systems in a DCTCP network is adjusted in a particular way, but no new mechanisms are required. Specifically, the forwarding systems are configured to mark packets as `congestionExperienced` based on instantaneous measurements of queue depth rather than computing an average queue depth. This greatly improves the responsiveness of the algorithm.

In a classic ECN implementation, the TCP receiver reacts to receiving a `congestionExperienced` indication by setting `ecnEcho` to `1` in every TCP Ack transmitted to the sender until receiving a TCP data or Ack packet with `congestionWindowReduced` set to `1`. In DCTCP, on the other hand, there is a one-to-one correlation between receiving `congestionExperienced` indications and transmitting TCP Ack packets with `ecnEcho` set to `1`.[12] The TCP sender examines a short sequence of TCP Ack packets from the receiver and uses the proportion of `1`s to `0`s to determine the setting of its congestion window size. Thus the sender can react quickly and appropriately, avoiding the usual TCP sawtooth as it hunts for a transmission rate that avoids congestion.

By successfully avoiding congestion, DCTCP enjoys low queue occupancy (i.e., low packet buffer occupation) and low latency, even during periods of significant incast (i.e., lots of source endpoints sending data to a single destination endpoint).

DCTCP is referred to as data center TCP because it is really best (or only) suited for controlled environments of modest physical diameter (i.e., low round-trip time) such as a data center.

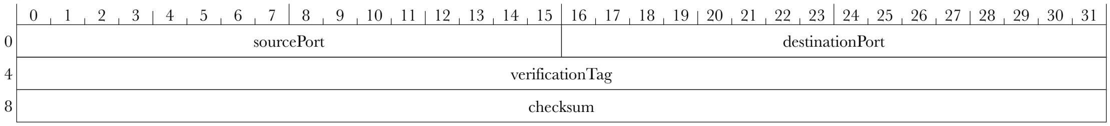
Stream Control Transmission Protocol (SCTP)

SCTP (IETF RFC 4960) was developed to overcome some of TCP's shortcomings. Specifically, TCP:

- insists on in-order delivery even for applications that don't require it;
- operates as a stream of bytes when several applications may need to multiplex discrete messages; and
- is relatively vulnerable to a variety of malicious attacks, including `sync` attacks.

SCTP addresses each of these areas and is well suited for applications such as telephony signaling.

Forwarding systems that encounter SCTP packets (IP `nextheader` value: `132`) can, for the most part, treat such packets as if they were TCP packets. This is so because, like TCP, SCTP packets do not generally terminate at a forwarding system and are, instead, end-to-end, session-oriented, transport packets. A forwarding system should, though, use the `Sctp.sourcePort` and `Sctp.destinationPort` values for computing load-balancing hash values. The format of an SCTP header is depicted in Figure 85.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	sourcePort																destinationPort															
4	verificationTag																															
8	checksum																															

Figure 85 *SCTP Header Structure Diagram*

Notice that the `sourcePort` and `destinationPort` fields are in the same location as their counterparts in TCP and UDP, making parsing and hashing trivially simple.

[12] The ratio may not be 1:1 if TCP Ack coalescing is being used by the TCP receiver.

13 Quality of Service

If packets never arrived at output queues any faster than the queues can transmit those packets, then the queues would operate at essentially zero depth and no packets would ever be delayed or dropped due to queuing resource exhaustion or excessive delays. In such a universe, there would be no need to assign different priorities to various packets nor to worry about queue service policies or congestion avoidance. A single, zero-depth queue at every transmit interface across the entire network would be all that is ever needed.

This fantastical universe is, of course, wildly impractical as it would entail grossly over-provisioning the entire network so as to have bandwidth capacity that exceeds the absolute worst case demand scenarios. In the real world in which we are compelled to exist, budgets and physics conspire to drive us to build networks that can be oversubscribed by a little or by a lot. Once those output queues start to fill up because the dequeue rate falls behind the enqueue rate, that's when quality of service (QoS) policies, protocols and mechanisms come into play.

QoS is fundamentally about managing congestion. Specifically, avoiding it in the first place and, when it can't be avoided, minimizing its negative consequences.

Congestion causes delays. Excessive congestion causes packet loss. Delays and packet loss lead to high network latencies. Variable delays lead to jitter. Packet loss is problematic for network performance and for most administration and management protocols. Both high latencies and jitter are problematic for a variety of real-time applications such as voice over IP (VoIP) telephony.

The Causes of Congestion

Congestion occurs whenever a queue (i.e., a FIFO) is compelled to accept new entries at a rate that is, on average, higher than the rate at which those entries are being read from a queue. During periods of congestion, the depth of the affected queue increases (i.e., entries accumulate). If the conditions that give rise to congestion persist indefinitely, then an infinitely deep queue is required to avoid overrunning the queue. Of course, queues of infinite capacity are more than a little impractical.

The two most common—and easily understood—causes of congestion are displayed in Figure 86.

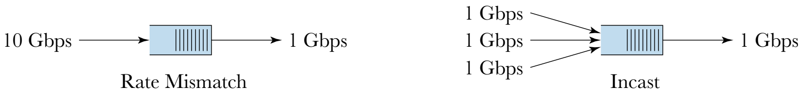


Figure 86 *Queue Congestion Scenarios*

A rate mismatch occurs when a receive interface is able to receive packet bytes at a rate that exceeds the maximum rate at which the corresponding transmit interface may accept the queued packet bytes and transmit them. On the right, the incast problem is shown. Here, multiple receive interfaces—each of which offers a receive data rate that is less than or equal to the corresponding transmit interface—conspire to combine their data rates so as to behave as would a single receive interface of a higher packet data rate. The net effect of these two scenarios is exactly the same: packet bytes are received faster than they can be transmitted.

One common variant of the rate mismatch scenario is the transmit flow control scenario. If a downstream forwarding system to which the current forwarding system is transmitting packets experiences congestion, it may send a flow control indication to the current forwarding system. In its most crude form, this flow control indication completely shuts down the transmit interface of the current forwarding system, dropping its transmit rate to zero. If packets continue to be received and enqueued into the affected queue of the current forwarding system, then that queue's depth is bound to increase—potentially quite rapidly.

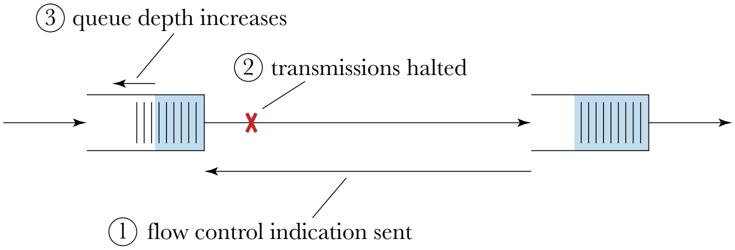


Figure 87 *Transmit Flow Control and Queue Congestion*

When a queue reaches its capacity, it is compelled to implement some kind of policy to deal with the excess traffic. The simplest and most common policy is to tail-drop. Here, packets that attempt to enqueue into an already-full queue are dropped instead of being enqueued—i.e., the packets are dropped at the tail of the queue. One of the hazards of the tail-drop congestion mitigation policy is that it may unfairly bias against long packets. Consider a scenario where a queue has room for just 100 more bytes. If a 64-byte packet attempts an enqueue into the queue, it'll succeed. If, instead, a 1,000-byte packet makes an attempt, it'll be dropped. If the queue advances slowly relative to the arrival rate of new packets, then very few (if any) long packets will be successfully enqueued. To defend against this bias, the queue must declare itself to be full when the number of bytes remaining in the queue drops below the longest possible packet length. This means, of course, that 64-byte packets will be dropped even though there may be 500 bytes of space left in the queue. But, drop them, it must.

The antipode of tail-drop is head-drop. Here, packets that appear at the head of a queue ready to be dequeued are dropped instead, making room for new packet enqueues at the queue's tail. The benefit of this approach is that the packets at the head of a queue are the oldest and, by getting rid of the oldest packets, the average queuing delay may be minimized. This, of course, comes at the expense of dropping packets. But, packets were going to be dropped anyway, so it might as well be the oldest of them.

The significant challenge of head-drop makes itself obvious when the time comes to actually implement the queuing hardware. Enqueuing and dequeuing packets are operations that both require a certain amount of work. And work is not free. It takes time. It consumes power. When managing a FIFO, the amount of work required to dequeue a packet for transmission is roughly the same as that required to dequeue a packet for a head-drop. In order to be able to actually make progress in making the queue shorter (i.e., reduce the number of entries in the face of new enqueues), the dequeue function must be appreciably faster than the enqueue function (remember, in the head-drop scenario, all packets are being enqueued). Tail-drop, by comparison works by simply doing no work—the received packet is dropped before the enqueue mechanism is engaged.

Congestion Detection

In order to react to congestion, it must be detected. Packet loss is, of course, one way to do so. But, it is better to detect congestion before it worsens to the point where packets must be dropped. The simplest way to do that is to compare the queue depth to a threshold that is set some distance before the maximum possible depth for the queue.

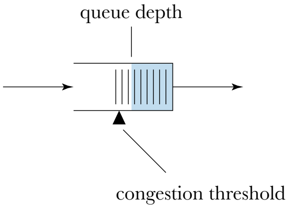


Figure 88 *Queue Congestion Thresholds*

Quite simply, whenever the queue depth exceeds the congestion threshold, the queue is considered congested. One potential problem with this very simple approach is that if some kind of action is taken (e.g., a message is sent) every time the queue depth crosses that threshold in either direction, it's possible for a lot of such actions to be executed if small packets are being rapidly enqueued and dequeued at the same rate while the queue depth hovers near the congestion threshold. To mitigate this, a pair of thresholds—a higher one for entering the congestion state and a lower one for exiting the congestion state—are used to introduce some hysteresis. The larger the hysteresis, the lower the maximum message rate will be.

Another approach to detecting congestion hinges on the observation that one of the most significant and damaging impacts of increasing congestion—aside from packet loss—is increasing delays in delivering data, i.e., the deeper the queue, the longer that it'll take for a just-enqueued packet to be transmitted by a particular forwarding system. So, rather than simply measuring the number of packet bytes that are enqueued, measure the amount of time that just-dequeued packets spent in the queue. To do this, an enqueue timestamp is included with the packet's metadata when it is enqueued. Upon dequeue, the timestamp value is compared to the current time of day. The difference represents the amount of time that particular packet resided within the queue. The delay value can either be reported directly or it may be compared to various thresholds in order to report two or more congestion states. When using queuing delays as a proxy for congestion, it is still important to keep track of the queue depth and to take some kind of drastic action if the maximum depth threshold is being approached or has been crossed because the delay-based congestion mitigation techniques can't necessarily protect the queue from overflows and packet loss.

For an example of reporting congestion state to the sources of congestion, see the preceeding discussion on DCTCP on page 183.

Congestion Avoidance Through Rate Controls

If a sender of packet data is aware that its packets are encountering congestion, and it is inclined to be well behaved—i.e., be cognizant of the very real possibility that it, along with other sources of packet data, are all contributing to the problem and it is willing to make a sacrifice in order to relieve the problem for its own and all other sources—it can modulate its transmission rates so as to relieve the pressure on the network and allow the queues to drain to the point where delays and congestion are no longer a concern.

The techniques for rate controls run the gamut from crude and brutal to subtle and refined. On the crude and brutal end of the spectrum is link-level flow control as codified by IEEE standard 802.3x.

Link-Level Flow Control (IEEE 802.3x)

In a nutshell, 802.3x “pause” packets are sent by an endpoint or forwarding system that is experiencing congestion to the link partner of the receive interface to which it attributes its congestion. The link partner is expected to cease all transmissions onto the interface via which it received the pause packet. The pause packet specifies a timeout period, after which the link partner is free to resume transmissions, unless the link partner receives a subsequent pause packet (setting the timeout timer to a new value) prior to the expiration of the current timeout period.

802.3x flow control messages are conveyed by Ethernet packets using the reserved multicast `destinationAddress` value `01-80-c2-00-00-01` and `ethertype` of `0x8808` (MAC control). The format is depicted in Figure 89 and the field definitions listed in Table 32.

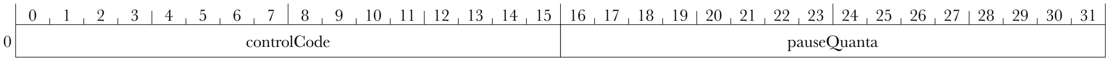


Figure 89 Link Level Flow Control (Pause) Header Structure Diagram

Table 32 Link Level Flow Control (Pause) Header Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`controlCode`	16	0.0	Defines the MAC control code. For link-level flow control, the code is `0x0001`.
`pauseQuanta`	16	0.16	The pause duration as measured in Ethernet slot times. An Ethernet slot time is 512 bit-times. So, the pause quanta is link bit-rate dependent. Up to 65,535 slot times may be specified for the pause duration. At, say, 10 Gbps, that’s 6.55 µs. Each pause message resets an Ethernet MAC’s pause timer to the value in the pause message. A message-sending Ethernet forwarding entity may transmit periodic pause messages in order to prevent its link partner from transmitting indefinitely. A pause time of zero takes a MAC out of the pause state and enables the immediate resumption of packet transmissions.

One of the serious weaknesses of 802.3x link-level flow control is that it shuts down all transmission from the affected sender (except MAC control packets). In all but the simplest of networks, this is overkill. It is overkill because Ethernet links typically carry a variety of flows of packets that are coming from and going to a variety of places. Some of those flows may be contributing to congestion while others are not. Shutting down all of the flows that happen to share the affected link casts a wide net and causes a lot of collateral damage. This phenomenon is discussed in more detail in *"Head of Line Blocking" on page 201*.

Random Early Detection (RED)

This method of discarding randomly-selected packets to modulate the rate at which a TCP sender transmits packets is described in *"Random Early Detection (RED)" on page 180*.

Resource Reservation Protocol (RSVP)

In the mid 1990s, the IETF embarked on a project to define a protocol to allow IP endpoints to request connection-like reservations for certain bandwidth and delay guarantees. The first version of RSVP is described by RFC 2205.

RSVP represents an instance of an "integrated services" model wherein a path through a network is identified and bandwidth guarantees (and other parameters) are granted to that path. Integrated services are in contrast to "differentiated services" wherein prioritization is used to grant superior service to important traffic (see *"Congestion Management Through Prioritization" on page 200*).

Following the multicast model where receivers subscribe to sources of data, receivers transmit RSVP messages toward the data source and the intermediate nodes along the path are expected to process the requests and set aside the necessary resources. Flows from a sender are identified by the three-tuple consisting of IP `destinationAddress`, IP `nextHeader` and (optionally) TCP or UDP `destinationPort`. The parameters of the resource reservation request also specify the quality of service guarantees that are required (bandwidth, delay, etc.).

When a receiver sends a request message toward the sender with the expectation that all of the intermediate nodes successfully received, processed and accepted the request, it's taking it on faith that the reservation was successfully set up. This, of course, won't do. So, the sender sends a reciprocal message where all of the intermediate nodes indicate their status of the reservation request. This gives the receiver a reliable indication that the request has been accepted.

Both unicast and multicast data streams are supported by RSVP. And, as can be anticipated with multicast, the list of subscribers to a data stream may change over time. To accommodate this, periodic refresh messages are sent in order to keep the state current.

It should be fairly apparent at this point that RSVP has scaling difficulties. The number of flows that a single IP forwarding entity may have to manage can easily swell into the millions for a large-scale network. It is because of these difficulties that RSVP has not seen widespread adoption in IP networks. However, RSVP has been embraced by MPLS as RSVP-TE.

RSVP with Traffic Engineering (RSVP-TE)

MPLS is well suited for applications where performance guarantees are a requirement. For example: telephony. Though the data associated with any particular phone call is modest (just tens of kilobits per second), it absolutely must be delivered with a minimum of delay and loss. Delays lead to high latency, which can make voice communication very difficult. (Have you ever been on a phone call that's afflicted with high delays and you and the person you're speaking with are constantly stepping all over each other's sentences? The long delay is unnatural to our ears and we reasonably assume that a reply isn't coming.) Data loss, on the other hand, causes clicks and pops in the audio, degrading the quality of the sound and degrading the intelligibility of the speech.

MPLS and its label distribution protocol (LDP) provide a means for configuring all of the MPLS label switching routers (LSRs) in a network so that label swapping provides pre-defined pathways through the network for all of the required connections. What LDP lacks, however, is a means for making QoS guarantees. This is where RSVP-TE comes in.

Unlike RSVP for IP—which relies on routing protocols for the establishment of network forwarding behavior—RSVP-TE encompasses both a means for defining the paths through a network and (optionally) establishing or provisioning QoS guarantees for each path. So, RSVP-TE is used in lieu of LDP on those networks where QoS guarantees are important.

RSVP-TE uses a bidirectional path allocation scheme. An MPLS label switch path (LSP) is defined as an ingress MPLS label edge router (LER) connected to an egress label edge router through a series of MPLS label switch routers (LSRs). The ingress label edge router serves as the MPLS tunnel entrance while the egress label edge router serves as its exit. The establishment of an RSVP-TE label switch path starts at the tunnel entrance. It does this by sending an RSVP-TE path message toward the egress label edge router. The message includes parameters that describe the requirements of the path being established. The MPLS routers along the path keep a record of these requirements, but do not act on them otherwise at this time.

When the RSVP-TE path message reaches the egress label edge router, it starts the process of actually reserving the resources required by the requested path. It does this by sending an RSVP-TE reservation message back toward the ingress label edge router. Along the way each label switch router verifies that sufficient resources are available and then commits those resources to the new path. Each router also generates an `Mpls.label` value for use by the upstream neighbor label switch router to use to forward packets to the current node along the path being created.

If the QoS parameters requested by the reservation message cannot be granted, an alternate path is attempted. If a path cannot be found that meets the requirements of the reservation message, then the reservation is declined.

The path is kept alive through the transmission of periodic refresh messages (nominally, every 30 seconds per path). If a path fails to receive any refresh messages for three minutes, the reservation is torn down and must be reestablished.

Ordinarily, the path reservation messages and, consequently, the to-be-established path follows the shortest path as defined by the Interior System to Interior System protocol (IS-IS, see page 332) running on the MPLS network. However, the shortest path can be overridden through the use of explicit route objects (EROs). Quite simply, the explicit route objects conveyed by a reservation message list the routers through which the path must pass. The list can either be loose or strict. A loose list specifies a subset of required hops (the remainder being chosen automatically along a computed shortest path). A strict list exhaustively specifies all of the hops that the path must follow.

The details of RSVP and RSVP-TE have no significant impact on the design of forwarding system hardware aside from setting the stage whereby promises made by the sources of data and the forwarding systems must be enforced.

Enforcing Rate Controls

When a client and a service provider enter into a contract to provide networking services, part of that contract inevitably specifies the minimum bandwidth, maximum latency and maximum packet loss rate that the service provider promises to the client. Further, the client is often bound to limit the rate at which packet bytes are transmitted toward the service provider. Taken together, these parameters make up what is commonly known as a service level agreement (SLA). Specialized hardware functions within forwarding systems are used to monitor and enforce these service level agreements. Broadly speaking, these functions fall into four categories. They are *counters*, *meters*, *policers*, and *shapers*. Each is described in detail in the following sections.

Counters

A counter is, as its name plainly infers, a mechanism that counts events. In a forwarding system, the events that are counted are packets and bytes that pass certain defined points in the forwarding system. These points may be physical receive or transmit interfaces, logical ports, forwarding domains, virtual forwarding entities, etc.

In their ideal form, a counter starts off at zero at the beginning of time and is able to count an infinite number of events without loss for the rest of time. More practically, the width of these counters must be bounded to some reasonable limit. Consider a 1 Tbps interface (10^{12} bits per second). If a counter is assigned the task of counting every byte that passes that point, a 64-bit counter can do so without rolling over for just over four and a half years.

Absolute totals since the beginning of time are not necessarily of any practical value. It is events per unit of time that really matter since service level agreements and billing cycles are often expressed in periods of days, weeks, months or years. For example: How many gigabytes were received from a customer last month? So, even if the period is large, we care more about event rates rather than absolute counts.

There are two fundamental methods for managing counters. The first and, perhaps, most obvious approach is to simultaneously read and reset a counter so that it resumes counting from zero immediately upon being read. This method certainly works, but in practice it can be a bit tricky to read and reset a counter without losing any counting events. The less obvious—but simpler in practice—approach is to allow the counter to run forever without being reset and for the counter to roll over through zero after reaching its maximum value. The process that is monitoring the counter simply reads the current value whenever necessary. By retaining the previously-read counter value and subtracting it from the just-read counter value, the delta from the previous sample to the current sample can be determined. That delta can then added to an accumulator that is of any desired width. Of course, care must be taken when the counter rolls over through zero between two samples. The solution is to ensure that the counter monitoring process samples the counter at least a bit faster than the counter's minimum rollover period. By doing so, a rollover can be detected by noticing that the difference between two consecutive samples is negative rather than positive.

Consider a simple case where a 4-bit counter rolls over from 15 to zero upon reaching its terminal count. The counter monitoring process should be sampling the counter frequently enough that it should never see a count delta greater than 15. If the previous counter sample value is, say, 13 and the current counter sample value is, say, 3, then the usual subtraction of the previous value from the current value yields 3 - 13 = -10. To arrive at the correct value, add 16 (i.e., $2^{\text{counterWidth}}$). This yields the correct answer: 6.

Meters

A meter is, essentially, a counter with a leak.

Consider a bucket with a calibrated hole in its bottom. If water is poured into the bucket faster than the hole allows water to escape, then the water level in the bucket will increase, indicating that the designed rate is being exceeded. Conversely, slow the pour rate below the drain rate and the water level will decrease. By monitoring changes in the water level, it is possible on a moment-by-moment basis, to determine if the ingress rate exceeds some target value. And, when comparing the characteristics of a flow of packet data to a service level agreement, determining whether or not the flow exceeds a defined rate is exactly what's needed.

Rather than water, we pour in and drain out tokens in order to determine if a flow is complying with a service level agreement. A service level agreement can allow a customer to maintain some maximum long term average data rate (also known as an information rate). However, it is recognized that networking can be bursty, so bursts are accommodated by also allowing a committed burst size (i.e., length) and a peak or excess burst size. If any of those thresholds are exceeded by a flow, the offending packets are "marked" accordingly. The marking of packets is discussed further along. For the time being, we'll focus on rate metering.

IETF RFC 2697 describes, among other things, a single-rate metering method using token buckets. The metering method is quite simple:

1. Two token buckets—committed and excess burst length—are initially filled all the way up with a number of tokens that represent the maximum committed burst length and excess burst length, respectively. Thereafter, the token buckets are refilled (i.e., a counter value is incremented) at a rate that represents the committed information rate whenever the token buckets are below their initially-configured maximum values. However, only one of the two buckets may be refilled with each increment; the committed burst size bucket takes precedence over the excess burst size bucket. In other words, the tokens are replenished at the committed information rate (up to the bucket maximums), but the committed burst bucket is refilled first.
2. With each received packet belonging to the flow that's being metered, the length of the packet (i.e., the number of bytes) is compared to the committed burst token bucket. If the number of tokens in that bucket equals or exceeds the packet length, then the packet is designated "green" and the packet's length is deducted from the committed burst token bucket.
3. If the packet is longer than the number of tokens available in the committed burst bucket, then the excess burst bucket is checked. If the packet's length may be deducted from the excess burst token bucket without it going negative, then the packet is designated as "yellow" and the deduction from the excess burst bucket is performed.
4. If both the committed and excess checks described above fail, then the packet is designated as "red."

The green, yellow, and red designations in Figure 90 correspond to "within the committed sustained rate or committed burst rate," "above the committed burst size limit, but within the excess burst size limit," and "above the excess burst size limit," respectively.

The packet color marking algorithm described above is what's known as "color-blind marking," meaning, the current color of a packet—as determined by an upstream forwarding system—is not taken into account.

In color-aware packet marking, a packet's color may never be promoted. For example, a packet that is received with a yellow color may not be tested by the committed-burst token bucket and then marked as being green. Only a green packet may be marked as green. Green packets may be marked green, yellow, or red. Yellow packets may be marked yellow or red. Red packets must remain red. This is accomplished by steering packets to appropriate color marking steps according to their received color. Green packets execute steps 1–4 as described above. Yellow packets skip step 2. Red packets skip steps 2 and 3. Very simple, really.

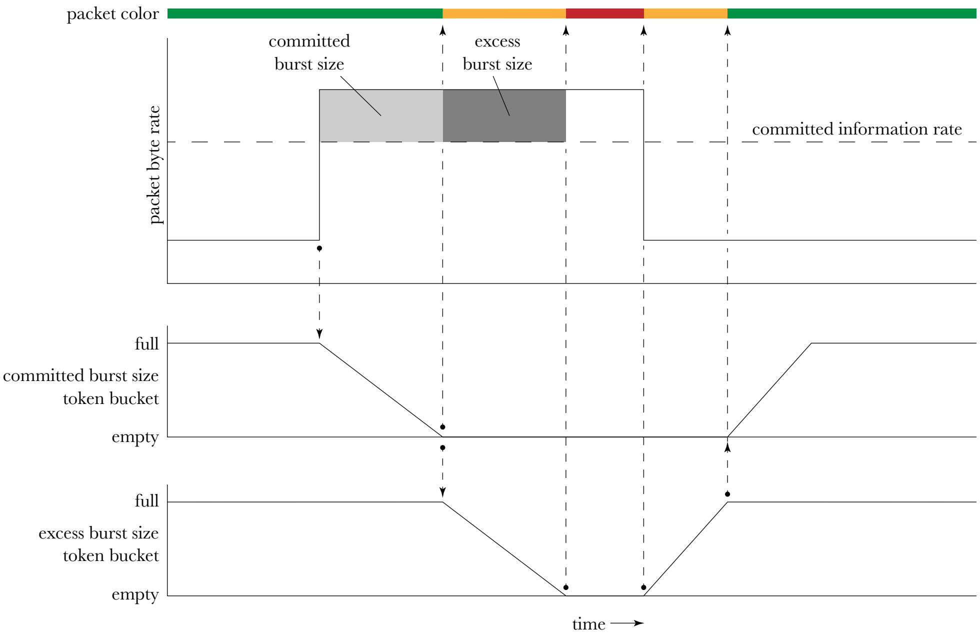


Figure 90 *Single-Rate Packet Color Marking*

IETF RFC 2698 defines a two-rate metering method that is distinct from the single-rate method just described and illustrated in Figure 90. The two-rate meter introduces a peak information rate to complement the single-rate meter's committed information rate:

1. The two token buckets—committed and peak—are initially set to values that represent the configured committed burst size and peak burst size. These values represent the maximum levels that the two token buckets are allowed to achieve. Tokens are added to the two buckets at rates that represent their respective information rates—committed and peak—clipping the sums to the maximum allowed values for the buckets.
2. A received packet's length is compared to the peak burst bucket and, if the packet's length exceeds the byte length implied by that bucket's token count, the packet is marked "red."

3. If a received packet's length is less than or equal to the peak burst token count, but greater than the committed burst token count, then the packet is marked "yellow" and the peak token bucket is decremented by the packet's length.
4. If a received packet's length is less than or equal to both token buckets, then the packet is marked "green" and the committed token bucket is decremented by the packet's length.

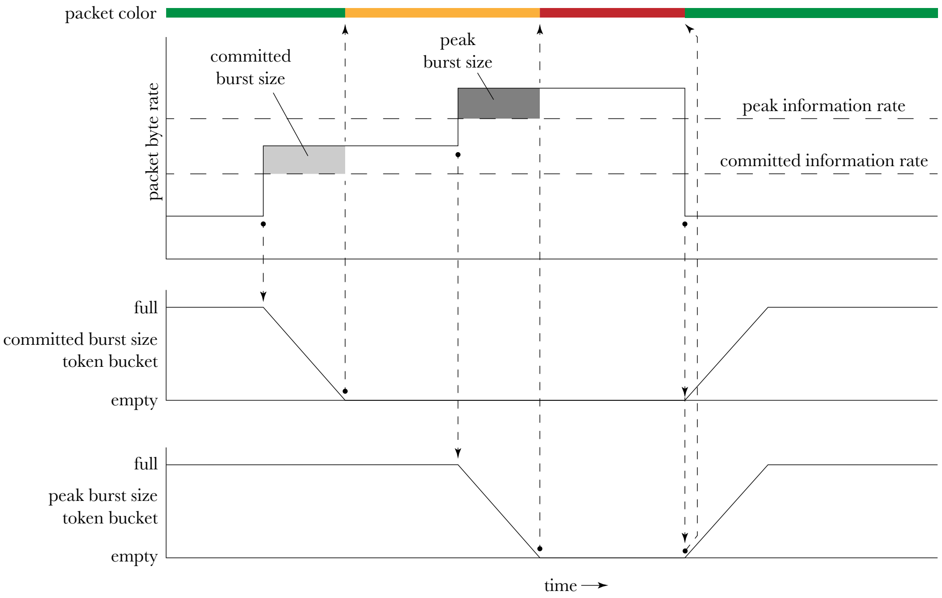


Figure 91 *Two-Rate Packet Color Marking*

The method described for two-rate packet color marking is the color-blind version of the algorithm. The color-aware version differs in that the current color marking of a packet is taken into account when determining its new marking. Specifically, a packet's color may never be promoted. For example, a packet that is received with a yellow color may not be tested by the committed-burst token bucket and then marked as being green. Only a green packet may be marked as green. Green packets may be marked green, yellow, or red. Yellow packets may be marked yellow or red. Red packets must remain red. This is accomplished by steering packets to appropriate color marking steps according to their received color. Green packets execute steps 1–4 as described above. Yellow packets skip step 4. Red packets skip steps 3 and 4.

The difference between a single-rate and a two-rate metering function is subtle, but important. With a single-rate meter, an incoming packet data rate that exceeds the committed information rate will eventually progress from green, through yellow, to red. With a two-rate meter, the incoming packet data rate can exceed the committed information rate forever and will only progress from yellow to red if the peak information rate is exceeded.

The marking of packets with colors has been mentioned a number of times in the preceding paragraphs, but it's not been explained how this is done and why it matters. The mechanics of packet color marking is fairly straightforward. Both IPv4 and IPv6 headers have a field that is dedicated for use in conveying a packet's quality of service attributes. This is the IP `trafficClass` field. This six-bit field is used to encode all aspects of a packet's quality of service attributes, how a particular packet should be handled differently from other packets that have different traffic class values. One of those parameters is the packet's color marking by rate-meters. There are no specific codepoint values or subsets of bits that are explicitly defined to convey a packet's color marking. A set of codepoints are simply defined within a network to mean, for example, low-priority red, or high-priority green, etc.

So, what do these colors represent? They represent the degree to which a flow is complying with the service level agreement that applies to the flow. A green flow is completely within the most conservative limits of the agreement and there is a reasonable expectation that the vast majority of the packets within the flow will successfully arrive at their intended destination. Packets that are marked yellow are exceeding the basic bandwidth commitment and that excess is either short-lived (single-rate meter) or the degree to which it exceeds the committed rate is small (two-rate meter). These packets are subject to being dropped if they encounter any meaningful congestion on their way to their destination. Finally, packets in flows that are marked red are subject to being dropped even if congestion is not encountered.

Packet order is an important consideration when marking packets. Red packets are generally dropped (but not necessarily always so), while yellow packets are generally forwarded (though sometimes dropped) and green packets are almost always forwarded. All of these packets—regardless of their color—may be part of the same flow, meaning they need to be delivered to their destination in per-flow order. All of these packets started off green and some of them may be changed to other colors as they are forwarded. It is important that the color of a packet does not influence the choice of the queue into which it is enqueued. Further, it is important that, when LAG or ECMP operations are performed, the link choice is not influenced by a packet's color. It is generally good practice to map an IP packet's `trafficClass` value to separate priority and color values and to only consider the priority of a packet when making queue and link choices.

Which bytes get counted during metering? An IP packet typically consists of the IP packet itself, the encapsulating Ethernet packet and the Ethernet overhead (preamble, inter-packet gap, CRC). The Ethernet encapsulation and the overhead represent a significant and varying amount of additional byte-times on top of an IP packet. However, when considering IP differentiated services and implications of the IP `trafficClass` field, it is just the bytes that make up the IP packet that matter. That's all of the bytes from the start of the IPv4 or IPv6 header through the last byte of the IP payload (not including any Ethernet pad bytes).

It is important to point out that the RFCs that define the behavior of the two types of meters do not necessarily dictate how these functions are implemented in hardware or how a forwarding system must handle packets that have been marked specific colors. The meters may be implemented in whatever fashion is desired as long as the intended behavior is achieved. The reactions to various color markings are a matter of policy which may have just a local scope.

Finally, metering and marking packets doesn't count for very much if there isn't some means for dropping packets that have been marked yellow or red. That is the job of the policers.

Policers

Policers consider a packet's marking (green, yellow, or red) and the congestion state of the queues through which it may pass and make a drop/don't-drop decision on a packet by packet basis. Policers may use fixed queue-depth threshold, statically or dynamically adjustable thresholds, tables, curves or some other means to determine just how congested a queue is. The representation of queue congestion is then combined with a packet's color to determine either whether or not to drop a packet, or the probability of dropping a packet (à la RED).

The sequence of events matters with regard to meters, markers, and policers. Consider Figure 92.

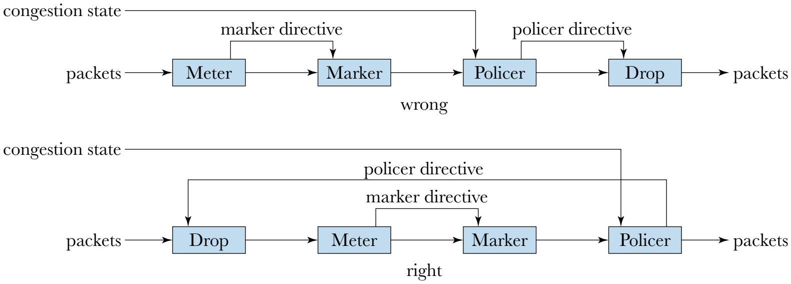


Figure 92 Meter, Marker and Policer Sequence of Events

First, let's take a look at how *not* to arrange these related events. In the top half of Figure 92, the meter, marker, and policer are followed by a drop function that either actually drops the packet or simply marks the packet to be dropped at a later stage. The problem with this arrangement is that if a flow of packets exceeds the upper limit of the relevant service level agreement, and they are consequently all marked red, the policer will dutifully direct the drop function to drop all of those red packets. Unfortunately, this means that the incoming flow of packets goes from exceeding its allowed limits to zero packets actually being forwarded. This is certainly not in keeping with the service level agreement.

The right way to do this is to place the drop function ahead of the meter function. With this arrangement, the dropped packets are never registered by the meter function. Thus, when a flow exceeds its rate limits and packets are marked red, the drop function prevents those packets from being registered by the meter, reducing the information rate of the affected flow. Eventually, the meter will cease marking the packets red and the drop function will start to let packets through. So, over time, the policer/drop function will drop just enough packets so that those that are forwarded fall within the service level agreement that applies to the flow.

Shapers

Policers limit data rates by dropping excess packets. Shapers, on the other hand, rely on buffering to modulate data rates to remain at or below some configured limit. Figure 93 shows a policer in relation to buffering/queuing and a shaper.

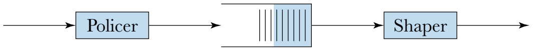


Figure 93 Policing, Buffering and Shaping

Policers drop packets as they come in from a network receive interface. A shaper modulates the rate at which a queue is allowed to deliver packets to a network transmit interface. Ideally, a policer and a shaper work in concert. The policer drops packets in order to prevent overrunning the queuing resources while the shaper works to smooth out the bursts without dropping packets, improving delay and congestion conditions further downstream. Figure 94 shows the effect of a shaper.

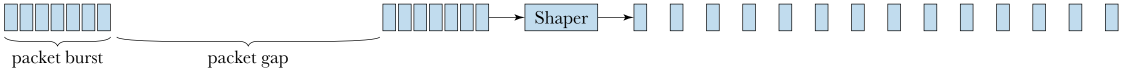


Figure 94 Shaper Effect on Packet Bursts

Essentially, a shaper takes in bursts of packets—or, more precisely, the packets are queued in a buffer upstream of the shaper—and then spaces the packets out in order to smooth the flow of data.

It may seem reasonable that, if all of the data source endpoints transmit their data into the network in a shaped fashion, there should be no need for shaper functions elsewhere in the network since the traffic is already well behaved. Unfortunately, as well-shaped packets arrive at a forwarding system, a certain amount of clumping is bound to happen as illustrated in Figure 95.

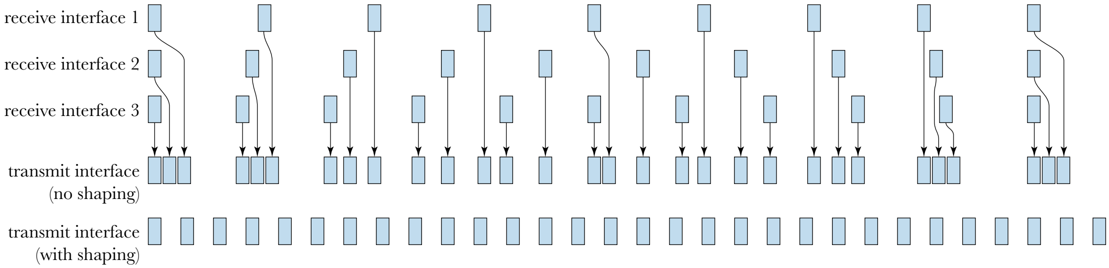

receive interface 1
receive interface 2
receive interface 3
transmit interface (no shaping)
transmit interface (with shaping)

Figure 95 *Packet Clumping*

When packets are received at a forwarding system via several interfaces either simultaneously or in an overlapping manner, some of those packets must be delayed slightly if they must be transmitted by a common interface in a serial manner. This inevitably leads to the formation of clumps of packets. A properly configured shaper, instead, imposes a little bit of additional delay so that the packets are spaced out, relying on the packet buffering that is upstream of the shaper to hold onto some of the packets during the additional delays.

The relatively simple (and completely contrived) example shown Figure 95 illustrates what's possible at just single forwarding system with just a few receive interfaces. If no shaper was in use and those clumps of packets were transmitted as un-shaped bursts, those bursts may eventually overlap with other bursts further downstream in the network, creating even bigger bursts. Eventually, these bursts grow large enough to trigger the marking of some of the packets red, even though the data rates offered by all of the data sources don't exceed any of the service level agreements. The use of shapers throughout a network prevents the formation of these bursty clumps of packets and consequent packet drops.

In a well-designed network with accurately provisioned service level agreements, the traffic shapers are busy smoothing out traffic and the policers are never compelled to drop packets.

Congestion Management Through Prioritization

Not all packets are created equal. Some are more important than others. For example, packets that convey important protocol information that keep a network operating smoothly are certainly more important than packets conveying an email whose content promises to make you young, rich, and thin. Other candidates for preferential treatment include packets related to interactive applications such as two-way voice/video communication, online gaming or industrial process control.

Head of Line Blocking

It is inevitable that a lot of traffic on any particular network will be non-provisioned (i.e., no advance reservation is made before packets begin transmission). These packets may react nicely to TCP flow control methods (RED, ECN, etc.), but will not automatically avoid causing congestion; much like cars on a single lane road trying to get past a traffic accident. In the scenario depicted in Figure 96 all vehicles are equally affected by the cause of the congestion, even those whose destination is before the accident and the emergency vehicles trying to get to the scene. This is known as head-of-line blocking.

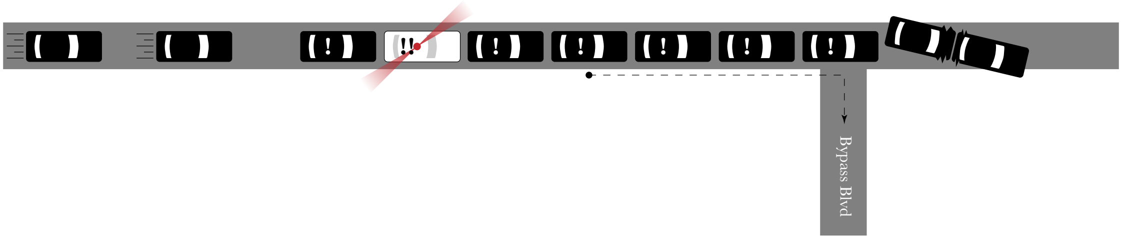


Figure 96 *Congestion on a Single-Lane Road*

If, instead of a single-lane road, there was a multi-lane road, where each lane was strictly reserved for certain kinds of traffic, Figure 97 is a much different picture.

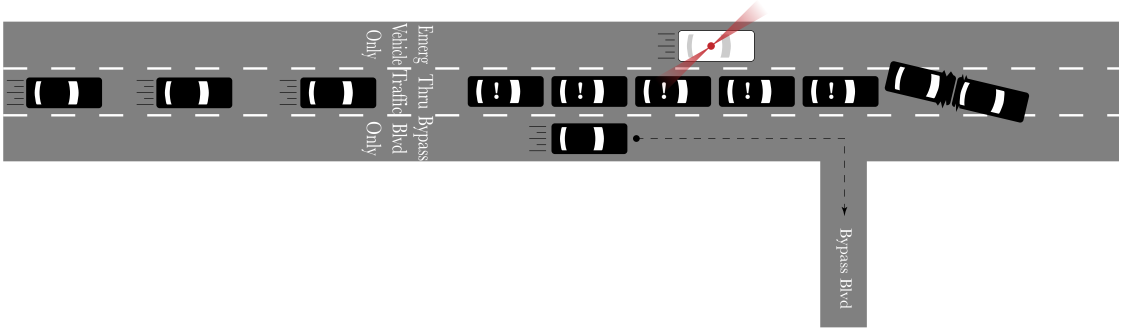


Figure 97 *Congestion on a Multi-Lane Road*

Now, high-priority traffic can quickly bypass the congestion, and traffic that isn't bound for a destination that runs near the accident can bypass all of the stuck traffic. Of course, this doesn't scale indefinitely. It isn't practical to build dedicated lanes for all possible destinations and traffic types.

In network forwarding systems, lanes are replaced with queues to solve head-of-line blocking problems.

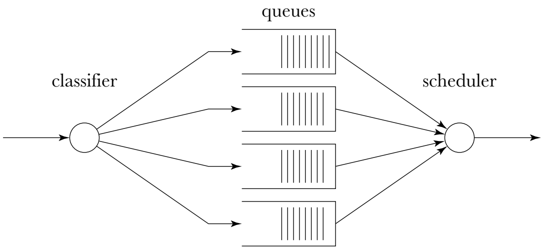


Figure 98 *Queues*

In Figure 98, packets arrive at a priority or behavior-aggregate classification function that feeds the packets into respective queues. The queues, in turn, converge on an arbitration/scheduling function that selects which queue to service next.

A queue is available to be serviced by the scheduling function whenever the queue is non-empty and whenever enough time has elapsed since the previous packet was popped from the same queue such that the queue's shaper function is satisfied. Once both of those conditions are met, the queue throws its metaphorical hat into the ring for the scheduler to consider.

If only one queue is ever ready to be serviced at any particular time, then the scheduling algorithm is pretty simple: just service the currently non-empty queue. It's when more than one queue is simultaneously ready to be serviced that things get interesting.

Scheduling

A huge variety of scheduling algorithms have been developed over the years. The algorithms described below are a representative sample of some of the more popular schemes.

Strict Priority

Strict priority is about as simple and fundamental as queue scheduling can possibly get. In zero time:

- If the highest priority queue is ready to be serviced, service it; else...
- Check the next lower priority queue. If it is ready to be serviced, service it; else repeat this step until all queues have been tested.

It is fairly easy to see how this works in practice. Let's say three priority levels have been established: a highest level for protocol packets, a medium level for streaming media and a lowest level for best-effort data packets. The flows associated with the high and medium priority levels have been provisioned such that, absent all other traffic, the sum of the two can never create congestion in the network. If a burst of medium-priority packets has been received and queued, they'll be serviced ahead of any low-priority data packets. The medium-priority queues are, however, forced to wait if any high-priority packets arrive. The low priority best-effort queue is only serviced when the high- and medium-priority queues are empty.

Difficulties arise, however, when it is deemed useful to further subdivide the non-provisioned, best-effort data packets into two or more priority levels. If the higher of the two best-effort traffic classes is the source of a sufficient amount of network data, its queues may rarely be non-empty, completely starving the lowest priority best-effort queue from being serviced. Round robin scheduling is used to ensure that the lowest priority queue gets at least some access to the transmit interface.

Round Robin

The simplest form of round robin scheduling operates over two or more queues that are considered equal for all intents and purposes. To understand how this works, simply consider a set of queues of equal priority that are all always ready to be serviced. To service them in a round robin fashion, service the first queue by popping a packet and then move on to the next queue in sequence and popping one packet from that queue. Upon finishing with the last queue, return to the first one. If any of the queues are not ready to be serviced, simply skip over them.

This simple scheme ensures that packets from all of the non-empty queues are neatly interleaved on the transmit interface. Of course, if you have several queues that are all being treated equally, you might as well enqueue all of those packets into a single queue in the order in which they arrive and service that single queue serially.

To provide preferential service to some queues without starving out other queues, weighted round robin scheduling is used.

Weighted Round Robin

In weighted round robin, each of the queues in a round robin set are assigned weighting values that represent the ratios (or weights) of the number of packets that the members of the round robin group are allowed transmit relative to the other member queues. For example, consider two queues in a weighted round robin group that are weighted 2 and 1. The queue whose weight is 2 is allowed to transmit twice as many packets as the queue whose weight is 1.

Queues that are not ready to be serviced are eliminated from the ratio computation. Consider instead, a set of three queues whose weights are 1, 2, and 4. If the 2-weight queue is not ready to be serviced while the other two are, then the remaining two queues are serviced with a packet ratio of 1:4.

The shortcoming of round robin and weighted round robin scheduling schemes is that they operate on packets, not bytes. If all the queues that are competing for access to a transmit interface are filled with packets that are all the same length, then each queue would be granted network bandwidth in a fair manner. Unfortunately, packets are not all the same length and it is very common for different types of traffic to have wildly different average packet lengths.

To account for this, weighted fair queuing was devised.

Weighted Fair Queuing

Weighted fair queuing differs from the round robin schemes previously described in that it takes the lengths of packets into account when choosing which queues to service. The objective is to time-sort a series of events—specifically, the starts and ends of packet transmissions—as if simultaneous packet transmission were allowed, portioning bandwidth to each of the overlapping packets in proportion to the queues' weights.

Figure 99 shows weighted fair queuing at work. The top third of the figure shows a number of packets of varying lengths queued in four queues: A, B, C, and D. The four queues are weighted 1, 2, 2, and 1, respectively.

In the middle third of the figure, the packets are shown in a time-proportional manner. This depiction assumes that it is possible to multiplex the packets at a bit granularity so that, for all intents and purposes, the packets from the four queues are being transmitted simultaneously. The weighting proportions of the queues dictate that queues B and C can transmit two bits for each bit transmitted by queues A and C. Notice that, when queues B and C have finished transmitting all of their queued packets, the bit rate for queues A and D jumps up to fill the void. The labels at the bottom of the time-proportional depiction mark the transmit completion times for each packet. This is the desired transmit order that weighted fair queuing strives to achieve.

The bottom third of Figure 98 shows the resulting packet sequence with normal packet length proportions. Notice that queues B and C are serviced faster than A and D, but A and D are not neglected.

Weighted fair queuing is very effective at providing ideal bandwidth sharing behavior. However, it is fairly computationally expensive. At each scheduling event, the length of the packets at the heads of all of the queues and the weights of those queues must be combined to compute the packet completion times and those results must then be sorted to identify the next queue to service. The workload is approximately O(log *n*) where *n* is the number of queues. The development of a scheduling algorithm that requires just O(1) workload per scheduling event brings us to weighted deficit round robin.

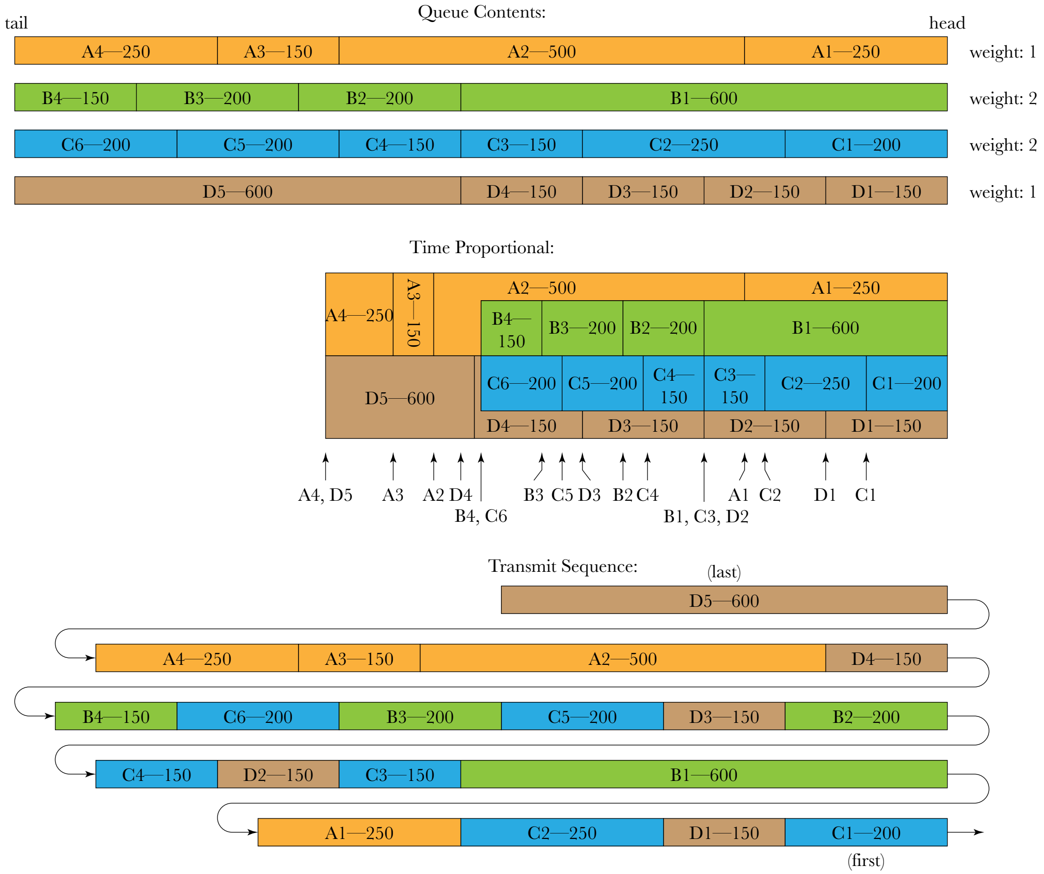
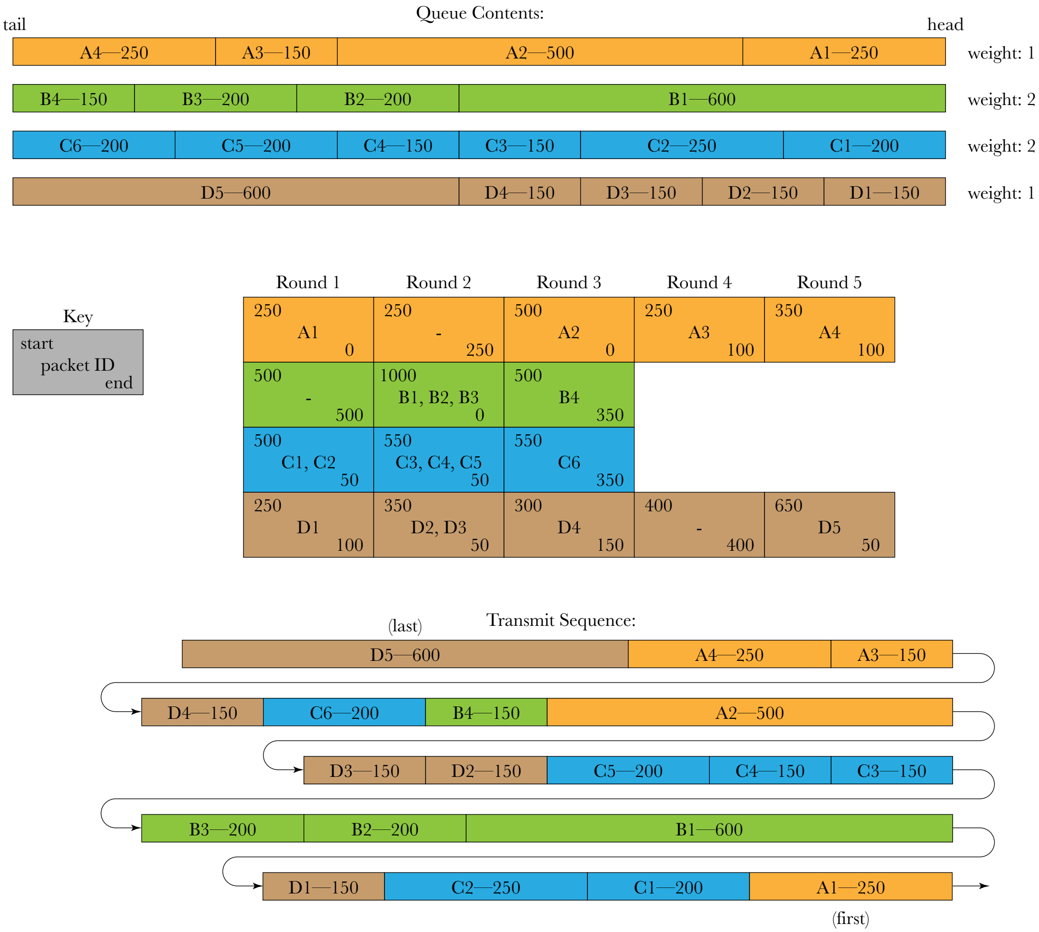


Figure 99 *Weighted Fair Queuing Behavior*

Weighted Deficit Round Robin

Weighted deficit round robin has behavior characteristics that are comparable to weighted fair queuing, but with greatly reduced workload. The algorithm is quite simple. During each arbitration round, each non-empty queue is awarded credits in proportion to their configured weights. A number of packets are popped from each queue where the sum of the number of bytes of the popped packets is less than or equal to the credits held by the queue. The queue's credits are decremented by the number of bytes just popped. Any remaining credit is carried over to the next round.

Let’s examine the same packet queuing scenario from the weighted fair queuing example and see how it behaves under weighted deficit round robin.



Round 1	Round 2	Round 3	Round 4	Round 5
250 A1 0	250 - 250	500 A2 0	250 A3 100	350 A4 100
500 - 500	1000 B1, B2, B3 0	500 B4 350		
500 C1, C2 50	550 C3, C4, C5 50	550 C6 350		
250 D1 100	350 D2, D3 50	300 D4 150	400 - 400	650 D5 50

Figure 100 *Weighted Deficit Round Robin Behavior*

Like the weighted fair queuing example, the top third of Figure 100 shows the contents of four queues (A, B, C, and D) and their respective weights (1, 2, 2, and 1). The center third shows the behavior of the weighted deficit round robin credit allocation scheme. Each rectangle contains three values. The upper left value shows the number of credits after a tranche of credit has been allocated to a queue at the start of a round. The value in the middle of each rectangle shows the ID of the packet(s) successfully popped during the current round. The value in the lower right corner shows the credit retained by the queue after popping the listed packets. For this example, the credit allocation quanta is 250 bytes. Thus, queues A and D receive 250 credits at each round while B and C receive 500.

In the first round, 250 credits are granted to queue A. This is sufficient to pop the 250-byte packet A1, leaving queue A with 0 credits. Next, queue B is granted 500 credits. (Remember, queue B has a weight of two whereas A's is one.) This is not sufficient to pop the 600-byte packet at the head of queue B, so no action is taken and queue B retains the full 500 credits. This process continues with queues C and D. After completing round 1, round 2 starts and the overall process continues until all of the queues are empty.

The resulting packet transmit sequence is shown in the bottom third of Figure 100. The sequence differs in the details from the very precise weighted fair queuing, but, in the aggregate, the results are the same. Specifically, for as long as all of the queues are non-empty, queues B and C are granted twice as much bandwidth as A and D. Once B and C are empty, A and D split all of the available bandwidth evenly between them.

Priority Flow Control (IEEE 802.1Qbb)

Even with fancy queue scheduling schemes in use, queue congestion is still a very real possibility. Exerting a flow control indication to the source of the offending traffic is one way for mitigating the onslaught of packets. However, using the beat-up, rusty old hammer of a tool that is link pause (IEEE 802.3x) would be a poor choice when priority-based queues are being used. Why pause an entire link when, perhaps, just one priority is experiencing congestion? This is where priority flow control comes into play.

IEEE 802.1Qbb priority flow control (often referred to as PFC) extends 802.3x link pause to support reporting flow control indications on a per-priority basis.

802.1Qbb priority flow control messages are conveyed by Ethernet packets using the reserved multicast `destinationAddress` value `01-80-c2-00-00-01` and `ethertype` of `0x8808` (MAC control). The format and field definitions of the flow control message are shown in Figure 101 and Table 33.

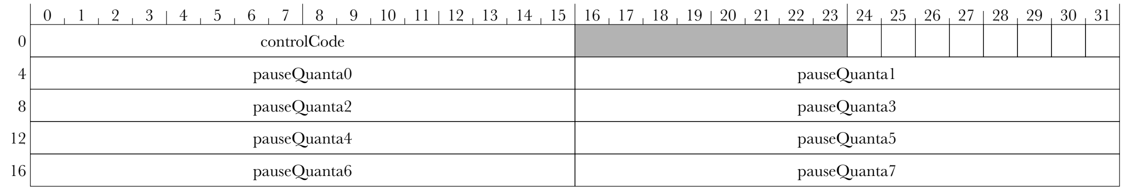


Figure 101 Priority Flow Control Header Structure Diagram

Table 33 Priority Flow Control Header Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`controlCode`	16 bits	0.0	Defines the MAC control code. For priority flow control, the code is `0x0101`.
`priorityEnable7`	1 bit	0.24	Per-priority enables. Each bit in this vector corresponds to one of the eight priority levels supported by 802.1Qbb. When set to `1`, the corresponding `pauseQuanta[0...7]` value is considered valid. When a `priorityEnable[0...7]` bit is set to `0`, the corresponding pause quanta value is considered invalid. However, the 16-bit pause quanta fields are present in this message regardless of the state of these bits.
`priorityEnable6`	1 bit	0.25	
`priorityEnable5`	1 bit	0.26	
`priorityEnable4`	1 bit	0.27	
`priorityEnable3`	1 bit	0.28	
`priorityEnable2`	1 bit	0.29	
`priorityEnable1`	1 bit	0.30	
`priorityEnable0`	1 bit	0.31	
`pauseQuanta0`	16 bits	4.0	The pause duration as measured in Ethernet slot times. An Ethernet slot time is 512 bit-times. So, the pause quanta is link bit-rate dependent. Up to 65,535 slot times may be specified for the pause duration. At, say, 10 Gbps, that's 6.55 µs. Each pause message resets an Ethernet MAC's pause timer to the value in the pause message. A message-sending Ethernet forwarding entity may transmit periodic pause messages in order to prevent its link partner from transmitting indefinitely. A pause time of zero takes a MAC out of the pause state and enables the immediate resumption of packet transmissions.
`pauseQuanta1`	16 bits	4.16	
`pauseQuanta2`	16 bits	8.0	
`pauseQuanta3`	16 bits	8.16	
`pauseQuanta4`	16 bits	12.0	
`pauseQuanta5`	16 bits	12.16	
`pauseQuanta6`	16 bits	16.0	
`pauseQuanta7`	16 bits	16.16	

The behavior of priority flow control is essentially identical to link-level flow control except that it operates on individual priority levels instead of an entire link. Ethernet link partners that receive a priority pause message are expected to immediately cease transmitting packets from the paused priority levels for the specified amount of time.

14 Time Synchronization

Time Sync Primer

The IEEE 1588–2008 Precision Time Protocol (PTP) standard specifies a means for synchronizing clocks across asynchronous, packet-based networks with sub-microsecond accuracies. IEEE 1588 can operate over legacy networks where just the endpoints—i.e., the master and slave clock nodes—have special hardware provisions. However, by upgrading the midpoint systems, faster convergence times and improved accuracies at lower computational costs are possible.

Clocking Hierarchies

IEEE 1588 arranges itself into a hierarchy of clock systems. This is, in many ways, analogous to the way that IEEE 802.1D bridges arrange themselves into a spanning tree. A root (or master) is chosen and loops are eliminated by disabling redundant or circular paths.

For any particular timing domain (roughly analogous to a VLAN), a grand master is chosen using a “best master clock” algorithm. Once this master is chosen, all of the slave clocks synchronize themselves to this master. If the grand master clock fails or becomes unreachable, a new grand master is chosen.

In Figure 102, the various types of clocks are depicted. The (M), (S) and (T) designations on ports indicate whether they operate as masters, slaves or transparently, respectively. A master port is an origin of time information. A clock domain has just one grand master but may have many master ports in the form of downstream-facing boundary clock ports. Ordinary clocks and boundary clocks may have just one slave port per clock domain. Transparent clocks have neither master nor slave ports as they do not terminate timing from a master; rather, they pass the timing information along toward a downstream slave.

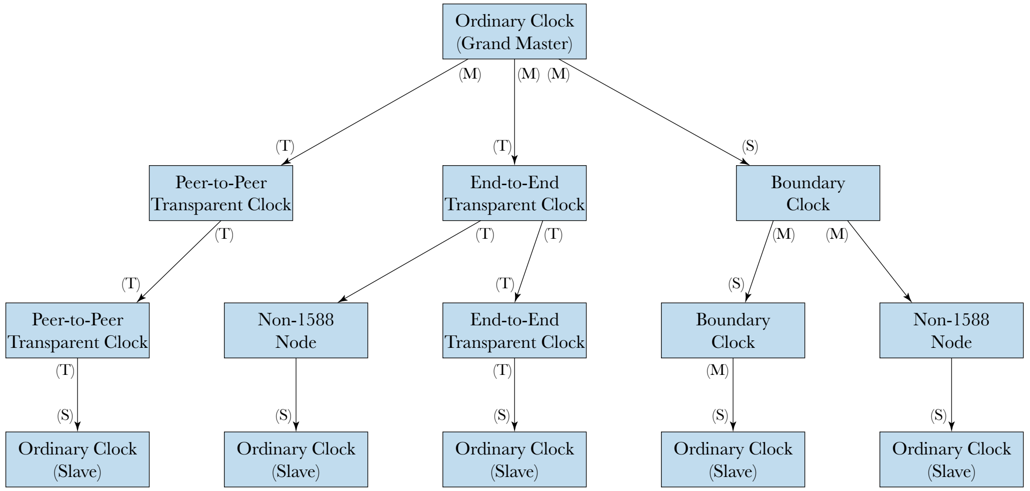


Figure 102 *Time Sync Hierarchy*

Types of Clocks

A 1588 clocking hierarchy must have at least two ordinary clocks (one master and one slave) and may have any number of boundary clocks, end-to-end transparent clocks and peer-to-peer transparent clocks interposed between the master and the slave(s). These various types of clocks are explained in the following sections.

Ordinary Clocks

Ordinary clocks are used to terminate timing domains. A timing domain must have one grand master and at least one slave ordinary clock. A grand master ordinary clock may have one or more active master timing ports—i.e., ports that support the 1588 protocol. Slave ordinary clocks may only have one active slave timing port per domain. In other words, within a domain, a master may deliver time sync information to multiple slaves while a slave may only have a single master.

A grand master ordinary clock typically uses some kind of external time reference. Examples of time references include GPS, atomic clocks and local crystal oscillators. So, in effect, a grand master ordinary clock is a slave to this timing reference.

Conversely, a slave ordinary clock typically has the ability to generate clock output signals for use by various kinds of time-dependent equipment and processes.

Any number of boundary or transparent clocks or even non-1588 forwarding nodes may be interposed between a master ordinary clock and its slave ordinary clocks. While non-1588 equipment may be used to build a time-aware network, their unpredictable (and not easily measured) latencies contribute to longer convergence times, reduced accuracy and increased cost and complexity of the filtering algorithms or circuits required to recover the grand master's timing information at the slave ordinary clocks.

Boundary Clocks

A 1588 boundary clock terminates a timing domain much in the same way that a router terminates a bridged network's broadcast domain. A simple way to think of a boundary clock is that it is a series of ordinary clocks that are integrated into a single network node. These separate ordinary clocks share a common time reference that is synchronized to the grand master. One of the ordinary clocks acts as a slave to a master further up in the hierarchy, while the remainder of the ordinary clocks act as masters to the nodes further down in the hierarchy.

Because the only thing that is shared by these ordinary clocks that make up a boundary clock is the synchronized timing information, the various non-administrative PTP messages sent to a boundary clock are never forwarded. Instead, the communications are terminated at the boundary clock.

Terminating these messages makes it possible to scale a time distribution network by breaking it up into zones or levels. Rather than the grand master having to communicate with, potentially, many thousands of slaves, it need only communicate with a much small number of boundary clocks that, in turn, perform the role of master for the slave ordinary clocks (or further boundary clocks) beyond it.

There are two disadvantages to boundary clocks. First, they are a bit more complex and more expensive than a non-1588 bridge or router or a 1588-compliant transparent clock (discussed below). This expense comes from having to fully terminate and recover the grand master's timing information. This can be computationally expensive and/or require sophisticated clock recovery circuits.

Second, the clock recovery process introduces errors and inaccuracies into the clock that are passed along to subsequent clock nodes. If several boundary clocks are interposed between a grand master and a slave, the inaccuracies may compound to the point where the slave's recovered clock is no longer suitable for the intended application.

Transparent Clocks

Transparent clocks come in two forms: end-to-end and peer-to-peer. Though they share a lot of common functionality, there are important differences between these two types of transparent clocks. Their type-specific behavior is described in subsequent subsections. Below is a summary of their common characteristics.

Transparent clocks differ from boundary clocks in that they do not terminate all timing messages. Instead, certain PTP messages are forwarded by transparent clocks to their addressed destinations. Unlike a non-1588 bridge or router, though, transparent clocks compensate for their forwarding delays. Therefore, a transparent clock must be able to accurately measure the reception time and transmission time of these PTP messages in order to make the necessary corrections with a high degree of accuracy.

For maximum accuracy, a transparent clock's local time reference should run at the same frequency as the grand master's clock. Because a transparent clock is only concerned with forwarding delays (known as "residency time" in the 1588 standard), all of its measurements are deltas between receive times and transmit times. Thus, a transparent clock does not need to know the time of day. A clock that is frequency-locked to a master but not phase-locked is referred to as being "syntonized." A clock that is both frequency and phase locked to a master is referred to as being "synchronized." Transparent clocks benefit from syntonized clocks but gain no additional benefit from synchronized clocks.

By eschewing clock synchronization, a transparent clock need not compute the transmission delay between it and the master clock. This reduces the complexity of the transparent clock and reduces the computational load on the master clock.

It is important to point out that clock syntonization is not strictly required by a transparent clock. A transparent clock can operate perfectly adequately using a free-running, asynchronous time base, but can achieve somewhat higher degrees of accuracy if the time base is syntonized. The key is the transparent clock's residency time. If the residency time is kept small, then the amount of error that a free-running clock can introduce in measuring this residency time is minimized. If a forwarding system's architecture is such that long delays (multiple microseconds) are commonplace for even high-priority traffic, then using a syntonized time base is beneficial.

End-to-End Transparent Clocks

An end-to-end transparent clock gets its name from the way in which delay measurements are performed. End-to-end transparent clocks forward delay measurement messages between a master and its slave in largely the same way that a non-1588 bridge or router does. This means that a slave is measuring the delay between itself and the master. The end-to-end transparent clock, however, makes adjustments to the delay measurement messages so that the slave is able to compensate for the highly variable queuing delays that can occur within forwarding systems. By doing this, the delay value that remains is just the link delay (i.e., the delay "on the wire"). Because link delay values change very slowly and by relatively small amounts, compensating for these delay variations is fairly easy.

So, like a non-1588 bridge or router, an end-to-end transparent clock forwards the timing protocol messages between a master and a slave. But, unlike a non-1588 bridge or router, an end-to-end transparent clock measures the queuing delays of these messages and makes adjustments to certain parameters so that the variable forwarding delays can be factored out with ease.

Peer-to-Peer Transparent Clocks

Unlike an end-to-end transparent clock, a peer-to-peer transparent clock actually terminates delay measurement messages rather than forwarding them. Because of this, a peer-to-peer transparent clock is able to measure the delay on the medium via which it communicates with its link peer. This means that a peer-to-peer transparent clock can compensate for both the link delay and the forwarding delay between the master and its slaves.

The significant benefit of the peer-to-peer transparent clock is that the master does not have to process the delay measurement messages from all of the slaves associated with the master. Instead, it need only process the delay measurement messages from the directly attached peer-to-peer transparent clocks. In a large network, this can result in a significant reduction in a master clock's workload.

For peer-to-peer transparent clocks to work, the entire path between a master and a slave (ordinary or boundary clocks) must be capable of peer-to-peer transparent clock operation. In other words, a clock can only operate in the peer-to-peer transparent clock mode if it is directly attached to a link peer that also supports peer-to-peer delay measurement operations. Ordinary and boundary clocks (in both master and slave roles) can be configured to support peer-to-peer delay measurement.

One interesting aspect of peer-to-peer transparent clocks is that the delay measurement method should be active on all ports even if those ports are disabled by PTP for timing use (i.e., redundant paths to the master) or blocked by the spanning tree protocol. The reason for this is that it enables a quick transition to a new master clock by avoiding having to perform a link delay measurement prior to using the link.

Precision Time Protocol

Let's review the operation of the 1588 precision time protocol (PTP).

Message Encapsulation

The 1588 precision time protocol depends on the exchange of a series of protocol-specific messages. These messages may be encapsulated in one of three ways:

- Ethernet
- IPv4/UDP
- IPv6/UDP

When transporting PTP messages over Ethernet, the packets carrying these messages are distinguished by their MAC `destinationAddress` and `ethertype` values. Two different MAC destination addresses are available for 1588 PTP messages:

01-1b-19-00-00-00 and 01-80-c2-00-00-0e. The 01-1b-19-00-00-00 destination address is used for all PTP messages except PTP peer delay messages. To ensure that PTP peer delay messages (used by the peer-to-peer transparent clock method) are not blocked by spanning tree protocols, 01-80-c2-00-00-0e is used for such messages.

All PTP messages share the **ethertype** value 0x88f7.

When transporting PTP messages over IPv4 and UDP, the IPv4 destinationAddress value 224.0.1.129 is used for all messages except peer delay messages while 224.0.0.107 is used for peer delay messages. The UDP destinationPort number for all PTP event messages is 319 while general messages use port 320. (PTP message types are discussed a bit further below.)

When transporting PTP messages over IPv6 and UDP, the IPv6 destinationAddress ff0x::181 is used for all messages except peer delay messages while ff02::6b is used for peer delay messages. The “x” in the IPv6 destination address is substituted according to the values listed in Table 34.

Table 34 PTP Primary IPv6 Address Value

Value for “x”	Meaning
1	interface — local scope
2	link — local scope
4	admin — local scope
5	site — local scope
8	organization — local scope
e	global scope
0, 3 & f	reserved
all others	unassigned

Message Types

All PTP messages are of one of two types: event messages or general messages. Event messages are distinctive because they trigger timestamping events in the hardware. Both of these message types are discussed in the following subsections.

Event Messages

The receipt or transmission of a PTP event message triggers a hardware-based timestamp mechanism. The purpose of this mechanism is to capture the exact time that the packet’s reference point passes the receive or transmit interface. According to the 1588 standard, the timing reference point is the dividing line between the Ethernet packet’s preamble and the first bit of the packet itself. This is the leading edge of the least significant bit of the first byte of the MAC destinationAddress field (i.e., the "multicast" bit). In practice, of course, every packet may trigger a timestamp event and subsequent processing may filter out those timestamps that do not belong to PTP event messages. Table 35 lists and defines the various PTP event messages.

Table 35 *PTP Event Messages*

messageType	Name (std. name)	Description
0	`Sync` (Sync)	Conveys the master clock's time of day information to its slave clocks.
1	`DelayRequest` (Delay_Req)	Requests the master to generate a `DelayResponse` message. Used to measure the master-to-slave delay time.
2	`PeerDelayRequest` (Pdelay_Req)	Requests the link peer to generate a `PeerDelayResponse` message. Used to measure the link delay time between link peers.
3	`PeerDelayResponse` (Pdelay_Resp)	Generated in response to a `PeerDelayRequest` message.

General Messages

PTP general messages fall into two broad categories: messages that convey timing data (i.e., interesting messages) and messages that are used for the control and management of the timing network (i.e., boring messages). Table 36 lists and defines the various PTP general messages.

Table 36 *Table 3 PTP General Messages*

messageType	Name (std. name)	Description
8	`FollowUp` (Follow_Up)	Provides the slave with the corresponding `Sync`'s transmission time in a two-step timing network.
9	`DelayResponse` (Delay_Resp)	Generated in response to a `DelayRequest` message.
10	`PeerDelayResponseFollowUp` (Pdelay_Resp_Follow_Up)	Provides the delay-requesting peer with the corresponding `PeerDelayResponse`'s transmission time in a two-step timing network.
11	`Announce` (Announce)	PTP control message.
12	`Signaling` (Signaling)	PTP control message.
13	`Management` (Management)	PTP control message.

Clock Domains

1588 PTP timing networks support multiple clock domains. Clock domains are somewhat analogous to the per-VLAN spanning tree protocol. Each clock operates independently of the others, each domain has its own grand master clock, and a single PTP system may operate in multiple domains.

Clock domains are distinguished from one another by the eight-bit `clockDomain` field that is common to all PTP messages.

Multiple clock domains may be supported by all PTP clock types: ordinary clocks, boundary clocks and transparent clocks.

One-Step vs. Two-Step Operation

All PTP systems (ordinary clocks, boundary clocks and transparent clocks) must include hardware support for accurately timestamping the reception and transmission of PTP event messages. However, there are two levels of hardware support for the conveyance of transmit timestamps of the event messages: one-step and two-step.

The one-step method requires that the timestamp of a transmitted PTP event message be incorporated into the transmitted event message itself. This means that the transmit Ethernet MAC must be able to both timestamp the message (i.e., measure its transmit time) and perform a number of arithmetic and rewrite operations on the very same Ethernet packet. At very high data rates this can be quite challenging. The following operations are typically required:

- timestamp transmit PTP event messages
- add timestamp value to the messages' 64-bit `correctionField` value
- update `Udp.checksum` (if applicable)
- compute the packet's `Mac.crc` using updated packet contents

The principal benefit of one-step operation is improved accuracy in transparent clocks (residency time is minimized) and greatly reduced CPU overhead in all clock types.

Two-step clocks, while computationally more expensive (from a CPU's perspective) are comparatively simple from a hardware implementation point of view. Rather than attempting to incorporate a transmit timestamp into an event message "on the fly," a follow-up message containing the timestamp-related information is sent some time after the event messages. A follow-up message is correlated with an event message by the receiving device by simply comparing the messages' `clockDomain` and `sequenceId` fields. A match on these two fields indicates that the messages are related.

There are two types of follow-up messages:

- `FollowUp`
- `PeerDelayResponseFollowUp`

Both of these messages are general messages and not event messages (i.e., they are not, themselves, timestamped). Follow-up messages may be transmitted any time after their corresponding event messages. However, it is recommended that the follow-up messages be transmitted as quickly as possible and that they are transmitted prior to the transmission of a subsequent event message.

A single PTP system may operate in both one-step and two-step modes simultaneously—even on the same physical port. The PTP system must examine each PTP event message's `twoStep` bit to determine how to handle the message. If this bit is set to `1`, then two-step operation is required. Otherwise, one step operation is required. A node that is not capable of transmitting in the one-step mode may receive a one-step event message and transmit a two-step message pair.

Time Sync Methods

Now that the various types of PTP clock systems and messages have been introduced, the actual time sync methods are described.

Time Representations

There are two means for representing time that are of interest. The first of these is the time interval. Time intervals are used for measuring the time that has elapsed between events. A specific example of their use is the `correctionField` value that appears in all PTP message types. A time interval is a 64-bit integer that counts 2^{-16} nanosecond units of time. This representation gives a time interval value a resolution of 1/65,536 of a nanosecond and a maximum value of 281,475 seconds, or 3 days, 6 hours, 11 minutes and 15 seconds.

For example, a time interval value of 2.5 nanoseconds is represented as: `0x0000_0000_0002_8000`.

The other time representation is the timestamp. This is an absolute time counting from the start of a particular epoch. The default epoch for PTP started at the first instant of January 1st, 1970 (UTC). A timestamp is an 80-bit value that is divided into two parts: a 48-bit seconds part and a 32-bit nanoseconds part. A 48-bit seconds value means that an epoch has a maximum duration of 8.926 million years. Therefore, there should be no need to ever use an alternate epoch.

It is important to note that the 32-bit nanoseconds part of the timestamp value is an actual representation of integer nanoseconds and does not represent fractions of a second. Therefore, the nanoseconds value does not roll over at 2^{32}-1. Instead, its value is always less than 10^9.

Syncing a Slave to a Master

The simplest way to visualize clock synchronization is to first assume that there exists some kind of zero-delay communications medium (we'll deal with delays later). In this imaginary world, time sync is simply a matter of multicasting a Sync message from the grand master clock to all of the slaves.

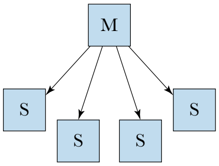


Figure 103 Simple Master-Slave Time Sync Scenario

The grand master timestamps the precise transmit time of its Sync message using its local clock value (ostensibly the correct, absolute time of day) and the slave clocks each capture a timestamp of the receipt of the Sync messages using their own local time value. Because of our magical zero-delay transmission medium, these transmit and receive timestamp events occur simultaneously. By simply subtracting the transmit timestamp conveyed by a Sync message from its receive timestamp, a slave clock can determine the offset that is required to bring its local clock into sync with the master:

```
clockOffset = syncTransmitTimestamp – syncReceiveTimestamp
synchronizedTime = localTime + clockOffset
```

By repeating this process with some regularity, a slave clock can determine the ratio of the grand master's clock to its local clock and make the necessary rate adjustments to its local clock so that the slave clock may track the grand master accurately between Sync messages. Sync messages are typically transmitted 10 times per second. However, rates well above and well below this rate are allowed.

In reality, of course, all transmission media impose delays and, even worse, forwarding systems (1588 and non-1588, alike) installed between a master and a slave can impose delays that are both significant and highly variable. The next section discusses how transmission delays are measured and utilized.

End-to-End Delay Measurement

If the delay between a master and a slave cannot be measured, it is possible to achieve syntonization between the grand master and a slave (i.e., frequency lock) but not synchronization (i.e., phase or time of day lock). The first step in measuring the delay between a master and a slave is to syntonize the slave's clock to the grand master. Doing so increases the accuracy of the delay measurements because both ends of the system then have a common definition of the duration of a second.

Delay measurements require a two-way measurement of the path between a master and a slave. This is necessary because the master and slave clocks, though syntonized, are operating with an unknown offset between them. The `Sync` message sent from a master to its slaves provides a slave clock with one half of the necessary data: the transmission and reception times of the `Sync` message. Of course, with the unknown offset between the master and slave clocks, this data is not useful on its own for measuring the path delay.

Part two of the path delay measurement process is initiated by the slave clock. It sends a `DelayRequest` message to its master clock. The transmit timestamp of this message is captured and retained by the slave clock. The master clock captures the `DelayRequest`'s receive timestamp and returns it to the slave via a `DelayResponse` message. This exchange of messages is diagrammed in Figure 104.

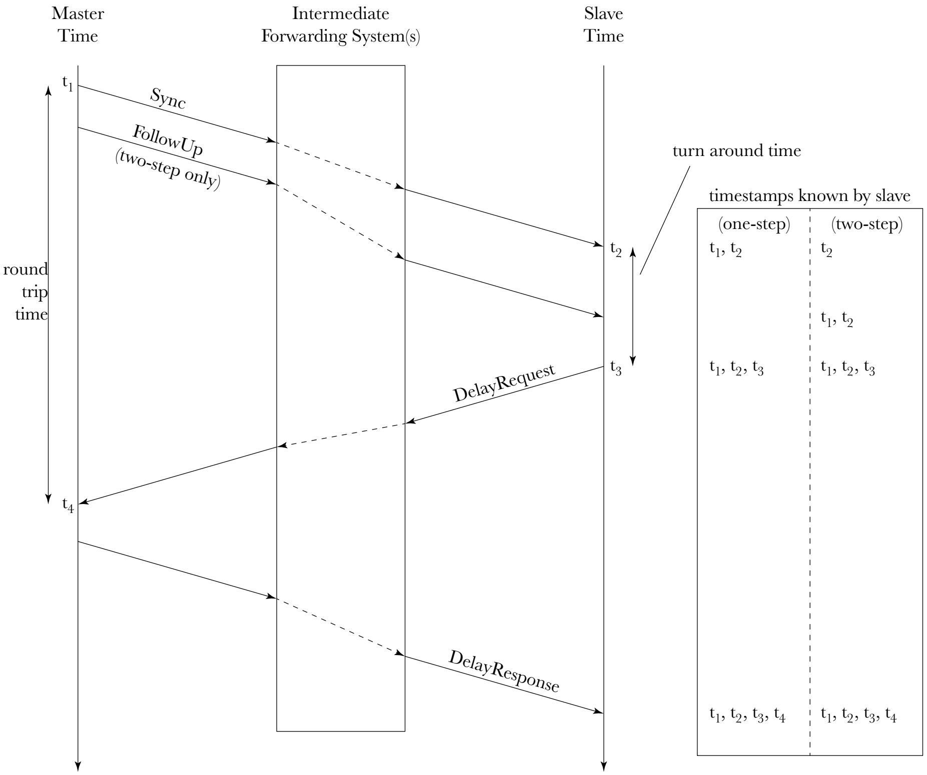


Figure 104 *Basic Sync and Delay Message Exchange*

In one-step time sync, the `Sync` transmit time (t_1) is embedded in the `Sync` message, so the slave knows t_1 and t_2 (`Sync` receive time) upon receipt of the `Sync`. Two-step time sync does not embed the transmit time in the `Sync` message. A `FollowUp` message is used instead to convey t_1 to the slave. Because t_4 (DelayRequest receive timestamp) is measured at the master, this value is conveyed back to the slave via a DelayResponse message.

An intuitive way to view the path delay is that the difference between t_4 and t_1 represents the round trip time, while the difference between t_3 and t_2 represents the turnaround time between a Sync message and a DelayRequest message. By subtracting the turnaround delay from the total path delay and dividing by two, the one-way path delay can be computed:

$$\text{pathDelay} = \frac{(t_4 - t_1) - (t_3 - t_2)}{2}$$

The path delay value is used to correct the offset in the slave's clock to exactly synchronize it to the master.

Though Sync and DelayRequest messages are depicted as being paired, they are actually transmitted at different rates in practice. Because link delays vary comparatively slowly over time (relative to the drift rates of crystal-based local clocks), the delay measurement messages are typically sent about once per second. Of course, higher and lower rates are supported.

The intermediate forwarding systems depicted in Figure 104 introduce a variable and unpredictable amount of delay in the path between the master and the slave. It is possible to filter out this delay variation, but this filtering is quite complex, time consuming and tends to diminish the accuracy of the slave clocks. To address this problem, transparent clocks were introduced into the standard. By measuring the residency time and accumulating it into a correction value, it is possible to zero out the effects of a transparent clock's variable queuing delays.

There are two versions of transparent clocks defined:

- end-to-end transparent clock
- peer-to-peer transparent clock

These are described in detail in the following two subsections.

End-to-End Transparent Clock Delay Measurement

The delay measurement message exchange sequence between a master and a slave for end-to-end transparent clocks is illustrated in Figure 105.

The delay request and response messages are sent between a slave and its master via any intervening transparent clocks. The accumulation of useful time information occurs as depicted. The slave directly captures time t_1 while the receive time (t_2) is delivered from the master directly in DelayResponse. The transparent clocks' participation in this exchange is limited to making adjustments to DelayRequest and DelayResponse so as to compensate for their queuing and forwarding delays.

In two-step operation, the transparent clocks keep track of `DelayRequest` and corresponding `DelayResponse` message time parameters so as to compensate for queuing delays.

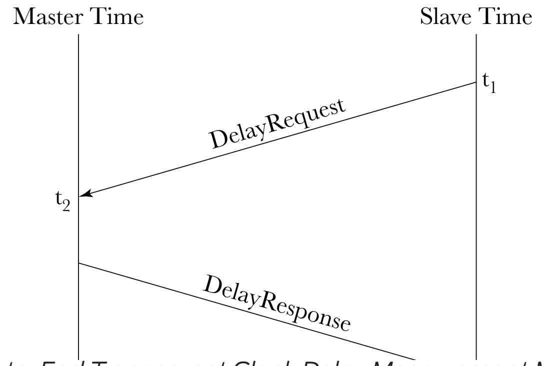


Figure 105 End-to-End Transparent Clock Delay Measurement Message Exchange

The ultimate goal of end-to-end delay measurement is to measure the wire delays and not the queuing delays. The queuing delays are measured during sync operations (by computing and compensating for residency time through each transparent clock) and the previously-measured wire delays are then used to determine the absolute offset between the master and the slave.

Peer-to-Peer Transparent Clock Delay Measurement

The delay measurement message exchange sequence between link peers for peer-to-peer transparent clocks is shown in Figure 106.

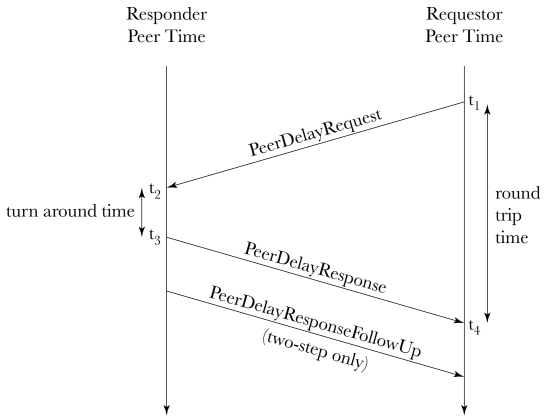


Figure 106 Peer-to-Peer Transparent Clock Delay Measurement Message Exchange

Each peer-to-peer transparent clock and each slave ordinary or boundary clock generates PeerDelayRequest messages on each of the ports that are directly attached to a clock that is capable of peer-to-peer link delay measurement. This is true even of the ports that are not currently being used for time sync (i.e., not receiving Sync messages) and for ports that are blocked by spanning tree. Maintaining current link delay measurements for all ports reduces the recovery time if it becomes necessary to switch from one grand master to another or if the path between the grand master and a slave should change.

PeerDelayRequest, PeerDelayResponse and PeerDelayResponseFollowUp messages are never forwarded by peer-to-peer transparent clocks. There is no need for the slave clock to send DelayRequest messages directly to the master clock. Instead, it sends PeerDelayRequest messages to its link peers. Hence, the master clock receives PeerDelayRequest messages from only the directly connected peer-to-peer transparent clocks rather than DelayRequest messages from all of the slaves in the clock domain. This significantly reduces the master clock's CPU workload.

Nuts and Bolts of Time Sync

IEEE 1588-compliant, time-aware forwarding systems have some specific hardware requirements. First and foremost, the Ethernet media access controllers must be able to generate accurate packet receive and transmit timestamp values.

Timestamps

A packet is said to begin reception at the divide between the start-of-packet delimiter (i.e., the last bit of the Ethernet preamble) and the first bit of Mac.destinationAddress. The start-of-transmission point in a packet is similarly defined. For both receive and transmit timestamps, the timestamps must be measured as close to the physical medium as possible. By "close," it is not meant to imply spatial closeness, but temporal closeness. If there are delays between the physical medium and the timestamp point, these are, ideally, fixed delays. Any fixed delays may simply be accounted for as propagation delays along the physical medium. It is also ideal for any delay between the physical medium attachment point and the timestamp point to be the same for receive and transmit. This is important because the one-way delay is assumed to simply be one half of the round-trip delay. Skew between receive and transmit delays will reduce the accuracy of the one-way delay assumption. Any skew must be characterized and included in a compensation computation for the one-way delay.

All receive packets, regardless of their type, must be timestamped upon detection of the start of reception. This is required because there is no way to know which packets are PTP event messages without first receiving and examining the PTP message. Unneeded timestamp values may be disregarded.

Transmit timestamping behavior depends on whether the forwarding system is operating in one-step or two-step mode. In one-step mode, the hardware must update `correctionField` as required during the transmission process. Specialized hardware within the Ethernet media access controller is required to perform these updates. For two-step operation, the transmit timestamp value is returned to the forwarding system's control plane so that an appropriate follow-up message may be generated and transmitted. To correlate a transmit timestamp value with a particular transmit packet, certain identifying information (transmit interface ID, `clockDomain`, `sequenceId`) must be included along with the timestamp value.

To enable timestamping, a time-aware forwarding system must have a local timebase. This local timebase can either run off of a local, asynchronous clock source (e.g., a crystal oscillator) or a frequency-locked clock source (e.g., synchronous Ethernet). In the case of an asynchronous clock source, the timebase counter must increment by a time value that represents the count interval period of the grand master clock as expressed in units of the local clock source. For example, if we presume that the timebase must count nanoseconds and it is determined by comparing a sequence of `Sync` messages that the local asynchronous clock source is actually running at 501 MHz instead of the 500 MHz at which it is supposed to operate, then the local timebase counter must increment by 1.996 on every clock instead of 2.000. Thus, the local timebase counter will advance time at the same rate as the timing grand master. The timebase counter's increment amount is adjusted dynamically as continuous analysis of `Sync` messages is performed in order to compensate for variable drift between the timing grand master and each timing slave.

For synchronous Ethernet, there is a continuous timing chain from the timing grand master to each time-aware forwarding system. Therefore, no such compensation and tracking is necessary.

Timebase Counters

There are two types of timebase counters: an interval counter and a time-of-day counter. It is possible to build a time-aware forwarding system with just one type of counter, but often both types are present. For time sync purposes, the interval counter is the most useful form since it makes `correctionField` math quite simple. The time-of-day counter can also be used for this math, but, more importantly, it is really quite useful for OAM applications where a time-of-day value is required.

For a maximal implementation, two separate but synchronous counters are used for easy operation of the PTP time sync protocols. The first counter counts seconds and nanoseconds (time-of-day). The second counter counts nanoseconds and fractions of a nanosecond (interval).

The reason that two separate counters are required is that the interval counter's nanoseconds value rolls over at 2^{48}-1 nanoseconds (a little over three days) while the time-of-day's nanoseconds value is always less than 10^9 (exactly one second). Maintaining two counters that advance at exactly the same rate (but with two different modulo values) makes the conversion from one time space to the other a fairly simple matter.

The `intervalNanoseconds` value is what is captured by the network and device-to-device interfaces when PTP event messages are received or transmitted. This value is subsequently used as the basis for adjustments to PTP messages' `correctionField` value.

> NOTE The names used here for the two counter systems can cause some confusion with the terms defined in the IEEE 1588 standard. For these counters, the timestamp value represents an absolute time of day reference. The time-of-day counter's value is not actually used directly by the timestamping hardware to measure receive and transmit times. Instead, the `intervalNanoseconds` counter is used. Because `intervalNanoseconds` operates modulo 2^{48} nanoseconds, computing intervals between timestamp events is made much simpler.

Drift and Offset Adjustment

In a large system, the devices that receive, forward and transmit packets will, inevitably, be driven by separate, local crystals. Hence, they will all operate at somewhat different rates. Even those devices that operate off of a common crystal clock source will likely emerge from reset at varying times, introducing an offset between their notions of the time of day. Of course, it is also unlikely that any system built today can come out of reset on January 1st, 1970. So, not only is there an offset between devices, there's a huge offset between these devices and the start of the time domain's epoch.

The drift and offset controls are used to correct for these differences among the various local clocks and between these local clocks and the grand master. Drift control was described briefly above. Offset adjustment is a bit more tricky. Generally, introducing a significant discontinuity in time by suddenly adding, say, 20 years to the time of day can seriously confuse things. However, during system startup, introducing an offset is necessary in order to get the clocks in fairly close alignment with the grand master. Once an offset has been established, small adjustments to the timebase counters' increment values are used to maintain frequency and phase lock with the grand master clock. So, the offset is set once and then small rate adjustments are used thereafter to maintain sync.

CPU and External Event Timestamping

In order for the CPU to participate in time sync operations and make any necessary drift and offset adjustments, it must be able to determine the value of the various timebase counters at some instant. Similarly, when an external event occurs, such as, for example, the assertion of a one-pulse-per-second sync signal input, the time at which that event occurred must be measured so that the CPU can compare the external time of that event and the internal time of that same event.

In support of these requirements, the hardware must have multiple sets of sampling registers that simultaneously capture all of the timebase counter values for subsequent examination by the CPU. One set of timebase sampling registers is controlled by (i.e., triggered by) the CPU. One or more additional sets of registers are triggered by external events (e.g., one-pulse-per-second external signal). External events include signals generated by external functions as well as signals generated by internal functions such as digital frequency synthesizers or analog PLLs.

Message-Related Operations

This section describes the hardware and software operations required to generate or respond to various PTP messages.

NOTE In the discussions that follow, a number of variables are used whose names begin with `cpuSampled`. These variables represent CPU-accessible hardware registers into which sampled time values are deposited in response to specific events. These variables are not part of any PTP message structure.

NOTE When “timestamps” is used as a verb, it refers to the sampling of time values in response to a PTP event message. It does not imply that the corresponding message is updated in any way by the timestamping action. However, subsequent action taken by the hardware may, indeed, modify the corresponding packet as a consequence of the timestamping action. These details are spelled out as needed.

PTP Message Prefix

All PTP messages share a common prefix and only differ in their suffixes. The prefix occupies the first 32 bytes of each PTP message and its structure is shown Figure 107.

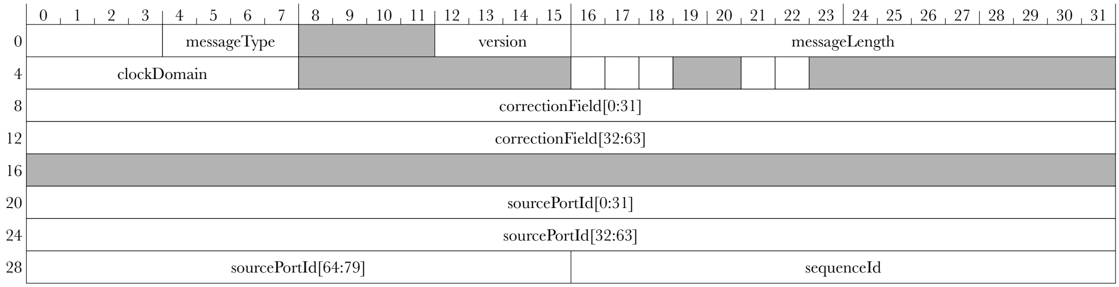


Figure 107 PTP Message Prefix Structure Diagram

The fields that make up the PTP message prefix are defined in Table 37.

Table 37 PTP Message Prefix Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`transportSpecific` (transportSpecific)	4 bits	0.0	May be used by lower-layer protocols.

`messageType`
(messageType)
	4 bits	0.4	Identifies the message type.
			This field is enumerated as follows.
			`0 = Sync`
			`1 = DelayRequest`
			`2 = PeerDelayRequest`
			`3 = PeerDelayResponse`
			`8 = FollowUp`
			`9 = DelayResponse`
			`10 = PeerDelayResponseFollowUp`
			`11 = Announce`
			`12 = Signaling`
			`13 = Management`

`version` (versionPTP)	4 bits	0.12	Indicates the version of PTP running the originating node.
`messageLength` (messageLength)	16 bits	0.16	The total length of the PTP message as measured in bytes.
`clockDomain` (domainNumber)	8 bits	4.0	Indicates the clock domain to which the message belongs.
`alternateMaster` (alternateMasterFlag)	1 bit	4.16	Set to `1` if the originating port is not a master port.
`twoStep` (twoStepFlag)	1 bit	4.17	Indicates one-step vs. two-step operation. Set to `1` for two step operation. Set to `0` for one-step operation.
`unicast` (unicastFlag)	1 bit	4.18	Set to `1` to indicate that the transport layer protocol address for this message is a unicast address, otherwise it is a multicast address.
`profileSpecific1` (PTP profile Specific 1)	1 bit	4.21	As defined by an alternate PTP profile.
`profileSpecific2` (PTP profile Specific 2)	1 bit	4.22	As defined by an alternate PTP profile.
`correctionField` (correctionField)	64 bits	8.0	A time interval used for correction purposes. This interval is measured in nanoseconds and then multiplied by 2^{16}. For example, 2.5 nanoseconds is represented as `0x0000_0000_0002_8000`.
`sourcePortId` (sourcePortIdentity)	80 bits	20.0	A globally-unique source port identity value.

Field Name (std. name)	Width	Offset (B.b)	Definition
`sequenceId` (sequenceId)	16 bits	28.16	Uniquely identifies an event message within a clock domain. `sequenceId` is incremented monotonically using a unique sequence for each combination of clock domain and message type. Note: `PeerDelayResponse`, `FollowUp`, `DelayResponse` and `PeerDelayResponseFollowUp` all share a common `sequenceId` pool.

Sync and FollowUp

The structure of a `Sync` message is shown in Figure 108 and defined in defined Table 38.

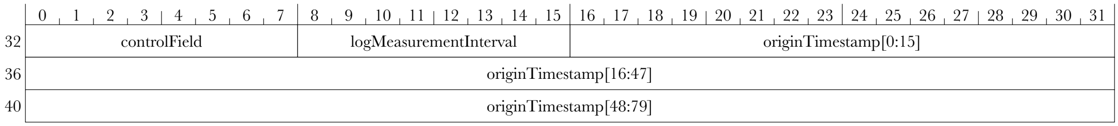


Figure 108 Sync Message Structure Diagram

Table 38 Sync Message Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`controlField` (controlField)	8 bits	32.0	This field is provided for compatibility with hardware designed to conform to version 1 of the PTP standard.
`logMessageInterval` (logMessageInterval)	8 bits	32.8	Indicates the rate at which the current message type is to be generated.
`originTimestamp` (originTimestamp)	80 bits	32.16	The time at which the `Sync` message was transmitted.

The structure of a `FollowUp` message is shown in Figure 109.

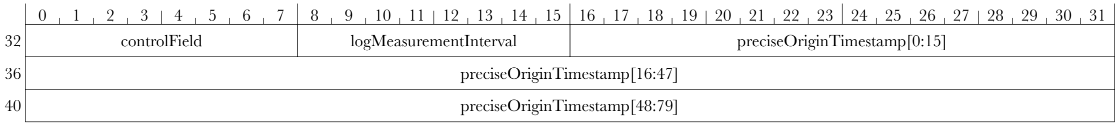


Figure 109 FollowUp Message Structure Diagram

And the fields of the FollowUp message are defined in Table 39.

Table 39 FollowUp Message Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
controlField (controlField)	8 bits	32.0	This field is provided for compatibility with hardware designed to conform to version 1 of the PTP standard.
logMessageInterval (logMessageInterval)	8 bits	32.8	Indicates the rate at which the current message type is to be generated.
preciseOriginTimestamp (preciseOriginTimestamp)	80 bits	32.16	The time at which the associated Sync message was transmitted.

Sync Transmission by Ordinary and Boundary Clocks

Sync messages originate in ordinary and boundary clocks and are terminated by ordinary and boundary clocks. The local CPU of the master ordinary or boundary clock is responsible for initiating the transmission of Sync messages. Nominally, these messages are transmitted 10 times per second and are multicast to all of the slave ordinary and boundary clocks. However, faster or slower message rates and unicast distribution are also possible.

For a one-step clock (i.e., twoStep is set to 0), the CPU must fill in originTimestamp with an estimate of the time of day of the transmission the Sync message. This estimate must be within ±1 second of the actual transmission time. During transmission, the hardware fills in correctionField such that the sum of originTimestamp and correctionField equals the actual time of day of the transmission of the message.

To accomplish this in practice, the CPU triggers a simultaneous sampling of the forwarding system's local timeOfDaySeconds, timeOfDayNanoseconds and intervalNanoseconds counter values. The CPU sets originTimestamp equal to a concatenation of the sampled timeOfDaySeconds and timeOfDayNanoseconds values, and correctionField is set equal to intervalNanoseconds. During transmission, the Ethernet media access controller timestamps the message (i.e., by sampling the intervalNanoseconds counter) and updates the Sync message's correctionField value as follows:

```
correctionField = intervalNanoseconds - correctionField
```

Thus, as transmitted, the sum of the Sync message's originTimestamp and correctionField values equal the message's actual transmission time.

For a two-step clock (i.e., twoStep is set to 1), the CPU must fill in originTimestamp with an estimate of the actual transmission time of the `Sync` message. This estimate must be within ±1 second of the actual transmission time. To accomplish this, the CPU triggers a simultaneous sampling of the forwarding system's local `timeOfDaySeconds`, `timeOfDayNanoseconds` and `intervalNanoseconds` counter values and storing them in `cpuSampledTimeOfDaySeconds`, `cpuSampledTimeOfDayNanoseconds` and `cpuSampledIntervalNanoseconds`, respectively.

The CPU sets its `Sync` message's `originTimestamp` equal to the concatenation of previously-sampled `cpuSampledTimeOfDaySeconds` and `cpuSampledTimeOfDayNanoseconds`, and sets `correctionField` to zero. The CPU then initiates the transmission of the `Sync` message.

The Ethernet media access control hardware timestamps the `Sync` message during transmission by sampling the `intervalNanoseconds` counter and delivering that value to the CPU where it is stored in `transmitIntervalNanoseconds`. The hardware also delivers the `Sync` message's transmit port ID, `Sync.clockDomain` and `Sync.sequenceId` values to the CPU to correlate `transmitIntervalNanoseconds` with the just-transmitted `Sync` message. The CPU composes a `FollowUp` message by setting `FollowUp.preciseOriginTimestamp` to the same value placed in `Sync.originTimestamp` and setting `FollowUp.correctionField` as follows:

```
correctionField = transmitIntervalNanoseconds – cpuSampledIntervalNanoseconds
```

Thus, the sum of `FollowUp.preciseOriginTimestamp` and `FollowUp.correctionField` equals the `Sync` message's actual transmission time.

Sync Reception and Transmission by a Transparent Clock

`Sync` messages are forwarded by transparent clocks toward their addressed destinations. However, adjustments are made to the messages (or their follow-up messages) as they are being forwarded in order to compensate for the queuing delay imposed by the transparent clock (i.e., residency time).

For a one-step transparent clock, the residency time of the `Sync` message must be added to its `correctionField`. To accomplish this, the hardware timestamps the `Sync` message upon reception by sampling the receive `intervalNanoseconds` counter and immediately updates `correctionField` as follows:

```
correctionField =  correctionField – receiveIntervalNanoseconds + pathDelay
```

> NOTE If the clock is an end-to-end transparent clock, then `pathDelay` is set to zero.

The modified `Sync` message is then forwarded to the appropriate transmit interface(s). During transmission, the hardware timestamps the `Sync` message by sampling the transmit `intervalNanoseconds` counter and updates `correctionField` as follows:

```
correctionField = correctionField + transmitIntervalNanoseconds
```

Thus, correctionField is updated to reflect the residency time across the transparent clock and, if the clock is a peer-to-peer transparent clock, corrected for the peer path delay as well.

For a two-step transparent clock, the Sync message is forwarded normally and without modification. The message's hardware-generated timestamp values receiveIntervalNanoseconds and transmitIntervalNanoseconds (along with its transmit port ID, Sync.clockDomain and Sync.sequenceId for correlation purposes) are delivered to the CPU. Upon receipt of the associated FollowUp, its correctionField value is updated as follows:

```
FollowUp.correctionField = FollowUp.correctionField +
  (transmitIntervalNanoseconds – receiveIntervalNanoseconds) + pathDelay
```

Thus, FollowUp.correctionField is updated to reflect the residence time of the associated Sync message. The FollowUp message is forwarded normally after it has been updated.

Sync Reception by Ordinary and Boundary Clocks

Ordinary and boundary clocks use Sync messages to align their local clocks to the grand master by measuring the absolute value and the rate of change of the offset between the two. Received Sync messages are not forwarded by ordinary and boundary clocks.

For a one-step clock, the reception of a Sync message triggers a timestamp event for the received message. This timestamp value (receiveIntervalNanoseconds) is delivered to the CPU along with the receive packet's interface ID, Sync.clockDomain and Sync.sequenceId. The Sync message itself is also forwarded to the CPU as a normal packet reception.

In response, the CPU triggers the simultaneous sampling of timeOfDaySeconds, timeOfDayNanoseconds and intervalNanoseconds from the local timebase clock in order to compute the time of day of the reception of the Sync message. These sampled values are stored in variables with the prefix "cpuSampled...". By calculating the difference between the cpuSampledIntervalNanoseconds and the Sync message's hardware-timestamped receiveIntervalNanoseconds value, the CPU can determine the time of day that the Sync was received (based on the local slave clock's understanding of time of day). This is done as follows:

1. Figure out how much time has transpired between the reception of the Sync message and the CPU-driven sampling of the slave device's various time counters:

```
sampleDelayNanoseconds = cpuSampledIntervalNanoseconds − receiveIntervalNanoseconds
```

2. Subtract `sampleDelayNanoseconds` from the CPU-sampled time-of-day values (`cpuSampledTimeOfDaySeconds` and `cpuSampledTimeOfDayNanoseconds`) to arrive at the time of day (according to the slave device's time counters) of the arrival of the `Sync` message. Because the interval counters are modulo 2^{48} while the time of day counters are a two-part counting system counting seconds and nanoseconds (modulo 10^9) separately, a little extra math is required as described below.
   - If `sampleDelayNanoseconds` is less than 10^9, then: Subtract `sampleDelayNanoseconds` from `cpuSampledTimeOfDayNanoseconds`. If this subtraction results in a carry, subtract 1 from `cpuSampledTimeOfDaySeconds`.
   - Else, if `sampleDelayNanoseconds` is greater than or equal to 10^9, then: Subtract 10^9 from `sampleDelayNanoseconds` and subtract 1 from `cpuSampledTimeOfDaySeconds`. Now, subtract the newly-updated `sampleDelayNanoseconds` from `cpuSampledTimeOfDayNanoseconds`. If this subtraction results in a carry, subtract an additional 1 from `cpuSampledTimeOfDaySeconds`.
3. At this point, `cpuSampledTimeOfDaySeconds` and `cpuSampledTimeOfDayNanoseconds` reflect the time of day of the reception of the `Sync` message according to the slave's clock. The offset between the slave and master clocks is computed as follows:

```
offsetFromMaster =  {cpuSampledTimeOfDaySeconds, cpuSampledTimeOfDayNanoseconds} −
  originTimestamp − pathDelay − correctionField[13]
```

This offset is then applied to the slave clock in order to align it with the master clock.

For a two-step clock, the process is roughly the same as described above for the one-step clock. The only significant difference is that the `Sync` message's `originTimestamp` value is delivered by a `FollowUp` message rather than in the `Sync` message itself. Therefore, the clock offset computation must be deferred until the reception of the `FollowUp` message, and both the `Sync` and `FollowUp` messages must be delivered to the CPU along with the `Sync` message's hardware-generated receive timestamp value.

[13] The 80-bit values in this equation (offsetFromMaster and originTimestamp) are actually two-part numbers (seconds and modulo 10^9 nanoseconds). Carries and borrows between nanoseconds and seconds must be handled accordingly.

DelayRequest and DelayResponse

The structure of a `DelayRequest` message is depicted in Figure 110 and Table 40.

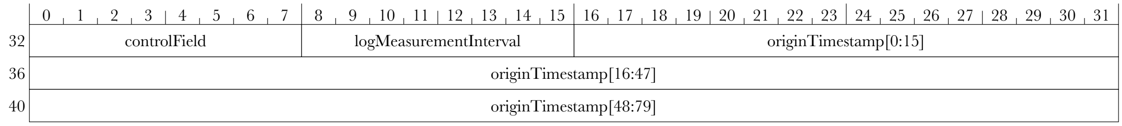


Figure 110 DelayRequest Message Structure Diagram

Table 40 DelayRequest Message Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`controlField` (controlField)	8 bits	32.0	This field is provided for compatibility with hardware designed to conform to version 1 of the PTP standard.
`logMessageInterval` (logMessageInterval)	8 bits	32.8	Indicates the rate at which the current message type is to be generated.
`originTimestamp` (originTimestamp)	80 bits	32.16	The time at which the `DelayRequest` message was transmitted.

The structure of a `DelayResponse` message is shown in Figure 111 and defined in Table 41.

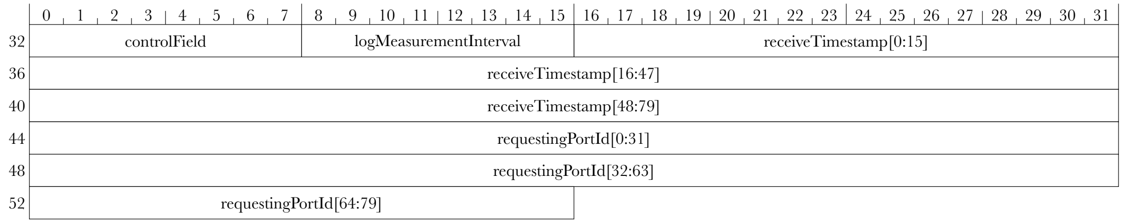


Figure 111 DelayResponse Message Structure Diagram

Table 41 DelayResponse Message Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`controlField` (controlField)	8 bits	32.0	This field is provided for compatibility with hardware designed to conform to version 1 of the PTP standard.
`logMessageInterval` (logMessageInterval)	8 bits	32.8	Indicates the rate at which the current message type is to be generated.

Field Name (std. name)	Width	Offset (B.b)	Definition
`receiveTimestamp` (receiveTimestamp)	80 bits	32.16	The time at which the `DelayRequest` message was received.
`requestingPortId` (requestingPortIdentity)	80 bits	44.0	The identity of the port that issued the `DelayRequest` message.

DelayRequest Transmission by Ordinary and Boundary Clocks

`DelayRequest` messages are transmitted periodically by ordinary and boundary slave clocks toward their master clock in order to measure the round trip delay between the two. Dividing this round trip time by two provides the slave system with the `pathDelay` parameter.

To generate a `DelayRequest` message, the slave clock sets `DelayRequest.correctionField` value to zero and `DelayRequest.originTimestamp` either to zero or to an estimate of the message's transmit time that is within ±1 second of its actual transmission time.

Upon transmission, the hardware timestamps the message and returns the `transmitIntervalNanoseconds` value to the CPU along with the message's transmit interface ID, `clockDomain` and `sequenceId` for correlation purposes. The CPU saves `transmitIntervalNanoseconds` in anticipation of the reception of a `DelayResponse` message.

DelayRequest/DelayResponse Reception and Transmission by a Transparent Clock

`DelayRequest` messages are forwarded by transparent clocks toward their addressed destinations. However, adjustments are made to the messages as they are being forwarded in order to compensate for the queuing delay imposed by the transparent clock.

For a one-step transparent clock, the residency time of the `DelayRequest` message must be added to the `DelayRequest.correctionField` value. To accomplish this, the hardware timestamps the `DelayRequest` message upon reception and updates its `correctionField` as follows:

```
correctionField = correctionField – receiveIntervalNanoseconds + pathDelay
```

NOTE For end-to-end delay measurements, `pathDelay` is equal to `0`.

The modified `DelayRequest` message is then forwarded to the appropriate transmit interface. During transmission, the hardware timestamps the `DelayRequest` message and updates its `correctionField` as follows:

```
correctionField = correctionField + transmitIntervalNanoseconds
```

Thus correctionField is updated to reflect the residency time across the transparent clock and, if the clock is a peer-to-peer transparent clock, corrected for the peer path delay as well.

For a two-step transparent clock, the DelayRequest message is forwarded normally. The message's hardware-based receiveIntervalNanoseconds and transmitIntervalNanoseconds timestamp values (along with the receive and transmit interface IDs an the clockDomain and sequenceId values from the DelayRequest message for correlation purposes) are delivered to the CPU. Upon receipt of the associated DelayResponse message (traveling in the opposite direction), its correctionField value is updated by the CPU as follows:

```
correctionField = correctionField + (transmitIntervalNanoseconds –
  receiveIntervalNanoseconds)
```

Thus, The DelayResponse message's correctionField is updated to reflect the residence time of the associated DelayRequest message. The DelayResponse message is forwarded normally after it has been updated.

DelayRequest Reception and DelayResponse Transmission by Ordinary and Boundary Clocks

When a master receives a DelayRequest, its hardware detects the event message and generates receiveIntervalNanoseconds. This value (along with the receive interface ID, clockDomain and sequenceId for correlation purposes) is delivered to the CPU. The DelayRequest message itself is delivered to the CPU as a normal receive packet.

Upon receiving the DelayRequest message from the hardware, the CPU triggers the simultaneous sampling of the timeOfDaySeconds, timeOfDayNanoseconds and intervalNanoseconds values. The difference between receiveIntervalNanoseconds and cpuSampledIntervalNanoseconds is subtracted from cpuSampledTimeOfDaySeconds and cpuSampledTimeOfDayNanoseconds values in order to calculate the time of day when the DelayRequest message was received.

The CPU generates a DelayResponse message by setting its correctionField equal to the received DelayRequest message's correctionField value and sets its receiveTimestamp to the corrected receive time of day (described above).

The DelayResponse message is then transmitted to the slave that issued the DelayRequest message.

PeerDelayRequest, PeerDelayResponse, and PeerDelayResponseFollowUp

The structure of a `PeerDelayRequest` message is shown in Figure 112.

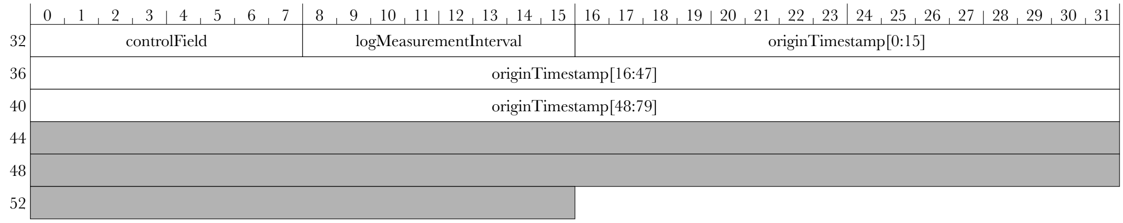


Figure 112 *PeerDelayRequest Message Structure Diagram*

The fields of a `PeerDelayRequest` message are defined in Table 42.

Table 42 *PeerDelayRequest Message Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`controlField` (controlField)	8 bits	32.0	This field is provided for compatibility with hardware designed to conform to version 1 of the PTP standard.
`logMessageInterval` (logMessageInterval)	8 bits	32.8	Indicates the rate at which the current message type is to be generated.
`originTimestamp` (originTimestamp)	80 bits	32.16	The time at which the `PeerDelayRequest` message was transmitted.

The structure of a `PeerDelayResponse` message is shown in Figure 113.

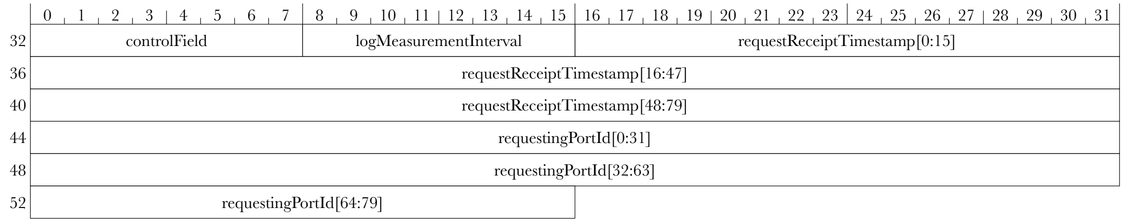


Figure 113 *PeerDelayResponse Message Structure Diagram*

The fields of a `PeerDelayResponse` message are defined in Table 43.

Table 43 PeerDelayResponse Message Field Definitions

Field Name (std. Name)	Width	Offset (B.b)	Definition
`controlField` (controlField)	8 bits	32.0	This field is provided for compatibility with hardware designed to conform to version 1 of the PTP standard.
`logMessageInterval` (logMessageInterval)	8 bits	32.8	Indicates the rate at which the current message type is to be generated.
`requestReceiptTimestamp` (requestReceiptTimestamp)	80 bits	32.16	The time at which the `PeerDelayRequest` message was transmitted.
`requestingPortId` (requestingPortIdentity)	80 bits	44.0	The identity of the port that issued the `PeerDelayRequest` message.

The structure of a `PeerDelayResponseFollowUp` message is shown in Figure 114.

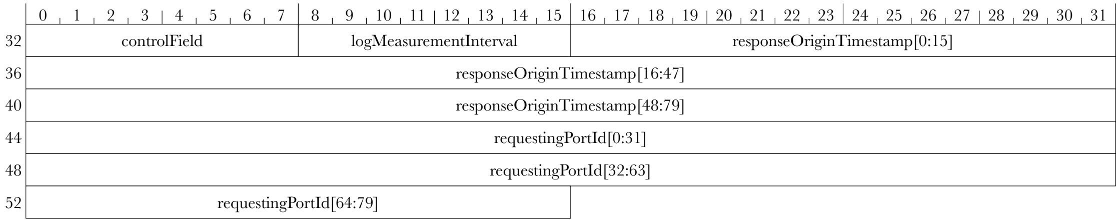


Figure 114 PeerDelayResponseFollowUp Message Structure Diagram

The fields of a `PeerDelayResponseFollowUp` message are defined in Table 44.

Table 44 PeerDelayResponseFollowUp Message Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`controlField` (controlField)	8 bits	32.0	This field is provided for compatibility with hardware designed to conform to version 1 of the PTP standard.
`logMessageInterval` (logMessageInterval)	8 bits	32.8	Indicates the rate at which the current message type is to be generated.
`responseOriginTimestamp` (responseOriginTimestamp)	80 bits	32.16	The time at which the `PeerDelayResponse` message was transmitted.
`requestingPortId` (requestingPortIdentity)	80 bits	44.0	The identity of the port that issued the `PeerDelayRequest` message.

PeerDelayRequest Transmission

Any type of clock (ordinary, boundary, transparent) that has a port that is configured for peer-to-peer delay measurement transmits periodic `PeerDelayRequest` messages. These messages are link-local and are not forwarded by the port's link partner.

The CPU generates a `PeerDelayRequest` message by setting its `originTimestamp` and `correctionField` values to zero. Upon transmission of the `PeerDelayRequest` message, the hardware generates a `transmitIntervalNanoseconds` value, which is delivered to the CPU along with the `PeerDelayRequest` message's transmit interface ID, `clockDomain` and `sequenceId` values (for correlation purposes). The `transmitIntervalNanoseconds` value is stored by the CPU in anticipation of the eventual reception of a `PeerDelayResponse` message.

PeerDelayRequest Reception and PeerDelayResponse/ PeerDelayResponseFollowup Transmission

Upon reception of a `PeerDelayRequest` message, the hardware generates a `receiveIntervalNanoseconds` value. This value is delivered to the CPU (along with the packet's receive interface ID and the `PeerDelayRequest` message's `clockDomain` and `sequenceId` values for correlation purposes). The `PeerDelayRequest` message itself is also delivered to the CPU via a separate path.

In response to the `PeerDelayRequest` message, the CPU generates a `PeerDelayResponse` message.

If the clock is a one-step clock, the `PeerDelayResponse` message is generated by setting its `requestReceiptTimestamp` to zero and setting `correctionField` to the two's complement of `receiveIntervalNanoseconds`.

During transmission, the hardware generates a `transmitIntervalNanoseconds` value and then updates `correctionField` as follows:

```
correctionField = correctionField + transmitIntervalNanoseconds
```

Thus, `PeerDelayResponse` message's `correctionField` contains a value that represents the turnaround time from the receipt of the `PeerDelayRequest` to the transmission of its associated `PeerDelayResponse`.

If the clock is a two-step clock, the `PeerDelayResponse` message is generated by setting both `requestReceiptTimestamp` and `correctionField` to zero.

During the transmission of `PeerDelayResponse`, the hardware generates `transmitIntervalNanoseconds` and delivers that value along with the message's transmit interface ID, `clockDomain` and `sequenceId` to the CPU. The CPU, in turn, generates a `PeerDelayResponseFollowUp` message by setting its `responseOriginTimestamp` to zero and `correctionField`[63:0] to `transmitIntervalNanoseconds`

– receiveIntervalNanoseconds. The PeerDelayResponseFollowUp message is then transmitted to the requestor.

PeerDelayResponse Reception

Upon reception of a PeerDelayResponse message, the hardware generates a receiveIntervalNanoseconds value and delivers it to the CPU along with the message receive interface ID, clockDomain and sequenceId.

If the clock is a one-step clock, the path delay is computed as follows:

```
pathDelay = ((receiveIntervalNanoseconds – transmitIntervalNanoseconds) –
  correctionField)/2
```

NOTE The transmitIntervalNanoseconds value used here is the value that was stored by the CPU when it transmitted the PeerDelayRequest message.

If the clock is a two-step clock, the path delay is computed using the correctionField value from the PeerDelayResponseFollowUp message instead of from the PeerDelayResponse message.

Accommodating UDP Checksums

The methods described in the preceding sections specify that changes are made to correctionField values. This means that Udp.checksum must be updated if a PTP event message is encapsulated by IP/UDP. The challenge that is faced by high performance Ethernet MACs is that the Udp.checksum value is likely to have been transmitted (or at least shifted toward the physical medium so far that it can no longer be modified) before the update to correctionField is performed. One method to account for this is to ensure that at least 2 pad bytes follow the PTP message. These pad bytes are included in the checksum computation, so any change made to correctionField can be balanced out—from a checksum perspective—by making a complementary update to these pad bytes at the end of the packet. Of course, this scheme only works if the original PTP event message was generated with these pad bytes in place.

15 OAM

As packet-based networking is being adopted more and more as a replacement for traditional synchronous, circuit-switched optical networks in telecom applications, certain functional aspects of these legacy networks must be grafted onto the packet-based standards in order to take advantage of their simplicity and efficiency without sacrificing the robustness and maintainability of the legacy networks. The fault detection and maintenance functions of the legacy synchronous networks are collected into a suite of functions known as Operations, Administration and Maintenance, or OAM. This chapter provides an overview of OAM and describes the requirements for implementing these functions in a hardware-centric environment (as opposed to a software or micro-code environment).

OAM strives to test and monitor a network to help ensure that its fundamental operating characteristics are within specified tolerances. These operating characteristics include:

- basic connectivity
- forwarding delays
- packet loss
- user data throughput

A series of specialized messages, protocols and hardware features combine to test these characteristics.

OAM functions have been defined for Ethernet and MPLS. Though they both serve roughly the same purpose, there are enough differences in the details that it is beneficial to describe them separately. Let’s start with Ethernet OAM.

Ethernet OAM Architecture

Ethernet OAM is defined by two overlapping international standards: ITU-T Y.1731 and IEEE 802.1ag. The ITU standard is a superset of the IEEE standard. Where they do overlap, the message formats and algorithms are compatible, but the terminology is slightly different. In this chapter, I have chosen to present the ITU version of Ethernet OAM.

Ethernet OAM is intended to operate over an extended Layer 2 network in which certain elements are actually services provided by a variety of independent organizations. For example, a single customer may operate a series of distributed networks that it interconnects at Layer 2 using the services provided by one or more network operators. This customer must be able to test the connections between its sites without having to involve the network operators and without having to know anything about the operator's internal network topology. Similarly, each network operator must be able to test the end-to-end connectivity for their portion of the network, as well as the links to the other operators with which it has service exchange agreements.

Overview

The logical and physical connections in OAM are known as maintenance entities (MEs). Essentially, a maintenance entity is a relationship between two endpoints that require management. A maintenance entity group (MEG) is a group of maintenance entities that exist within the same administrative boundary, have the same maintenance entity group level (defined below) and belong to the same point-to-point or multipoint Ethernet network.

A maintenance entity group endpoint (MEP) represents the point at which OAM packets may originate or terminate. Maintenance entity group endpoints are further differentiated by the functional direction that they are facing (i.e., the direction that packets are launched and received) — either toward a network connection (down MEP endpoints) or toward the switching plane of a bridge (up MEP endpoints). The up vs. down naming convention is explained more completely later on in this chapter.

A maintenance entity group intermediate point (MIP) may also be configured. OAM packets do not originate or terminate at a maintenance entity group intermediate point, but a maintenance entity group intermediate point may react to certain OAM packets. The purpose of a maintenance entity group intermediate point is to provide additional visibility into a network without necessarily exposing the entire network topology.

Maintenance entity groups are often nested. The nesting of maintenance entity groups is beneficial when, say, a customer's distributed networks are interconnected by an independent network operator. In this case, it is required for a customer to be able to test the connections across the independent operators' networks while allowing those networks to be opaque. At the same time, the network operator must be able to test its own network using the same physical and virtual connections that the customer uses without interfering with customer traffic or exposing the customer to the operator's tests. The nesting of maintenance entity groups is accomplished by assigning maintenance entity group level values to the various maintenance entity groups.

A simple network case study is presented in Figure 115.

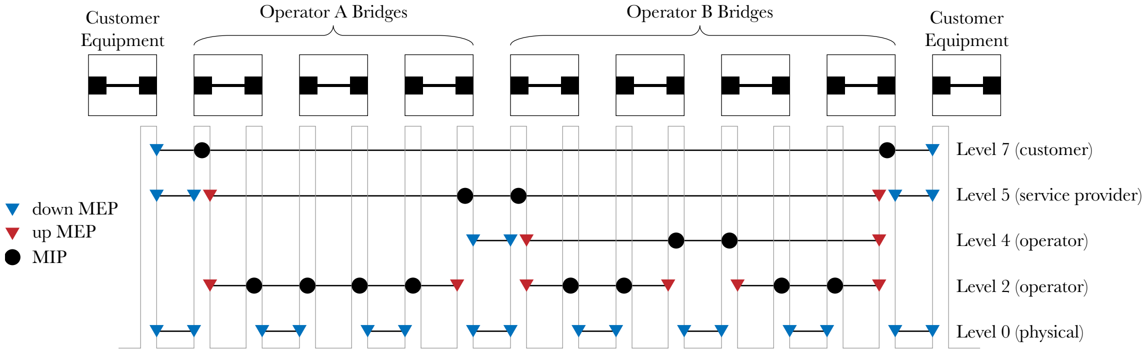


Figure 115 *Ethernet OAM Hierarchies and Service Access Points*

Figure 115 depicts a pair of bridges operated by a single customer that are connected to one another via a set of intermediate bridges that are operated by two independent network operators. Each bridge is depicted as a box enclosing two black squares: one square on each of their vertical edges. These black squares represent the per-port functionality of the bridge and the line between the squares is the bridge's packet filtering/forwarding function.

The thin line that serpentines up and down as it progresses across the bottom of Figure 115 represents the logical flow of packets through the various bridges. Notice that for each port function, there is a pair of vertical segments of this logical packet flow. These vertical segments are on either side of the port's topology enforcement function (i.e., spanning tree filtering). This means that OAM packets that are sourced or sunk by a port/VLAN are not subject to spanning tree blocking associated with that port/VLAN. On the other hand, OAM packets that are forwarded by a bridge are subject to spanning tree blocking.

The horizontal lines with triangles and, optionally, circles attached represent the various maintenance entities within various maintenance entity groups, arranged so that one or more maintenance entity groups is nested within a maintenance entity group of a higher maintenance entity group level number. The triangles in Figure 115 represent maintenance entity group endpoints. The circles represent maintenance entity group intermediate points.

Maintenance entity group endpoints may be instantiated either on the port side of the spanning tree state enforcement function or the filtering/forwarding side. The maintenance entity group endpoints on the port side are known as "down MEPs" and are depicted as blue triangles. Down MEPs send their generated OAM packets toward the port and receive their OAM packets from the port. Up MEPs (depicted as red triangles), on the other hand, send their generated OAM packets toward the bridge's filtering/forwarding function and receive them from the same. Maintenance entity group intermediate points straddle the dividing line formed by the spanning tree enforcement function. Several examples of maintenance entity groups terminated by down and up MEP endpoints are depicted in Figure 115.

The maintenance entity group at level 7 in the example is representative of a maintenance entity group that a customer may establish to test and monitor its connections through the third-party independent networks. The down MEP endpoints (blue triangles) show that the maintenance entity group spans from the port on the customer's bridge on one side of the extended network to a similar customer port on the other side. The OAM packets sent from one down MEP endpoint to the other do not actually pass through any of the bridging functions within the customer's networking equipment. Transmit OAM packets are generated just ahead of the Ethernet media access control (MAC) function and are terminated just beyond the MAC function by the receiving down MEP endpoint.

The maintenance entity group intermediate points that are established at level 7 by a cooperative agreement between the customer and the two network operators provide the customer with a degree of visibility into the network to help isolate faults more quickly.

Up MEPs face toward the bridges' forwarding function. This means, for example, that the service provider can establish an up MEP endpoint on one end of an operator's network and on the far end of another operator's network (red triangles).

OAM packets generated by one of these up MEPs must go through the bridge filtering/forwarding process of the bridge in which the up MEP endpoint is configured. This means that these OAM packets experience exactly the same forwarding and handling (though perhaps different QoS levels) as the customer's network packets. Specifically, the OAM packets and the customer packets use the same VLAN tagging.

Maintenance entity groups may be configured to reside within a VLAN. Therefore, it is possible for thousands of maintenance entity groups to be configured across a single set of physical connections. When maintenance entity group endpoints are configured within a VLAN, the expectation is that these are the same VLANs that carry customer, service provider or operator traffic across their respective networks. In other words, VLANs are not used to separate OAM packets from customer packets.

The physical connections between bridges are an implied maintenance entity group level (at level 0). The physical layer maintenance entity group endpoints may only be down MEP endpoints and are intended for point-to-point link testing and monitoring.

Maintenance Entity Group Levels and End Point Details

As was introduced in the section above, levels are used to allow the establishment of nested maintenance domains. This section provides more details and examples of the implications of maintenance entity group levels.

Within a single VLAN, multiple maintenance entity group endpoints and a single maintenance entity group intermediate point may be configured on a bridge port. The maintenance entity group endpoints are arranged in such a way that their maintenance entity group levels increase as their logical distance from the spanning tree port filtering function decreases. Refer to Figure 116.

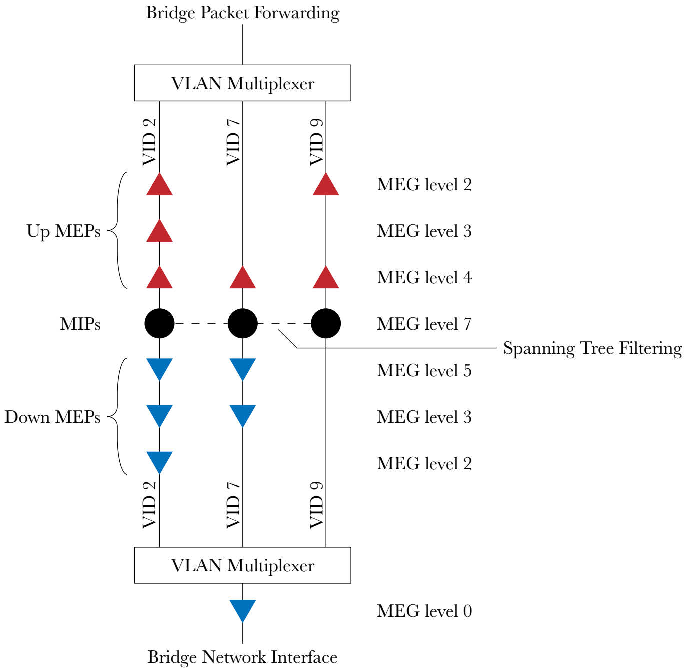


Figure 116 *Maintenance Point Placement*

The "pointy" end of a triangle is the active side of a maintenance entity group endpoint while the opposite side is considered the passive side. If a packet arrives at a maintenance entity group endpoint via its passive side, no action is taken and the packet passes through the maintenance entity group endpoint. A packet's arrival at a maintenance entity group endpoint's active side triggers an examination of the packet and, potentially, some kind of action is taken.

Referring to Figure 116, let's consider an OAM packet arriving at a bridge's port with a VLAN ID of 2 and a maintenance entity group level of 3. This packet is first considered by the down MEP at maintenance entity group level 0. This maintenance entity group endpoint is a physical port maintenance entity group endpoint and is VLAN unaware. Because the maintenance entity group level of the OAM packet is greater than that of the maintenance entity group endpoint, the maintenance entity group endpoint passes it along. The VLAN multiplexer function steers the packet to the appropriate column of maintenance entity group endpoints where it next encounters a down MEP at maintenance entity group level 2. Here also the maintenance entity group level of the packet is greater than that of the maintenance entity group endpoint so it is passed along to the next maintenance entity group endpoint.

The next down MEP that is encountered has a maintenance entity group level that matches that of the OAM packet, so the packet is terminated at this maintenance entity group endpoint and the appropriate processing is carried out. This down MEP may generate an OAM packet in reply. The down MEP can only direct its packets toward the port, so the packet encounters the down MEP at maintenance entity group level 2. Because this packet entered the passive side of the maintenance entity group endpoint, no action is taken and the packet is passed along to the down MEP at maintenance entity group level 0 and to the transmit port.

If an OAM packet is received via the port whose maintenance entity group level exceeds the maintenance entity group level of all of the configured down MEPs in the packet's VLAN, the packet is passed along to the packet forwarding function and is forwarded normally. This scenario occurs when, say, a customer packet (with a high maintenance entity group level) is processed by a network operator's bridge whose maintenance entity group endpoints all have lower maintenance entity group levels. Such a packet must be forwarded normally, making the operator's bridge appear transparent to the transient OAM packet.

Now consider an OAM packet received via the port whose VLAN ID is 7 and whose maintenance entity group level is 2. This packet will pass through the down MEP at maintenance entity group level 0 and will next encounter the down MEP at maintenance entity group level 3 in VLAN 7. When a maintenance entity group endpoint receives an OAM packet whose maintenance entity group level is lower than maintenance entity group level of the MEP, then some kind of configuration or forwarding error has occurred. Such a packet should have been consumed by a preceding maintenance entity group endpoint elsewhere in the network. It is incumbent upon the maintenance entity group endpoint to discard all such packets and indicate that a maintenance entity group level violation has been detected.

Another interesting case is where an OAM packet is processed whose maintenance entity group level falls between the configured maintenance entity group levels for its port and VLAN—i.e., the OAM packet's maintenance entity group level is greater than that of one or more of the configured maintenance entity group endpoints and also less than that of one or more of the configured maintenance entity group endpoints, but is not equal to any of them. Such a packet is discarded because the packet is eventually examined by an active maintenance entity group endpoint interface whose maintenance entity group level is greater than that of the OAM packet.

Generalized OAM Packet Filtering

OAM packets are detected based on their `ethertype` value. The value `0x8902` is reserved for this purpose. If the packet's `ethertype` matches this value, the packet is subjected to further tests. The packet's logical receive port (for ingress packets) or transmit next-hop (for egress packets) is used to retrieve a set of MEP and MIP vectors. For ingress packets (i.e., packet heading toward the Ethernet bridge forwarding entity), a down MEP endpoint vector is used. For egress packets, an up MEP endpoint vector is retrieved. A MIP vector is retrieved for both ingress and egress. Each of these vectors is eight bits wide with one bit set to `1` for each maintenance entity group level at which a MEP or MIP has been configured for the port/VLAN combination.

If an OAM packet has a unicast MAC destination address that matches the MAC address of the bridge port, then the packet is automatically directed to either the OAM processing function or to the local CPU, depending on the OAM packet's OAM opcode. If a unicast OAM packet's destination address does not match the bridge port's MAC address, then the packet is forwarded as a normal data packet.

If an OAM packet has a multicast MAC destination address, then the least significant three bits of that address are used to determine the packet's maintenance entity group level. The packet's maintenance entity group level is then tested against the MEP and MIP vectors that are associated with the packet. If the packet's maintenance entity group level corresponds to a bit that is set to `1` in one of the vectors, then the packet is directed to either a hardware-based OAM processing function or to the local CPU, depending on the OAM packet's opcode. If the packet's maintenance entity group level points to a bit in a maintenance entity group level vector that is set to `0` and that bit is lower-numbered than any of the vectors' `1` bits, then the packet is discarded. If the packet's maintenance entity group level is greater than the highest numbered bit that is set to `1` in the vectors, then the packet is forwarded normally. Figure 117 illustrates this process.

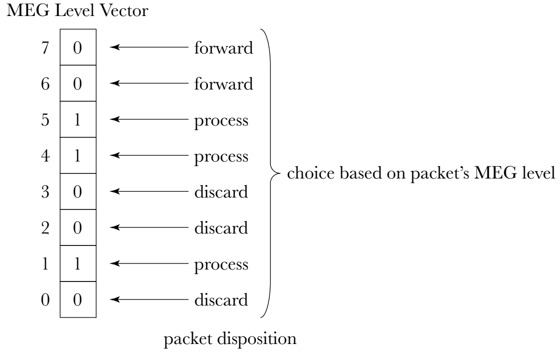


Figure 117 *Maintenance Entity Group Level Vector and Packet Disposition*

Once an OAM packet has been forwarded to the OAM processing function, the packet's opcode, maintenance entity group ID and maintenance entity group endpoint ID are checked to further validate the packet. The details of these operations are described next in *"Ethernet OAM Functions" on page 246*.

OAM Packet Addressing

OAM packets may either use a reserved multicast address or they may use the unicast MAC address of the MEP to which they are directed. There are two classes of multicast addresses reserved for Ethernet OAM: class 1 and class 2. Class 1 multicast addresses are used in OAM packets that are addressed to all of the maintenance entity group endpoints in a maintenance entity group. Class 2 multicast addresses, on the other hand, are addressed to all maintenance entity group endpoints *and* maintenance entity group intermediate points in a maintenance entity group.

The reserved multicast address is: `01-80-c2-00-00-3x`. The value of x is defined in Table 45.

Table 45 OAM Multicast Destination Address Least-Significant Bits Definition

x[0:3]	Meaning
x[0]	This bit distinguishes class 1 destination addresses from class 2 address as follows: 0 = `class1` 1 = `class2`
x[1:3]	These bits encode the OAM packet's maintenance entity group level value.

Ethernet OAM Functions

Ethernet OAM is a series of functions that use normal Ethernet packets and normal packet forwarding methods to monitor the health of an extended Layer 2 network and to quickly detect a variety of faults. These functions are listed below:

- continuity check
- loopback
- link trace
- alarm indication signal
- remote defect indication
- locked signal

- test signal
- automatic protection switching
- maintenance communications channel
- alarm loss measurement
- packet loss measurement
- delay measurement
- throughput measurement

Of the several functions listed above, only a few benefit significantly from some form of specific hardware-based assist. These are:

- continuity check
- loopback
- packet loss measurement
- delay measurement

These four functions have some common characteristics; they either have a very high packet rate (continuity check, loopback) or require real-time hardware interaction (loss measurement, delay measurement). Since the focus of this book is the hardware implications of implementing OAM, those few OAM functions that benefit from hardware assistance are introduced and described in the following sections. For more information on the remaining OAM functions and the software implications of all of the functions, please refer to ITU-T Y.1731.

Common Ingress/Egress Processing

There are certain processing steps that all OAM packets must go through in order to determine if the packets must be forwarded, discarded or delivered to a hardware-based Ethernet OAM processor module. The packet's receive interface and `vlanId` are used to determine to which maintenance entity group the packet belongs. This maintenance entity group is then represented by a `megIndex` value. The `megIndex` value is, in turn, used to access the maintenance entity group level vectors in the `MegLevelVectors` table. If a check of a packet's `Mac.destinationAddress` (if it's a unicast address) or the maintenance entity group level vectors (in the case of a multicast address) indicates that the packet must be delivered to an Ethernet OAM processor module, the packet is forwarded to that module, accompanied by `megIndex` as metadata.

Continuity Check

The continuity check function operates by detecting the loss of reception of regularly-transmitted continuity check messages (CCMs). These continuity check messages are generated by network nodes that serve as maintenance entity group end points. Continuity check messages are addressed and transmitted to all of the other end points within the same maintenance entity group using either unicast or multicast destination addresses. The continuity check messages are consumed by the maintenance entity group endpoints that define the boundary of the maintenance entity group. Each maintenance entity group endpoint expects to receive these continuity check messages on a regular basis from all of the maintenance entity group endpoints that make up the maintenance entity group (a maintenance entity group endpoint does not expect to receive continuity check messages from itself, though).

The continuity check message format and field definitions are presented in Figure 118 and Table 46.

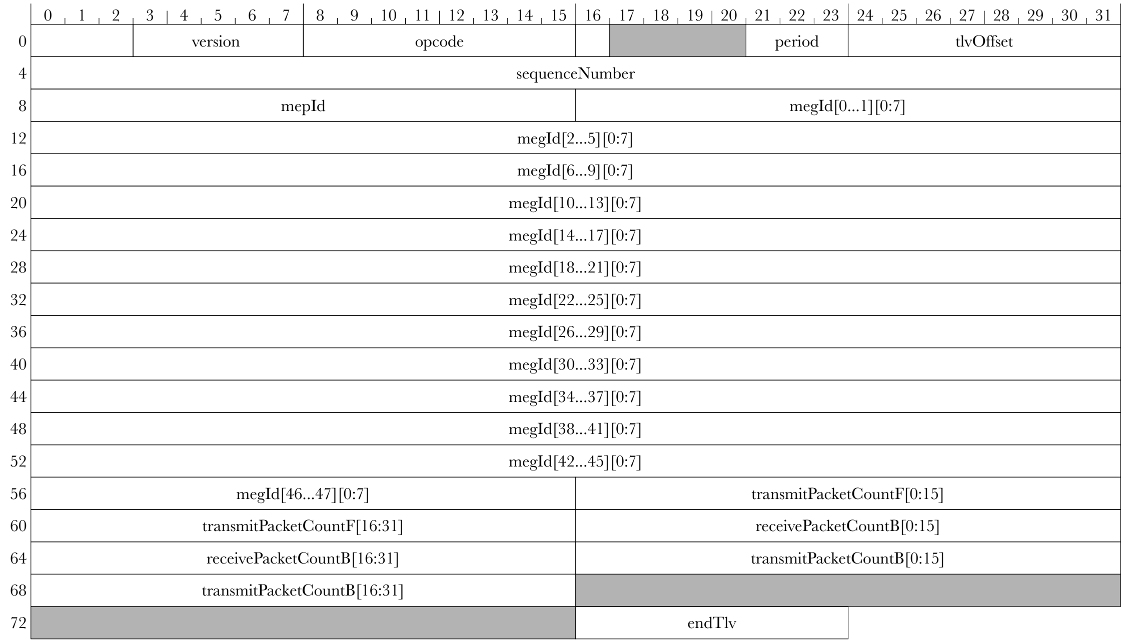


Figure 118 Ethernet OAM Continuity Check Message Structure Diagram

Table 46 *Ethernet OAM Continuity Check Message Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`megLevel` (MEG level)	3 bits	0.0	The continuity check message's maintenance entity group level. Higher maintenance entity group levels span greater distances across the network and are associated with end-to-end connections (i.e., customers). Lower levels are associated with shorter hops (i.e., network operators).
`version` (Version)	4 bits	0.3	Protocol version number. Must be set to `0`.
`opcode` (OpCode)	8 bits	0.8	The message's opcode. For a continuity check message, this field must be set to `1`.
`remoteDefectIndication` (RDI)	1 bit	0.16	Defect indication. This bit is set to `1` to indicate to a remote maintenance entity group endpoint that a fault or defect has been detected in the local maintenance entity group endpoint.
`period` (Period)	3 bits	0.21	Nominal continuity check message transmission period. This field specifies the period between continuity check messages that a remote maintenance entity group endpoint may expect. It is enumerated as follows: 0 = reserved 1 = 3.333 milliseconds 2 = 10 milliseconds 3 = 100 milliseconds 4 = 1 second 5 = 10 seconds 6 = 1 minute 7 = 10 minutes
`tlvOffset` (TLV offset)	8 bits	0.24	Offset of the next TLV. This field specifies the offset of the TLV that follows the current TLV. For continuity check messages, this value must be `70`.
`sequenceNumber` (Sequence number)	32 bits	4.0	The continuity check message sequence number. This field must be set to `0`.

Field Name (std. name)	Width	Offset (B.b)	Definition
`mepId` (maintenance entity group endpoint ID)	16 bits	8.0	Identifies the maintenance entity group endpoint within the context of a maintenance entity group. This value uniquely identifies the maintenance entity group endpoint that is the source of a continuity check message. The most significant three bits of `mepId` are always set to `0`.
`megId` (maintenance entity group ID)	384 bits (48 bytes)	8.16	Globally unique maintenance entity group identifier. All continuity check messages within a maintenance entity group (regardless of the maintenance entity group endpoint IDs of their source or destinations) must use the same `megId` value. Any continuity check message received with a `megId` value that does not match the ID configured for the packet's maintenance entity group (as identified by the packet's port and VLAN identifiers) must be discarded. A discard due to a mismatch indicates that a packet has somehow leaked out of its expected forwarding domain.
`transmitPacketCountF` (TxFCf)	32 bits	56.16	The number of packets transmitted toward a maintenance entity group endpoint addressed by this OAM packet.
`receivePacketCountB` (RxFCb)	32 bits	60.16	The number of packets received by the maintenance entity group endpoint addressed by this OAM packet.
`transmitPacketCountB` (TxFCb)	32 bits	64.16	The `transmitPacketCountF` value from the continuity check message most recently received from a maintenance entity group endpoint's peer maintenance entity group endpoint.
`endTlv`	8 bits	72.16	Marks the end of a sequence of TLVs. This field must be set to `0`.

All of the maintenance entity group endpoints and maintenance entity group intermediate points that make up a maintenance entity group are configured to expect these continuity check messages to arrive at a particular rate. Seven different continuity check message rates are supported. They are defined in Table 47.

Table 47 OAM Continuity Check Message Transmission Rates

Period	Rate	Intended Application
3.333 milliseconds	300 per second	Protection switching
10 milliseconds	100 per second	Protection switching
100 milliseconds	10 per second	Performance monitoring

Period	Rate	Intended Application
1 second	1 per second	Fault management
10 seconds	6 per minute	Fault management
1 minute	1 per minute	Fault management
10 minutes	6 per hour	Fault management

If the stream of continuity check messages from any maintenance entity group endpoint in a maintenance entity group is interrupted for more than 3.5 times the configured continuity check message period, a fault is indicated to the OAM management plane.

Other faults are detected as well. These faults are described in Table 48.

Table 48 *OAM Fault Detection*

Fault	Description
mismerge	A maintenance entity group endpoint detects a mismerge when a continuity check message packet with a correct maintenance entity group level is received that has an incorrect maintenance entity group ID. This indicates that packets from a different service instance (i.e., VLAN, etc.) have leaked into the maintenance entity group endpoint's service instance.
unexpected maintenance entity group endpoint	A maintenance entity group endpoint detects an unexpected maintenance entity group endpoint fault when it receives a continuity check message packet with a correct maintenance entity group level, and a correct maintenance entity group ID but an unexpected (or "unknown") maintenance entity group endpoint ID (including the maintenance entity group endpoint's own maintenance entity group endpoint ID). Each maintenance entity group endpoint maintains a list of peer maintenance entity group endpoint IDs and checks each continuity check message against this list.
unexpected maintenance entity group level	A maintenance entity group endpoint detects an unexpected maintenance entity group level fault when it receives a continuity check message packet with an incorrect maintenance entity group level.
unexpected period	A maintenance entity group endpoint detects an unexpected period fault condition when a continuity check message packet with a correct maintenance entity group level, a known maintenance entity group endpoint ID, and a correct maintenance entity group ID, is received whose `period` field differs from the continuity check message period configured for the maintenance entity group.
remote defect indication	A maintenance entity group endpoint detects a remote defect indication when it receives a continuity check message packet with its `remoteDefectIndication` bit set.

Continuity Check is a one-way protocol. Meaning each maintenance entity group endpoint transmits a stream of continuity check messages to each of its peer maintenance entity group endpoints, but never generates any kind of packet in reaction to the reception of a continuity check message. Maintenance entity group endpoints merely transmit continuity check messages periodically and listen for asynchronously received continuity check messages. Each locally configured maintenance entity group endpoint expects to receive one continuity check message per period from each remote maintenance entity group endpoint in the same maintenance entity group. If the stream of continuity check messages from one or more peer maintenance entity group endpoints is ever interrupted for more than 3.5 times the configured continuity check message period, an alarm indication is generated.

Only those continuity check messages whose maintenance entity group level matches that of the maintenance entity group endpoint are processed. Continuity check messages with a higher maintenance entity group level are forwarded with the expectation that a downstream maintenance entity group endpoint will have a matching maintenance entity group level. Continuity check messages whose maintenance entity group level is lower than that of the maintenance entity group intermediate point or maintenance entity group endpoint indicate that a fault has occurred — i.e., the continuity check message should have been consumed by an upstream maintenance entity group endpoint.

Loopback

OAM loopback packets (loopback messages, or LBMs) are generated at the discretion of the OAM management plane. These packets are generated by the local CPU and are addressed to either a unicast or multicast destination. The destination of a unicast loopback message may be either a peer maintenance entity group endpoint or a maintenance entity group intermediate point. The destination of a multicast loopback message must be a set of one or more peer maintenance entity group endpoints (maintenance entity group intermediate points are not responsive to loopback messages with multicast destination addresses).

The loopback message format and field definitions are presented in Figure 119 and Table 49.

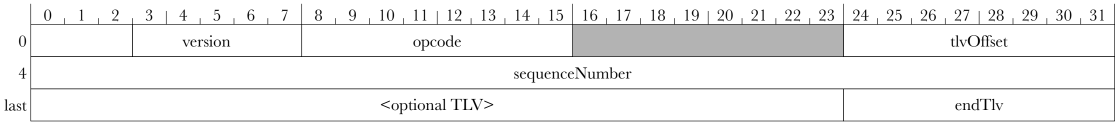


Figure 119 Ethernet OAM Loopback Message Structure Diagram

Table 49 *Ethernet OAM Loopback Message Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`megLevel` (MEG level)	3 bits	0.0	The loopback message's maintenance entity group level. Higher maintenance entity group levels span greater distances across the network and are associated with end-to-end connections (i.e., customers). Lower levels are associated with shorter hops (i.e., network operators).
`version` (Version)	4 bits	0.3	Protocol version number. Must be set to `0`.
`opcode` (OpCode)	8 bits	0.8	The message's opcode. For an loopback message, this field must be set to `3`.
`tlvOffset` (TLV offset)	8	0.24	Offset of the next TLV. This field specifies the offset of the TLV that follows the current TLV. For loopback messages, this value must be `4`.
`sequenceNumber` (Sequence number)	32	4.0	The transaction ID or sequence number. The receiver of this message is expected to copy this value into its loopback reply.
`endTlv`	8	last	Marks the end of a sequence of TLVs. This field must be set to `0`.

The loopback reply format and field definitions are presented Figure 120 and Table 50.

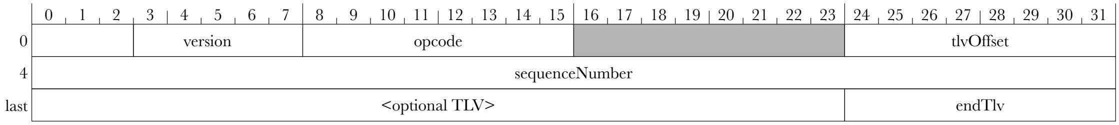


Figure 120 *Ethernet OAM Loopback Reply Structure Diagram*

Table 50 Ethernet OAM Loopback Reply Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`megLevel` (MEG level)	3 bits	0.0	The loopback reply's maintenance entity group level. Higher maintenance entity group levels span greater distances across the network and are associated with end-to-end connections (i.e., customers). Lower levels are associated with shorter hops (i.e., network operators).
`version` (Version)	4 bits	0.3	Protocol version number. Must be set to `0`.
`opcode` (OpCode)	8 bits	0.8	The message's opcode. For an loopback reply, this field must be set to `2`.
`tlvOffset` (TLV offset)	8	0.24	Offset of the next TLV. This field specifies the offset of the TLV that follows the current TLV. For loopback messages, this value must be `4`.
`sequenceNumber` (Sequence number)	32	4.0	The sequence number. This field is copied from the corresponding loopback message.
`endTlv`	8	last	Marks the end of a sequence of TLVs. This field must be set to `0`.

The receiver of a loopback message must formulate a loopback reply (LBR) and send it to the originator of the loopback message. Because it is possible for a large number of remote maintenance entity group endpoints to transmit an unlimited number of loopback messages at very high rates to the local maintenance entity group endpoint, a hardware-based loopback reply mechanism is beneficial.

In the simplest sense, the receiver formulates a loopback reply message by simply swapping the MAC destination and source addresses of the loopback message and changing its `opcode` value to the loopback reply opcode. Of course, if the loopback message was addressed to a multicast destination, the loopback message's multicast MAC destination address field cannot be used as the source address of the loopback reply. So, the transmitting maintenance entity group endpoint's MAC address is used instead in this case.

In order to avoid inundating an originator of a multicast loopback message with a flood of loopback replies from all of its peer maintenance entity group endpoints, the remote maintenance entity group endpoint inserts a randomized delay of from zero to one second ahead of its transmission of the loopback reply.

If a configured maintenance entity group intermediate point receives a multicast loopback reply message, it forwards the packet in the same direction that it was heading when it was received by the maintenance entity group intermediate point. This is done because maintenance entity group intermediate points are transparent to multicast loopback reply messages.

Packet Loss Measurement

Packet loss measurement bears some similarity to time sync and delay measurement. Specifically, statistics counters (rather than timestamp counters) are sampled in reaction to the transmission or reception of certain packet types. Packet loss is detected by comparing deltas of local counters with deltas of remote counters over a period of time. Differences between the deltas represent packet loss.

And, as is the case with time sync, the closer to the transmission medium the sampling of the counters occurs, the more accurate the measurements will be. Consider a CPU-only implementation of an OAM function; in such an implementation, the CPU determines that it is time to transmit a loss measurement message (LMM) to a remote maintenance entity group endpoint. First it must sample the relevant statistics counter in order to include its value in the loss measurement message. After sampling the counter, the CPU formulates the packet and queues it for transmission. Of course, during the period between the sampling of the counters and the queuing of the loss measurement message packet, a large and unpredictable number of packets may be transmitted that are (correctly) counted by the just-sampled counter. Thus, when the loss measurement message is actually transmitted, its counter value is immediately out of date. Similar accounting errors can occur on the receiving remote maintenance entity group endpoint.

Hardware-based sampling of the counter values and hardware-based writing of the loss measurement message's counter values are beneficial to enhance the accuracy of the loss measurement. The details of this follow, but in a nutshell:

CPU-generated loss measurement message and loss measurement reply packets are detected in the egress processing function prior to transmission. The relevant statistics are sampled and inserted into the packets upstream of the transmit MAC. When loss measurement message or loss measurement reply packets are received, the relevant statistics are sampled and their values are delivered to the CPU for processing. The points at which this work is accomplished must be close enough to the transmit MAC that it is impossible (or extremely unlikely) that other packets in the same flow may arrive at the counter sample point out of order relative to loss measurement message and loss measurement reply message packets between the transmission medium and the statistics sampling points.

The loss measurement message format and field definitions are presented in Figure 121 and Table 51.

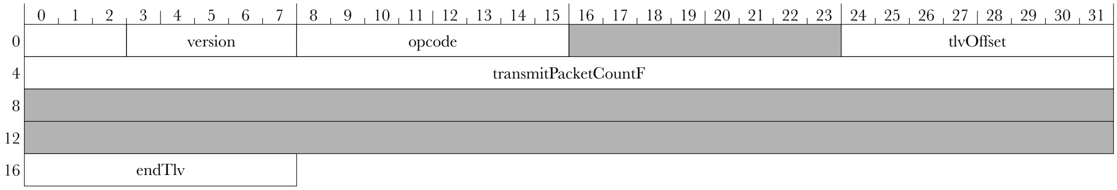


Figure 121 *Ethernet OAM Loss Measurement Message Structure Diagram*

Table 51 *Ethernet OAM Loss Measurement Message Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`megLevel` (MEG level)	3 bits	0.0	The loss measurement message's maintenance entity group level. Higher maintenance entity group levels span greater distances across the network and are associated with end-to-end connections (i.e., customers). Lower levels are associated with shorter hops (i.e., network operators).
`version` (Version)	4 bits	0.3	Protocol version number. Must be set to `0`.
`opcode` (OpCode)	8 bits	0.8	The message's opcode. For an loss measurement message, this field must be set to `43`.
`tlvOffset` (TLV offset)	8	0.24	Offset of the next TLV. This field specifies the offset of the TLV that follows the current TLV. For loss measurement messages, this value must be `12`.
`transmitPacketCountF` (TxFCf)	32	4.0	The number of packets transmitted toward a maintenance entity group endpoint addressed by this OAM packet.
`endTlv` (End TLV)	8	16.0	Marks the end of a sequence of TLVs. This field must be set to `0`.

The loss measurement reply format and field definitions are presented in Figure 122 and Table 52.

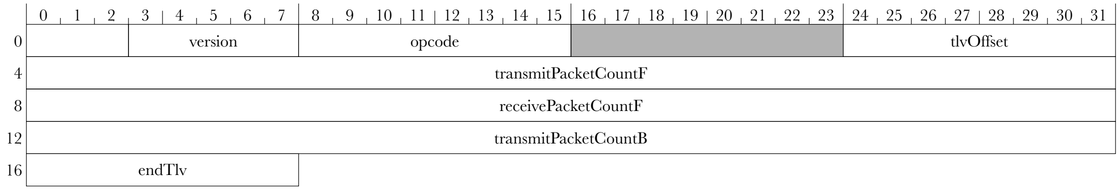


Figure 122 Ethernet OAM Loss Measurement Reply Structure Diagram

Table 52 Ethernet OAM Loss Measurement Reply Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`megLevel` (MEG level)	3 bits	0.0	The loss measurement reply's maintenance entity group level. Higher maintenance entity group levels span greater distances across the network and are associated with end-to-end connections (i.e., customers). Lower levels are associated with shorter hops (i.e., network operators).
`version` (Version)	4 bits	0.3	Protocol version number. Must be set to `0`.
`opcode` (OpCode)	8 bits	0.8	The message's opcode. For an loss measurement reply, this field must be set to `42`.
`tlvOffset` (TLV offset)	8	0.24	Offset of the next TLV. This field specifies the offset of the TLV that follows the current TLV. For loss measurement replies, this value must be `12`.
`transmitPacketCountF` (TxFCf)	32	4.0	The number of packets transmitted toward a maintenance entity group endpoint addressed by this OAM packet.
`receivePacketCountF` (RxFCb)	32	8.0	The number of packets received from the direction of the peer maintenance entity group endpoint at the time of the reception of the corresponding loss measurement message.
transmitPacketCountB (TxFCb)	32	12.0	The `transmitPacketCountF` value from the loss measurement message most recently received from a maintenance entity group endpoint's peer maintenance entity group endpoint.
`endTlv` (End TLV)	8	16.0	Marks the end of a sequence of TLVs. This field must be set to `0`.

Loss Measurement Method

There are two fundamental modes of operation for packet loss measurement: single-ended and dual-ended. Single-ended loss measurement utilizes dedicated message types: loss measurement message (LMM) and loss measurement reply (LMR). Loss measurement messages are generated by the local maintenance entity group endpoint in an on-demand basis and loss measurement replies are generated in response to received loss measurement messages. The exchange of messages between peer maintenance entity group endpoints gives the initiating maintenance entity group endpoint all of the information required to determine the packet loss counts for both directions on the path between the maintenance entity group endpoints.

Dual-ended loss measurement uses provisions in the continuity check messages (CCMs). Since OAM continuity check is a one-way protocol (i.e., no explicit reply is sent for each received continuity check message), each maintenance entity group endpoint must rely on the periodic reception of continuity check messages bearing the necessary statistics counter values. From these received continuity check messages and the sampling of its own local statistics counters, the local maintenance entity group endpoint is able to determine the packet loss counts for both directions on the path between the maintenance entity group endpoints.

Loss measurement messages, loss measurement replies and continuity check messages each have provisions for 32-bit counter value fields. These fields are described in Table 53.

Table 53 Loss Measurement Counter Fields

Field Name	Single-Ended		Dual-Ended
	LMM	LMR	CCM
`transmitPacketCountF`	Count of packets transmitted by local maintenance entity group endpoint to peer maintenance entity group endpoint at time of loss measurement message transmission.	Value of `transmitPacketCountF` from the last loss measurement message received from the peer maintenance entity group endpoint (i.e., the received value is copied back to the initiating maintenance entity group endpoint).	Count of packets transmitted by the local maintenance entity group endpoint to the peer maintenance entity group endpoint at the time of continuity check message transmission.
`receivePacketCountF`	Placeholder for loss measurement reply `receivePacketCountF` value.	Count of packets received from peer maintenance entity group endpoint at the time of the last loss measurement message reception.	N/A

Field Name	Single-Ended		Dual-Ended
	LMM	LMR	CCM
`transmitPacketCountB`	N/A	N/A	Count of packets received from a peer maintenance entity group endpoint at the time of the last continuity check message reception from that peer maintenance entity group endpoint.
`receivePacketCountB`	Placeholder for loss measurement reply `receivePacketCountB` value.	Count of packets transmitted by the local maintenance entity group endpoint to a peer maintenance entity group endpoint at the time of loss measurement reply transmission.	Value of `transmitPacketCountF` from the last continuity check message received from the peer maintenance entity group endpoint (i.e., the received value is copied back to the peer maintenance entity group endpoint).

Consider the single-ended loss measurement technique. The initiating maintenance entity group endpoint transmits a loss measurement message that contains the relevant statistics counter (the number of transmit packets at the time the loss measurement message is transmitted). The remote maintenance entity group endpoint has no real need for this information. It is simply embedded in the loss measurement message to avoid the local maintenance entity group endpoint (the initiator of the transaction) from having to store that state while it awaits a reply from its peer maintenance entity group endpoint; the peer maintenance entity group endpoint returns the value to the local maintenance entity group endpoint in its loss measurement reply.

When the peer maintenance entity group endpoint receives the loss measurement message, it immediately samples the relevant receive packet statistics counter and formulates the loss measurement reply. The loss measurement reply loops back the transmit packet statistics from the loss measurement message and includes the receive packet statistic as well as the transmit packet statistic that is valid at the time of the loss measurement reply's transmission.

Upon receipt of the loss measurement reply, the local maintenance entity group endpoint samples the relevant receive packet statistic. This gives the local maintenance entity group endpoint four statistics counter values:

1. outbound (from local to peer) transmit packet count
2. outbound receive packet count
3. inbound (from peer to local) transmit packet count
4. inbound receive packet count

By computing the differences between transmit and receive counts for the outbound and inbound paths, an offset for the two sets of counters can be determined. When a second loss measurement message/loss measurement reply transaction takes place a short time in the future, any change in the offset of the counter-pairs reflects packets that have been lost on the path being measured.

Now consider the dual-ended loss measurement technique. This loss measurement method is intended to give a local maintenance entity group endpoint an indication of packet loss among the packets that it has transmitted to a peer maintenance entity group endpoint. To do this, each local maintenance entity group endpoint transmits continuity check messages to its peer maintenance entity group endpoints on a periodic basis. These transmit continuity check messages convey the transmit packet count statistic that corresponds to the path that is being tested by the continuity check message.

When the remote maintenance entity group endpoint receives a continuity check message, it immediately samples the relevant receive statistics counter. The sampled receive statistics and the transmit statistic contained in the received continuity check message are packaged into the next available transmit continuity check message, which is then transmitted back to the local maintenance entity group endpoint.

Upon receipt of the continuity check message by the local maintenance entity group endpoint, it has both its own transmit count and the remote maintenance entity group endpoint's receive count, giving it the information necessary to detect any packet losses in the outbound path. The local maintenance entity group endpoint cannot compute the inbound packet losses using the dual-ended loss measurement method.

Counting Packets

A series of packet counters is required in order to support the OAM packet loss measurement function. Each configured local maintenance entity group endpoint must maintain a pair of counters for each of its peer maintenance entity group endpoints. One counter counts packets transmitted toward a peer maintenance entity group endpoint while the other counts packets received from a peer maintenance entity group endpoint.

Each MAC address/VLAN ID combination can be associated with a maintenance entity group endpoint through which packets must pass if they are addressed to that combination or originate from that combination. Therefore, it is a simple matter to associate each address in the source and destination MAC address tables with a loss measurement statistics counter. Because to-many relationships between the maintenance entity group endpoints and the stations that are addressable via the maintenance entity group endpoints are common, the number of loss measurement statistics counters can be far less than the number of known MAC addresses.

These statistics counters do not count every packet associated with a maintenance entity group endpoint. Specifically, while OAM packets whose maintenance entity group level is greater than that of the local maintenance entity group endpoint are counted just a normal data packet would be counted, other OAM packets (those at the same maintenance entity group level) are counted selectively. Table 54 summarizes the criteria.

Table 54 Loss Measurement OAM Packet Counting Criteria

OAM Function	Single-Ended Loss Measurement	Dual-Ended Loss Measurement
Loopback	do not count	do not count
Link Trace	do not count	do not count
Loss Measurement	do not count	do not count
Delay Measurement	do not count	do not count
Test Signal	do not count	do not count
Continuity Check	count	do not count
Automatic Protection Service	count	count

All loss measurement statistics counters are 32-bit counters. It is required that the loss measurement message, loss measurement reply and continuity check message packets are sent often enough relative to data packet rates in the maintenance entity group that counter rollover is not an issue. These counters do not need to be reset to zero since deltas between counter samples are used, rather than their absolute values, for packet loss determinations.

Supporting Loss Measurement in Up MEPs and Down MEPs

Supporting packet loss measurement in down MEPs (the ones that face the ports) is fairly straightforward. Packets entering ingress packet processing are treated like receive packets—just like their data-packet brethren. Similarly, packets that enter egress packet processing are typically on the verge of being transmitted by one of the local transmit ports.

Up MEPs, on the other hand, are a bit less intuitive. In the ingress path, up MEPs are a source of packets that need to be transmitted after being bridged. This also means that the packet loss measurement process needs to start prior to encountering the queuing functionality that is part of the bridging process. In other words, it is required for an up MEP to also measure the packet losses that occur in the queuing operations of the maintenance entity group endpoint's own bridge.

Similarly, an up MEP configured in the egress packet processing path is a destination for packets that just crossed the bridge's queuing and forwarding system. So, even though these packets are in the bridge's transmit path, they are received packets from the up MEPs' perspective. So, loss measurement packets that are terminated in an up MEP must trigger the same kind of receive packet behavior that a similar receive packet triggers as it is being terminated by a down MEP.

Delay Measurement

The OAM delay measurement function is used to measure the packet delay between peer maintenance entity group endpoints in the same maintenance entity and at the same maintenance entity group level. Delay measurement can be accomplished using either a one-way method or a two-way method. The one-way method requires that the clocks in the maintenance entity group endpoints be synchronized via IEEE 1588 or some other means. Because the clocks are synchronous, it is a simple matter to insert a transmit timestamp into a delay measurement message (DMM) and for the destination maintenance entity group endpoint to compare that timestamp value to the receive timestamp measured upon receipt of the packet. The difference between the two timestamp values is the delay from the originating maintenance entity group endpoint and the terminating maintenance entity group endpoint.

The delay measurement message format and field definitions are presented in Figure 123 and Table 55.

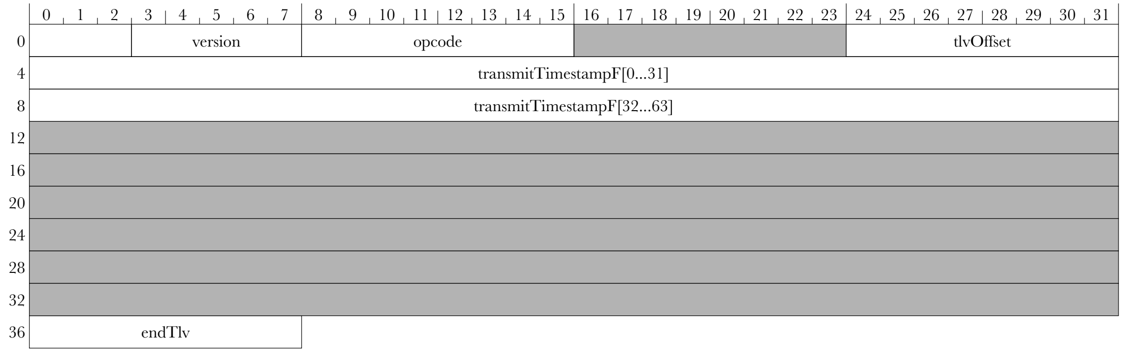


Figure 123 Ethernet OAM Delay Measurement Message Structure Diagram

Table 55 *Ethernet OAM Delay Measurement Message Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`megLevel` (MEG level)	3 bits	0.0	The delay measurement message's maintenance entity group level. Higher maintenance entity group levels span greater distances across the network and are associated with end-to-end connections (i.e., customers). Lower levels are associated with shorter hops (i.e., network operators).
`version` (Version)	4 bits	0.3	Protocol version number. Must be set to `0`.
`opcode` (OpCode)	8 bits	0.8	The message's opcode. For an delay measurement message, this field must be set to `47`.
`tlvOffset` (TLV offset)	8	0.24	Offset of the next TLV. This field specifies the offset of the TLV that follows the current TLV. For delay measurement messages, this value must be `32`.
`transmitTimestampF` (TxTimeStampf)	64	4.0	The timestamp of the transmit time of this delay measurement message packet.
`endTlv` (End TLV)	8	36.0	Marks the end of a sequence of TLVs. This field must be set to `0`.

The delay measurement reply format and field definitions are presented in Figure 124 and Table 56.

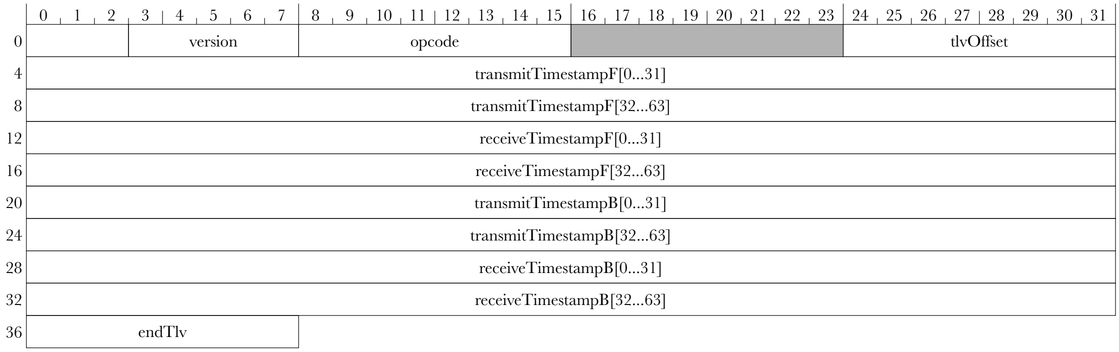

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
0
version
opcode
tlvOffset
4
transmitTimestampF[0...31]
8
transmitTimestampF[32...63]
12
receiveTimestampF[0...31]
16
receiveTimestampF[32...63]
20
transmitTimestampB[0...31]
24
transmitTimestampB[32...63]
28
receiveTimestampB[0...31]
32
receiveTimestampB[32...63]
36
endTlv

Figure 124 *Ethernet OAM Delay Measurement Reply Structure Diagram*

Table 56 Ethernet OAM Delay Measurement Reply Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`megLevel` (maintenance entity group level)	3 bits	0.0	The delay measurement reply's maintenance entity group level. Higher maintenance entity group levels span greater distances across the network and are associated with end-to-end connections (i.e., customers). Lower levels are associated with shorter hops (i.e., network operators).
`version` (Version)	4 bits	0.3	Protocol version number. Must be set to `0`.
`opcode` (OpCode)	8 bits	0.8	The message's opcode. For an delay measurement reply, this field must be set to `46`.
`tlvOffset` (TLV offset)	8	0.24	Offset of the next TLV. This field specifies the offset of the TLV that follows the current TLV. For loss measurement replies, this value must be `32`.
`transmitTimestampF` (TxTimeStampf)	64	4.0	The transmit time of the corresponding delay measurement message packet.
`receiveTimestampF` (RxTimeStampf)	64	12.0	The receive time of the corresponding delay measurement message packet.
`transmitTimestampB` (TxTimeStampb)	64	20.0	The transmit time of this delay measurement reply packet.
`receiveTimestampB` (RxTimeStampb)	64	28.0	The receive time of this delay measurement reply packet.
`endTlv` (End TLV)	8	16.0	Marks the end of a sequence of TLVs. This field must be set to `0`.

Two-way delay measurement must be used if the maintenance entity group endpoints do not have synchronized time-of-day clocks. The two-way method uses the following sequence of events:

1. Delay measurement message Transmission

   The local maintenance entity group endpoint periodically transmits delay measurement messages to the remote maintenance entity group endpoints. During transmission, a hardware-based timestamping mechanism inserts the actual transmission time into the delay measurement message packet.

2. Delay measurement message Reception and delay measurement reply Transmission

Upon reception of the delay measurement message packet by the remote maintenance entity group endpoint, the remote maintenance entity group endpoint timestamps the reception of that packet and formulates a delay measurement reply packet. The original transmit time stamp from the delay measurement message packet is copied over to the delay measurement reply packet long with the receive timestamp of the delay measurement message. As the delay measurement reply is being transmitted, a hardware-based timestamping mechanism inserts the actual transmission time into the delay measurement reply packet.

3. Delay measurement reply Reception

   Upon reception of the delay measurement reply packet by the local maintenance entity group endpoint, the local maintenance entity group endpoint timestamps the reception of that packet. This gives the local maintenance entity group endpoint a total of four timestamp values that can be used to compute the round trip time between the local and remote maintenance entity group endpoints using the following formula:

$$\texttt{delay} = (\texttt{t}_{\texttt{localReceive}} - \texttt{t}_{\texttt{localTransmit}}) - (\texttt{t}_{\texttt{remoteTransmit}} - \texttt{t}_{\texttt{remoteReceive}})$$

The packet formats differ between ITU-T Y.1731 and IEEE 1588 and the timestamping differs in the details as well. Whereas IEEE 1588 can get by just fine with a 64-bit timestamp mechanism that is used as a basis for interval time measurement (i.e., nanoseconds and fractions of nanoseconds), OAM requires a 64-bit absolute time of day (i.e., 32 bits of seconds and 32 bits of nanoseconds wall clock time) timestamp (essentially, the least significant 64 bits of the 80-bit IEEE 1588 timestamp).

Supporting Delay Measurement in Up MEPs and Down MEPs

In the ingress path, up MEPs are a source of packets that need to be transmitted after being bridged. This also means that the delay measurement process needs to start prior to encountering the queuing functionality that is part of the bridging process. In other words, it is required for an up MEP to also measure the delays that are incurred in the queuing operations of the maintenance entity group endpoints own bridge. To do this, the packet must be timestamped as it enters the ingress processing function (generally at a point that is just downstream of the Ethernet MACs). This is a separate timestamping function from the one implemented in the Ethernet MACs.

Similarly, an up MEP configured in the egress packet processing path is a destination for packets that just crossed the switching fabric. So, even though these packets are in the bridge's transmit path, they are received packets from the up MEPs' perspective. So, delay measurement packets that are terminated in an up MEP must trigger the same kind of receive packet behavior that a similar receive packet being terminated by a down MEP triggers. Meaning, the up MEP configured to support delay measurement also measures the delay encountered by a packet across the switching fabric of the maintenance entity group endpoint's bridge. Timestamping the packet in the egress processing pipeline prior to reaching the Ethernet MACs is required to support delay measurement in up MEPs.

MPLS OAM

Two different IETF RFCs specify OAM behavior for MPLS that largely mimics the OAM features previously described for Ethernet. The two standards are: RFC 6374 (Packet Loss and Delay Measurement for MPLS Networks) and RFC 5880 (Bidirectional Forwarding Detection).

Packet Loss and Delay Measurement

RFC 6374 specifies three different message types:

- loss measurement
- delay measurement
- loss and delay measurement

Since the loss and delay measurement message is really just a loss measurement message combined with a delay measurement message, the loss measurement and delay measurement messages and functions may be described separately while still adequately describing their combined message type.

Packet Loss Measurement Method

The MPLS packet loss measurement method entails configuring pairs of periodic MPLS query and response messages. When generating a query message, the querying node samples a packet (or byte) counter that is associated with the packet's outermost MPLS label (i.e., the packet's path) and inserts that counter value into one of four fields reserved for this purpose in the loss measurement message. Upon receipt of a loss measurement query, the responding node samples its own per-path counter and inserts the value in the appropriate field in the message. The loss measurement message is then turned into a response message and is transmitted back to the original querying node, gathering two more path-specific counter sample values along the way (a transmit count from the responder and a receive count from the querier). When two such two-way exchanges occur (with plenty of non-OAM traffic using the same path between the pair of exchanges), the differences between the counters are compared to determine how many packets or bytes were lost.

A loss measurement query begets a loss measurement reply. The reply that is received by the querier contains all four relevant packet counter values: querier transmit, responder receive, responder transmit, and querier receive. Table 57 lists some example values and how packet loss is derived.

Table 57 *Loss Measurement Example*

Counter Type	Response n	Response n+1	Delta	Loss
Query Transmit	1,000	1,500	500	0
Response Receive	800	1,300	500	
Response Transmit	3,400	3,850	450	5
Query Receive	2,100	2,545	445	

In the example in Table 57, the differences between the two loss measurement response measurements are equal in the querier-to-responder path, but there's a difference of 5 in the responder-to-querier path.

RFC 6374 allows 32- or 64-bit counters and the counting of packets or bytes. By default, 64-bit counters are used. However, if any of the forwarding systems in a maintenance entity group are only capable of supporting 32-bit counters, then all of the forwarding systems must revert to using 32-bit counters. When 32-bit counters are in use, the upper 32-bits of the 64-bit counter fields in the loss measurement messages are set to `0`.

A table of counters of some sufficient size is used to maintain the various loss measurement counters. The contents of this table are addressed by, at least, the `label` value from the topmost MPLS header. Additional information such as `Mpls.priority` and even `Mac.destinationAddress` may be included when selecting counters in order to provide greater loss measurement granularity. Separate tables of counters are required for received and transmitted packets.

Direct vs. Inferred Loss Measurement

IETF RFC 6374 specifies two methods for loss measurement: direct and inferred. The message formats for direct and inferred loss measurement are identical. The difference between direct and inferred has to do with whether or not it is possible to precisely count the number of non-OAM packets transmitted and received between pairs of OAM loss measurement messages. Direct loss measurement requires a hardware process that samples the counters in line with the data packets and can thus ensure that the counts are accurate. For inferred loss measurement, the loss measurement counters are sampled by a software process that then transmits or receives the required OAM messages. The accuracy of inferred loss measurement cannot be guaranteed because the highly variable delays of the software process allows a variable number of non-OAM packets to be counted by the counter between software sampling and OAM message reception or transmission.

Packet Delay Measurement Method

RFC 6374 supports both one-way and two-way delay measurement. In one-way delay measurement, a delay measurement query message is sent to a responder, but no response is generated. The responder is able to determine the delay encountered by the message simply by examining the single set of transmit and receive timestamp values in the one message. One-way delay measurement depends upon the sender and receiver having synchronized time-of-day clocks. If the two nodes do not have synchronized time-of-day clocks, then two-way delay measurement must be used.

In a two-way exchange of delay measurement messages, a response message is generated upon receipt of a query message. The timestamps of the query message are copied to the response messages and the response message's own transmit and receive timestamps are added to the list of timestamp conveyed to the querying node. Once the querying node has all four timestamps, the round trip time (exclusive of the turnaround time at the responder) is easily computed:

$$(t_{responseReceive} - t_{queryTransmit}) - (t_{responseTransmit} - t_{queryReceive})$$

Message Encapsulation and Formats

Like Ethernet OAM, MPLS OAM supports the concept of a hierarchy of maintenance associations that correspond to the various users, service providers and network operators that operate and maintain MPLS-based networks. However, unlike Ethernet OAM, MPLS OAM does not explicitly encode the hierarchical level at which an OAM message is being transported. Instead, MPLS OAM relies on the native hierarchical nature of MPLS that's realized through its use of stacks of MPLS headers. Specifically, the outermost header (i.e., the top of stack) is always associated with the owner/operator of the links and equipment that any MPLS packet is currently traversing.

A normal MPLS data packet's outermost MPLS label controls the forwarding decision for the current level of network hierarchy. Similarly, an MPLS OAM packet uses the same `Mpls.label` values as a data packet for forwarding from one point in a particular level of hierarchy to another point. The difference is that the outermost MPLS header of an MPLS OAM packet is not followed by more forwarding headers.

All RFC 6374 OAM messages are conveyed by an MPLS Generic Associated Channel Header (G-ACh, RFC 5586). The bottom-of-stack label for an MPLS OAM packet is a Generic Associated Channel label (GAL). An Associated Channel Header (ACH) follows the bottom-of-stack and is used to identify the type of MPLS OAM message being conveyed. This header stack-up is shown Figure 125.

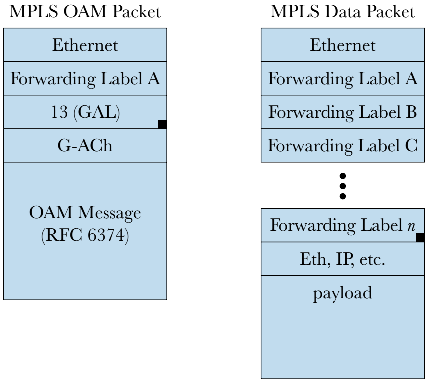


Figure 125 *MPLS OAM Header Stack-Up*

Figure 125 presumes that Forwarding Label A for both the MPLS OAM packet and the MPLS data packet have the same value—making them part of the same flow. What distinguishes the two packets is the presence of a Generic Associated Channel Header label versus a MPLS header that may be used to route the packet once the preceding labels of the MPLS stack have been popped.

MPLS associated channels are discussed in "*Generic Associated Channel Header (G-ACh)*" *on page 108*.

The generic associated channel header's `channelType` field is enumerated for OAM purposes as follows:

```
10 = direct loss measurement
11 = inferred loss measurement
12 = delay measurement
13 = direct loss and delay measurement
14 = inferred loss and delay measurement
```

The structure of RFC 6374 MPLS messages are shown in Figures 126, 127, and 128.

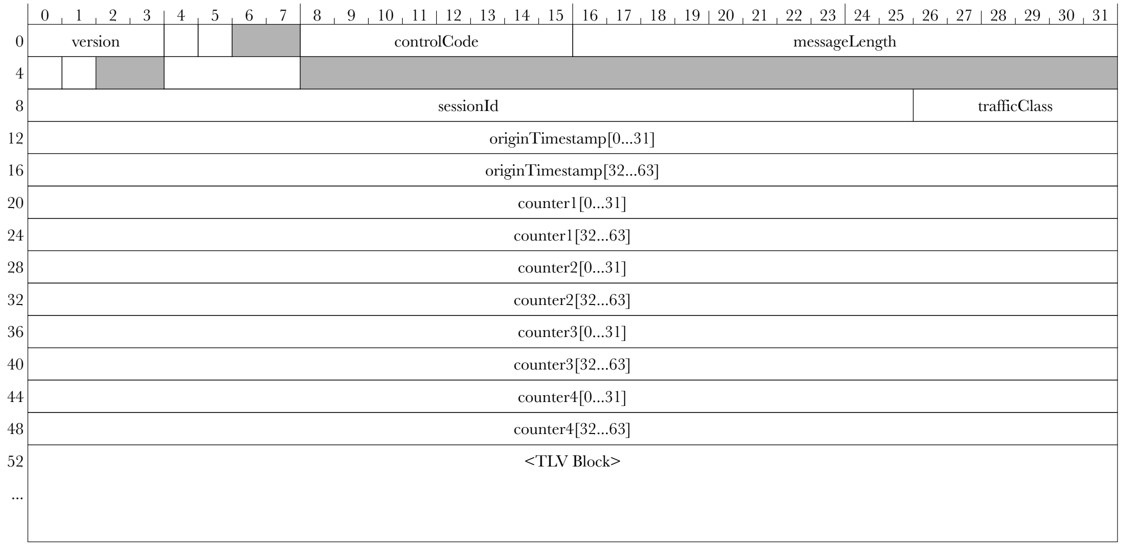


Figure 126 MPLS OAM Loss Measurement Message Structure Diagram

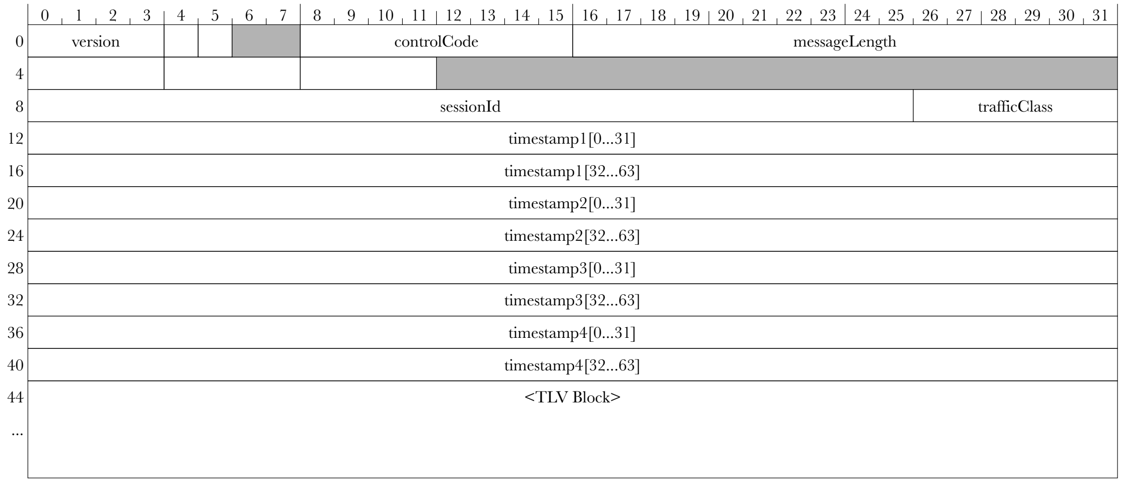


Figure 127 MPLS OAM Delay Measurement Message Structure Diagram

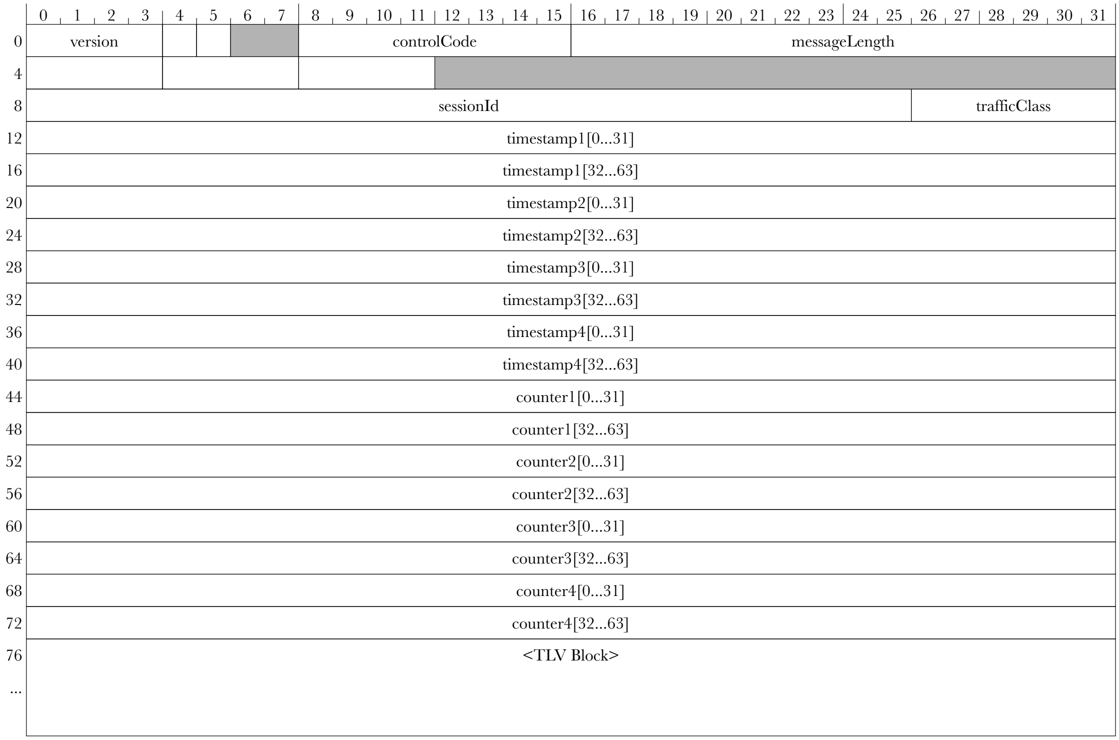


Figure 128 *MPLS OAM Loss and Delay Measurement Message Structure Diagram*

The various fields that make up the MPLS OAM loss and delay measurement messages are defined in Table 58.

Table 58 *MPLS OAM Loss and Delay Measurement Message Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`version` (Version)	4 bits	0.0	Protocol version. Must be set to `0`.
`responseMessage` (R)	1 bit	0.4	Distinguishes between query and response messages. This bit is set to a `1` to indicate that the current message is a response message. Setting this bit to a `0` indicates that it is a query message.
`trafficClassAware` (T)	1 bit	0.5	Distinguishes between traffic class-aware and -unaware behavior. If this bit is set to a `1`, then the OAM message applies just to the traffic class identified by `trafficClass`. Otherwise, `trafficClass` is ignored and the message is applied to all traffic classes.

Field Name (std. name)	Width	Offset (B.b)	Definition
`controlCode` (Control Code)	8 bits	0.8	Indicates how the message should be handled. For query messages, this field is enumerated as follows: 0 = `inBand` Response is expected over the same channel. 1 = `outOfBand` Response is expected via an out-of-band channel. 2 = `none` No response is expected. For response messages, this field is enumerated as follows: 1 = `success` 2 = `dataFormatInvalid` 3 = `initInProgess` 4 = `dataResetOccurred` 5 = `resourceTemporarilyUnavailable` ≥16 = error (see RFC 6374 section 3.1 for details)
`messageLength` (Message Length)	16 bits	0.16	The total length of the message in bytes.
`extendedCounterFormat` (X)	1 bit	4.0	Indicates the counter width. If this bit is set to `1`, then the counter values are 64 bits wide. Otherwise the counters are 32 bits wide (occupying the least significant 32 bits of the 64-bit counter fields). This field is only present in the loss measurement message.
`byteCounters` (B)	1 bit	4.1	Indicates byte versus packet counters. If this bit is set to `1`, then the counters count bytes. Otherwise, they count packets. This field is only present in the loss measurement message.
`querierTimestampFormat` (QTF)	4 bits	4.0	Indicates the timestamp format. These values are used to indicate the format of either the timestamps in the current message, or, in the case of responderPreferredTimestampFormat, the preferred format for a response. These fields are enumerated as follows: 1 = `sequenceNumber` Not a timestamp. 2 = `networkTimeProtocol` 32 bits of seconds and 32 bits of fractions of a second. 3 = `precisionTimeProtocol` 32 bits of seconds and 32 bits of nanoseconds (modulo 10^9)
`responderTimestampFormat` (RTF)	4 bits	4.4	
`responderPreferredTimestampFormat` (RPTF)	4 bits	4.8	

Field Name (std. name)	Width	Offset (B.b)	Definition
`sessionId` (Session Identifier)	26 bits	8.0	Arbitrary session identifier. The querying node fills this field with an arbitrary value. The responding node copies it to its response so that the querier can correlate the response with an earlier query.
`trafficClass` (DS)	6 bits	8.26	The message's traffic class. This field indicates to which traffic class the current message applies. This field is ignore if `trafficClassAware` is set to `0`.
`originTimestamp` (Origin Timestamp)	64 bits	12.0	The transmit time of the loss measurement message.
`timestamp[1...4]` (Timestamp [1...4])	64 bits	various	The timestamps associated with delay measurement operations. The timestamp fields are used as follows: `timestamp1` = current transmit time `timestamp2` = current receive time `timestamp3` = previous transmit time `timestamp4` = previous receive time
`counter[1...4]` (Counter [1...4])	64 bits	various	The counters associated with loss measurement operations. The counter fields are used as follows: `counter1` = current transmit count `counter2` = current receive count `counter3` = previous transmit count `counter4` = previous receive count

Bidirectional Forwarding Detection (BFD)

RFC 5880 defines a method for detecting faults in MPLS networks. Conceptually, the method is quite similar to Ethernet OAM continuity check message messaging as described in "*Continuity Check*" *on page 248*.

In a nutshell, control messages are sent periodically to a destination that is expecting to receive these messages. If a certain number of transmission periods transpire without the receipt of these messages, the receiver presumes that a link or a forwarding system along the path between itself and the sender has failed.

The BFD control message is the basic unit of information transfer between pairs of nodes that make up a continuity-check association. When the BFD protocol is first started up, the node begins transmitting control messages at a leisurely pace, advertising itself to another node with which it wishes to detect continuing connectivity. The addressed node reacts to these messages by sending its own control messages. Once the basic communication path has been established, the frequency of control message transmission is increased to a rate that provides the desired error detection reaction time.

Because RFC 5880 messages are conveyed via MPLS LSPs that are dedicated to the purpose, no special features are required to support their origination and termination. All MPLS packets addressed to a locally configured router whose outermost MPLS label matches a value that is configured for the BFD function are generally forwarded to the forwarding system's management CPU. Similarly, all transmitted BFD-related messages are generated by the forwarding system's management CPU.

The format of a BFD control message is depicted in Figure 129.

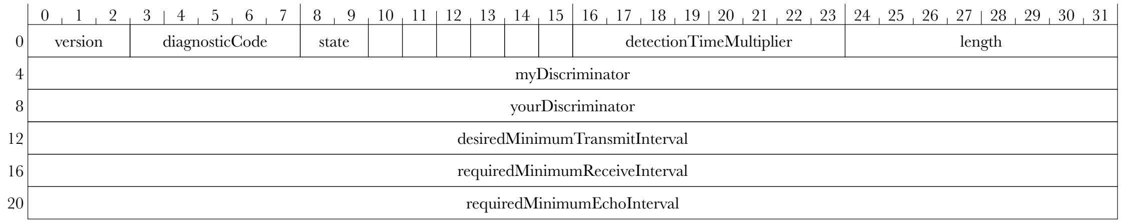


Figure 129 BFD Control Message Structure Diagram

BFD control message fields are defined in Table 59.

Table 59 BFD Control Message Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`version`	3 bits	0.0	The version of the protocol. RFC 5880 defines version `1`.

`diagnosticCode`
(Diag)
	5 bits	0.3	A diagnostic code that explains the reason for a state change.
			This field is enumerated as follows:
			`0 = noDiagnostic`
			`1 = controlDetectionTimeExpired`
			`2 = echoFunctionFailed`
			`3 = neighborSignaledSessionDown`
			`4 = forwardingPlaneReset`
			`5 = pathDown`
			`6 = concatenatedPathDown`
			`7 = administrativelyDown`
			`8 = reverseConcatenatedPathDown`
			`9–31 = reserved`

Field Name (std. name)	Width	Offset (B.b)	Definition
`state` (Sta)	2 bits	0.8	Transmitting system state. This field is enumerated as follows: 0 = `administrativelyDown` 1 = `down` 2 = `initial` 3 = `up`
`poll` (P)	1 bit	0.10	If this bit is set to `1`, the transmitting system is requesting verification of connectivity, or of a parameter change, and is expecting a packet with the `final` bit set to `1` in reply. If this bit is set to `0`, the transmitting system is not requesting verification.
`final` (F)	1 bit	0.11	If this bit is set to `1`, the transmitting system is responding to a received BFD control message whose `poll` bit was set to `1`. If this bit is set to `0`, the transmitting system is not responding to a Poll.
`controlPlaneIndependent` (C)	1 bit	0.12	If this bit is set to `1`, the transmitting system's BFD implementation does not share fate with the control plane (in other words, BFD is implemented in the forwarding plane and can continue to function through disruptions in the control plane). If this bit is set to `0`, the transmitting system does share fate with its control plane.
`authenticationPresent` (A)	1 bit	0.13	If this bit is set to 1, the Authentication Section (an optional header that follows this header) is present and the session must be authenticated. Refer to RFC 5880 for details.
`demand` (D)	1 bit	0.14	If this bit is set to `1`, demand mode is active in the transmitting system (i.e., the system wishes to operate in demand mode, knows that the session is up in both directions, and is directing the remote system to cease the periodic transmission of BFD control messages). If this bit is set to `0`, demand mode is not active in the transmitting system.
`multipoint` (M)	1 bit	0.15	This bit is reserved for future multipoint extensions to BFD. This bit must be set to `0` on transmit and ignored on receipt.
`detectionTimeMultiplier` (Detect Mult)	8 bits	0.16	The negotiated transmit interval, multiplied by this value, provides the detection time for the receiving system in asynchronous mode.

Field Name (std. name)	Width	Offset (B.b)	Definition
`length` (Length)	8 bits	0.24	The length of the BFD control message in bytes.
`myDiscriminator` (My Discriminator)	32 bits	4.0	A unique, non-zero discriminator value generated by the transmitting system, used to demultiplex multiple BFD sessions between the same pair of systems.
`yourDiscriminator` (Your Discriminator)	32 bits	8.0	The `myDiscriminator` value received from the corresponding remote system is copied to this field. This field is set to `0` if `myDiscriminator` is unknown.
`desiredMinimumTransmitInterval` (Desired Min TX Interval)	32 bits	12.0	This is the minimum interval—in microseconds—that the local system would like to use when transmitting BFD control messages, less any jitter applied. The value `0` is reserved.
`requiredMinimumReceiveInterval` (Required Min TX Interval)	32 bits	16.0	This is the minimum interval—in microseconds—between received BFD control messages that this system is capable of supporting, less any jitter applied by the sender. If this value is `0`, the transmitting system does not want the remote system to send any periodic BFD control messages.
`requiredMinimumEchoInterval` (Required Min Echo RX Interval)	32 bits	20.0	This is the minimum interval—in microseconds—between received BFD echo messages that this system is capable of supporting, less any jitter applied by the sender. If this value is `0`, the transmitting system does not support the receipt of BFD control messages.

16 Security

Network security is a huge topic; there's more than enough material to fill an entire book. Indeed, many books have been written on the topic. Fortunately, the hardware-centric focus of this book greatly narrows the scope of the topic. Though a lot of network security functions, features and protocols operate strictly in the control plane as software modules, there's still plenty for hardware to do.

Broadly speaking, network security has the following goals:

- forbid unauthorized or malicious access (access control)
- assure data authenticity and integrity (authentication)
- protect data privacy (encryption)
- protect the network itself (denial-of-service defenses)

Access Control

Generally, we associate access authorization with passwords used to gain access to various servers attached to the Internet. There is also a level of access protection that may be used at the forwarding systems themselves to prevent unauthorized access to the network long before these packets arrive at a server. The further you can keep those with malicious intent away from valuable resources, the better.

Port-Based Network Access Control (IEEE 802.1X)

The IEEE 802.1X standard defines a system of services, features and protocols that enable port-based access control. The concept is fairly simple, though it gets complicated when all of the options and the details of the authentication protocols are considered. In a nutshell, a physical network interface (i.e., a port) that supports 802.1X comes up in a "unauthorized" state wherein only 802.1X packets are permitted and all others are discarded.

There are three components involved in the authentication process as shown in Figure 130:

- supplicant (the host or user seeking access)
- authenticator (the access point into the network)
- authentication server (the shared arbiter of authentication credentials)

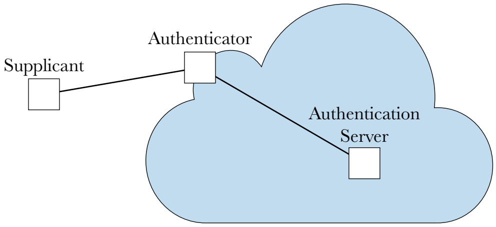


Figure 130 IEEE 802.1X Components

When a physical interface is in the unauthorized state and a new host (supplicant) is detected, the authenticator periodically transmits identity request messages to it. These messages are addressed to well-known multicast addresses and provide each supplicant with information regarding the requirements for authenticating the supplicant. In response, the supplicant identifies itself to the authenticator by offering up its credentials in an identity response packet. The authenticator, in turn, communicates with an authentication server in order to validate the authentication credentials supplied by the supplicant. Ultimately, the supplicant is either granted or denied access. If access is granted, then the affected physical interface transitions to an "authorized" state and the supplicant is permitted to send and receive packets in a normal manner. An authenticator may also serve as an authentication server.

IEEE 802.1X message packets are identified by their `Mac.destinationAddress` values and their `ethertype` values. The `destinationAddress` values of interest are:

- `01-80-c2-00-00-00` (bridge group address)
- `01-80-c2-00-00-02` (port access entity (PAE) group address)
- `01-80-c2-00-00-0e` (link layer discovery protocol (LLDP) multicast address)

The `ethertype` value used for all 802.1X messages is: `0x888e`.

A forwarding system that supports IEEE 802.1X must be able to assign an attribute to each physical interface that indicates its authentication state. If an interface has authenticated a host, then that host is permitted to send and receive packets normally. Otherwise, that host is restricted to communicating with the authenticator using 802.1X messages.

A forwarding system's physical interface may be a shared medium interface or may be attached to a further forwarding system such that multiple Ethernet endpoints are accessible via a single physical interface. It is desirable in this case to be able to authenticate on a per-endpoint basis rather than on a per physical interface basis.

To support this kind of packet filtering, a forwarding system may employ a list of `Mac.sourceAddress` and interface ID values that represent the endpoints (supplicants) that have been authenticated. If a packet is received whose interface and `sourceAddress` values are found in this list, then the packet is forwarded normally. For transmit packets, only packets that are addressed to authenticated address/interface combinations may be forwarded to the affected interfaces (in other words, unknown-unicast packets must not be flooded to multi-host interfaces).

IP/MAC Binding

This rather weak form of security is intended to ensure that an IP packet is coming from an expected or trusted source. This is done by “binding” an Ethernet MAC address to an IP address. In other words, it is expected that the `Mac.sourceAddress` value of a packet is that of an originating endpoint or an upstream IP router and that there is a one-to-one or one-to-many relationship with an IP `sourceAddress` value.

There are many ways to implement this check, but they all involve performing a lookup on the IP source address. The result of that lookup may include in its associated data the one and only Ethernet source address associated with that IP source address or it may include a token or index that is used to compare against a comparable index or token returned from an Ethernet MAC source address lookup that is performed for MAC address learning and aging.

If a physical or logical receive interface is configured to require IP/MAC binding checks and there is a mismatch between the IP source address and the expected MAC source address, then the packet must be discarded.

Reverse Path Forwarding (RPF) Checks

Reverse path forwarding checks are another form of weak access control. Here, the intent is to ensure that a packet was received via an expected interface in order to prevent spoofing of an IP endpoint.

Looking up an IP source address is already required for common multicast forwarding applications, so a forwarding system may already have the means necessary to perform an RPF check. To facilitate an RPF check once an IP source address lookup is available, some means for providing a list of authorized receive interfaces via which the packet may be received must be provided. This list may consist of physical interfaces, logical interfaces or VLANs. Lists—instead of single source identifiers—are required in order to support ECMP. With ECMP, a packet from a single origin may arrive at a forwarding system via a variety of paths and, consequently, interfaces.

If an interface (or VLAN) is configured to require RPF checks and the packet’s receive interface or VLAN is not listed among the approved interfaces (as determined by an IP `sourceAddress` lookup), then the packet must discarded.

Access Control Lists (ACLs)

In their simplest form, access control lists return a binary permit/deny decision on each submitted packet. Many more actions—priority changes, forwarding domain changes, etc.—are possible and are common in modern forwarding systems. Access control lists are discussed in detail in their own chapter ("*Firewall Filters*" *on page 315*), and are discussed here simply to cover their security-related applications.

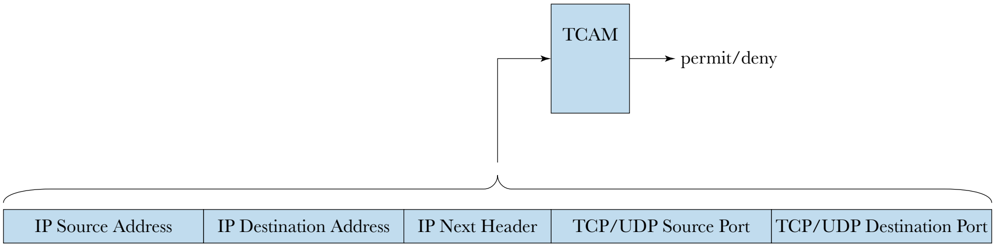


Figure 131 *TCAM-Based ACLs*

ACLs were initially developed as a spin-off technology during the development of experimental flow-based forwarding techniques using TCAMs (ternary content-addressable memories). These memories contain ternary keys (`0`, `1` and `x` values per bit) that can match against a concatenation of fields from a packet. A basic 5-tuple is commonly used: IP `sourceAddress`, IP `destinationAddress`, IP `nextHeader`, TCP or UDP `sourcePort`, and TCP or UDP `destinationPort`. With these values, it is possible to determine at a very high level the kind of communication being carried out between two IP endpoints. Network operators can configure access control lists (i.e., lists of rules) to specifically prohibit certain types of communications. For example: no FTP packets to or from certain IP prefixes.

Multiple, distinct keys in a TCAM may match any number of submitted search arguments. Consider a TCAM with the following keys:

- `01101110xxxx`
- `01101xxxxxxx`
- `011xxxxxxxxx`

If a search argument equal to `011011101100` is submitted to the TCAM, it will match all three keys. However, the TCAM can only return a single result. This is accomplished by sorting the keys in priority order; ensuring that the highest priority match is returned. Generally, the most specific key (i.e., the one with the fewest `x`'s) is made the highest priority key. Essentially, there is an implied “else” between each key:

```
if searchArgument == 01101110xxxx {action}
else if searchArgument == 01101xxxxxxx {action}
else if searchArgument == 011xxxxxxxxx {action}
else {defaultAction}
```

If none of the keys match a search argument, a default result is returned. This default may be either permit or deny—i.e., forward or discard.

Authentication and Encryption

Once an endpoint is authorized to have access to a network, it is important to ensure that the information being conveyed is authentic and private.

Data authentication performs three fundamental services. First, it ensures that the data is from whom the packets claim the data is from. Second, it ensures that the data has not been altered in any way. Third, it ensures that packets are short-lived (i.e., protection against replay attacks). Basically, authentication makes it possible to trust the data that you receive.

The principle behind authentication is quite simple. A secure hash is computed across the data to be authenticated (including the identity of the sender of the data) and then included in the packet. A secure hash is an irreversible mathematical function, meaning it is possible to compute the hash from the data, but not reconstruct the data from the hash. A secure key value is included in the hash computation, but is not made available in the packet itself. This secure key is, ostensibly, not known by or available to any attackers. Hence, if a packet's data is modified in any way, an attacker would not be able to compute a correct authentication hash. A mismatch between the hash value included in the packet and the hash value computed by the intended receiver makes it apparent to the intended receiver that the data's integrity has been compromised. A sequence number is also included in an authenticated packet to make sure that the contents of the packet can only be used once.

Data encryption, on the other hand, provides data privacy. Even if a packet is obtained by an unintended recipient, that unintended recipient would not be able to make use of the data. Encryption works by encoding messages using one of several encryption algorithms (e.g., AES) and one or more key values. Without access to these key values, it is, essentially, impossible to convert the encrypted message back to plain text.

Of course, it is not actually impossible to decrypt a message without access to its encryption keys. The key values are, after all, finite numbers. That means that they are guessable. It may be computationally expensive and impractical (i.e., time consuming) to guess key values, but not impossible.

> NOTE For an example of a truly unbreakable encryption method, consider the one-time pad. This method is brilliant in its simplicity. Paired lists of truly random numbers (using lottery ball techniques or similar methods) are generated that are at least as long as the messages to be encrypted. The random numbers are combined with a message's characters in a reversible fashion (modulo sums, exclusive ORs, etc.). The receiver, having access to an identical copy of the list of random numbers used to encrypt the original messages, is able to reverse the encryption. The "one-time" phrase in the method's name implies that each pad (i.e., list of random numbers) is used just once and is then destroyed. As long as the pads are kept secure, the encrypted messages cannot be cracked because no patterns can be detected in truly random sequences that are ever only used once.

Two networking standards operating at two adjacent layers provide authentication and encryption services. For Ethernet, the IEEE 802.1AE MACsec standard applies. For IPv4 and IPv6, the IETF RFCs 4301, 4302 and 4303 IPsec standards apply. Each are described in the following two sections.

MACsec

MACsec provides authentication and encryption services for Ethernet networks. MACsec is intended to operate on a hop-by-hop basis rather than end-to-end. This means that each packet is decrypted as it is received by an Ethernet bridge and is re-encrypted as it is transmitted.

Operating hop-by-hop offers some advantages over end-to-end. For example, by decrypting each packet as it is received by a bridge (or by a bridge/router system) the entire packet is made available as plain text and can, therefore, be used for proper load balancing and other operations that require a deeper inspection of a packet than just the forwarding header.

Other advantages of MACsec's hop-by-hop security model include the fact that it can be readily applied to certain MAC control protocols—e.g., link layer discovery protocol (LLDP), link aggregation control protocol (LACP) —and other protocols that have hop-by-hop behaviors such as the IEEE 1588 precision time protocol. The use of MACsec does not preclude the use of other security protocols that operate at a higher layer (e.g., IPsec).

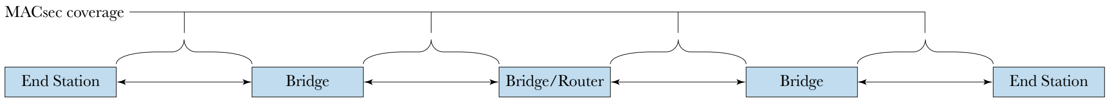


Figure 132 Hop-by-Hop MACsec Operation

MACsec operates within what are known as secure connectivity associations (CAs). Each connectivity association may have two or more Ethernet endpoints (end stations, bridges, routers, etc.) as members. For wired connections in modern, non-shared-medium networks, the number of endpoints in a connectivity association is two. For wireless networks where any number of wireless endpoints are connecting to, say, a wireless access point, the number of endpoints in a connectivity association may be quite large.

IEEE 802.1X link access control protocol is used to grant access to a connectivity association through a series of unidirectional secure channels (SCs). The secure channels are typically long-lived (think, a login session). However, within a long-lived secure channel, there may be a number of slightly overlapping, short-lived security associations (SAs) that are used to ensure that encryption key values are not used for dangerously long periods of time.

The identity of a packet's secure channel is encoded within the packet. For point-to-point links (e.g., full-duplex Ethernet), the leftmost six bytes (48 bits) of the secure channel identifier (SCI) is set equal to the source endpoint's `Mac.sourceAddress` value. The next 14 bits of the secure channel identifier serve as a port number. Shared medium networks such as EPON make use of this. The rightmost two bits are occupied by the association number. The association number is used to sequence through a series of short-lived secure associations. When a sender switches from the current secure association to the next secure association, it increments the two-bit association number by one. This informs the receiver that the next secure association is now being used and an appropriate decryption key must be used.

If the port number component of the secure channel identifier must be used, then the 64-bit (8-byte) `Macsec.secureChannelId` value is present in the MACsec header. Otherwise, just the two-bit association number is present in the header (the sender's `Mac.sourceAddress` value is, of course, always present in the encapsulating Ethernet header).

MACsec imposes a MACsec header between an Ethernet header and its payload. Like all well-designed protocols, MACsec is identified by the preceding header (i.e., the Ethernet header) with an `ethertype` value (`0x88e5` for MACsec). The MACsec header itself includes an `ethertype` value that identifies the type of the MACsec payload (e.g., IPv4, etc.). The IEEE, being particularly fussy about staying within Layer 2 (i.e., Ethernet, et. al.), refers to MACsec's header as a tag, diagramming it with an ethertype field at the start of the tag structure and being inserted between an Ethernet header's `sourceAddress` field and its `ethertype` field. In this book, the MACsec header, like a VLAN header, is treated like a proper header rather than a lowly tag.

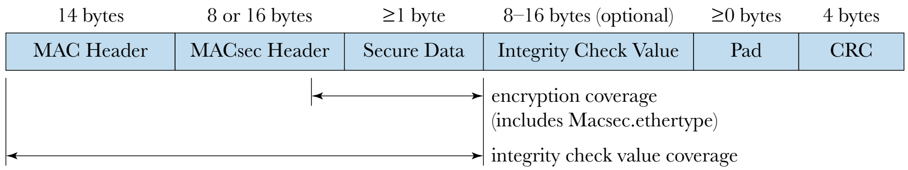


Figure 133 *MACsec Packet Structure*

As shown in Figure 133, MACsec's secure data region (i.e., the MACsec payload) is subject to encryption—along with the last two bytes of the MACsec header which encodes the `ethertype` value—to ensure that data's privacy. The Ethernet header and most of the MACsec header are not subject to encryption, which makes sense. If those headers were encrypted there would be no way to interpret the packet for decryption.

It is important to point out that, while the Ethernet header is in the clear, it's not really practical to forward MACsec packets without decrypting them first. This is because IEEE 802.1AE specifies that any VLAN header must appear *after* the MACsec header—i.e., in the secure data region. So, only very simple, VLAN-unaware forwarding is possible by Ethernet bridges that are not participating in the key exchange operations and are unable to decrypt the MACsec Ethernet packets.

The optional integrity check value is used to ensure the authenticity of the packet. The size of the integrity check value can range from eight to 16 bytes; its length being a function of the integrity check algorithm in use. The integrity check value is computed over all of the bytes that precede the integrity check value.

Data encryption is a functional superset of data integrity protection. If the packet's data is encrypted, then it is immune from useful (i.e., malicious) modification as well as eavesdropping. MACsec either provides modified, encrypted secure data, or integrity-checked data along with an inserted integrity check value.

The structure of a MACsec header is shown in Figure 134.

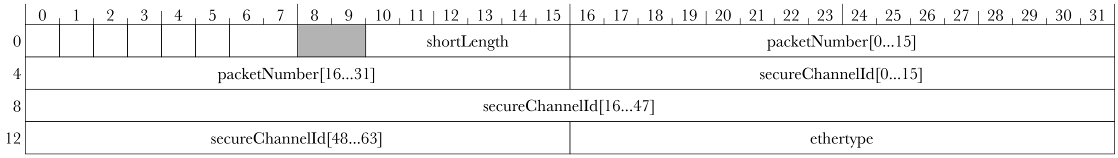


Figure 134 *MACsec Header Structure Diagram*

The fields that make up the MACsec header are defined in Table 60.

Table 60 *MACsec Header Field Definitions*

Field Name (std. name)	Width	Offset (B.b)	Definition
`version` (V)	1 bit	0.0	The version of the MACsec protocol. Must be set to `0`.
`endStation` (ES)	1 bit	0.1	Indicates that packet's `Mac.sourceAddress` value is to be used as a secure channel identifier (SCI). If this bit is set to `1`, then the packet is from an end station and the leftmost six bytes of the secure channel identifier are equal to `Mac.sourceAddress`.
`secureChannelIdPresent` (SC)	1 bit	0.2	Indicates that the 8-byte `secureChannelId` field is present. If this bit is set to `1`, then the overall length of the MACsec header is 16 bytes instead of 8 and that the packet's secure channel identifier is explicitly specified in the `secureChannelId` field. Otherwise, `secureChannelId` is not present and the MACsec header is just 8 bytes long.
`singleCopyBroadcast` (SCB)	1 bit	0.3	Indicates that the secure channel supports EPON single copy broadcast. If this bit is set to 1, the secure channel associated with the packet supports EPON's single copy broadcast, making it possible to send a single packet from an OLT that is received by all of the attached ONUs without having to set up separate secure connections.
`dataEncrypted` (E)	1 bit	0.4	Indicates whether encryption or just integrity is being provided. If this bit is set to 1, then the user data portion of the MACsec packet is encrypted. Otherwise, MACsec is only providing integrity assurances (i.e., authentication).
`changedText` (C)	1 bit	0.5	Indicates that the user data (i.e., text) has been changed. If this bit is set, then the user data has been modified due to encryption and `integrityCheckValue` is present.

Field Name (std. name)	Width	Offset (B.b)	Definition
`associationNumber` (AN)	2 bits	0.6	The packet's association number. This value is used to sequence among secure associations.
`shortLength` (SL)	6 bits	0.10	Identifies the location of `integrityCheckValue` for short packets. `integrityCheckValue` is alway immediately after the packet's user data. This means that, for particularly short packets, there may be some pad bytes between `integrityCheckValue` and the packet's CRC field. Hence, a length field is required to indicate where to find `integrityCheckValue`. This field indicates the number of bytes of secure data (which includes `macsecNextHeader`) as long as that number is less than `48`. If `shortLength` is set to `0`, then a secure data length of `48` or greater is presumed (i.e., no pad bytes present).
`packetNumber` (PN)	32 bits	0.16	Packet sequence number. This sequence number is maintained on a per-connectivity association basis and is incremented by `1` with each transmitted packet. It is used to detect replay attacks. To accommodate a certain amount of packet mis-ordering by the network, a `packetNumber` that is less than the expected value is tolerated up to some configurable threshold.
`secureChannelId` (SCI)	64 bits	4.16	Identifies the secure channel. A secure channel may be identified either by sourceAddress or this value. This field is only present in the MACsec header if `secureChannelIdPresent` bit is set to `1`.
`ethertype`	16 bits	12.16	Identifies the next header's type. This field is, essentially, an ethertype and is used to identify the type of the MACsec header's payload. This field is included in and protected by user data encryption.

The `dataEncrypted` and `changedText` bits are used to convey information about the state of the packet and how it's intended to be used. Table 61 defines all of the possible combinations of state of these two bits.

Table 61 *Encrypted and Changed Text State Interpretation*

dataEncrypted	changedText	Meaning
0	0	Integrity protection only, `integrityCheckValue` is present.
0	1	Integrity check only, but the data has been modified.
1	0	The packet is not processed by the security entity, but is reserved for the key agreement entity instead.
1	1	Data confidentiality (i.e., encryption) is being provided.

IPsec

One of the fundamental differences between MACsec and IPsec is their scope of coverage. MACsec is limited to covering individual links. IPsec, on the other hand, is capable of covering an end-to-end connection.

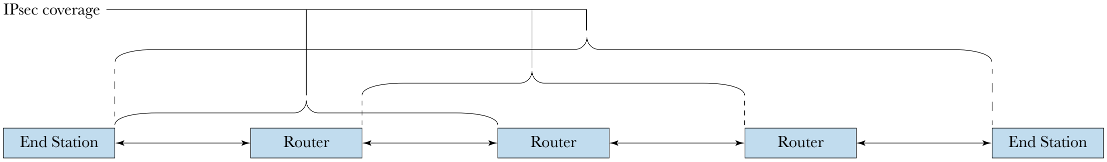


Figure 135 *IPsec Coverage*

As shown in Figure 135, IPsec can cover end station to end station (i.e., host-to-host or endpoint-to-endpoint) connections as well as connections between pairs of IP routers (known as security gateways) or between a security gateway and an end station. This range of scopes means that an end station (e.g., computer, server, etc.) need not implement IPsec in order to take advantage of the services afforded by IPsec for at least some portion of the path followed by the end station's packets. Ideally, however, IPsec is used end to end.

IPsec consists of two over-the-network protocols: the authentication header (AH) and the encapsulating security payload (ESP) protocol. The authentication header provides a means for conveying an integrity check value and it guards against data tampering, source spoofing and replay attacks. It does not offer data confidentiality through encryption. The encapsulating security payload protocol offers all of the services of the authentication header plus confidentiality through data encryption. Thus, the encapsulation security payload protocol is a superset of the authentication header. Though it is possible to do so, applying both an authentication header and using the encapsulation security payload protocol on the same packet is not practical or necessary. And, since the encapsulation security payload protocol can do everything that an authentication header can do, there's no strong motivation for supporting the authentication header. However, the authentication header is in use for the time being, so it's prudent to cover it.

Both the authentication header and the encapsulating security payload protocol support both a transport mode of operation and a tunnel mode. In the transport mode, the packet's original IP header is used to forward the packet from its origin to its destination. This means, of course, that the header must be transmitted in the clear so that intervening IP routers can interpret its contents and that certain fields in the header may be modified by the forwarding process (e.g., time-to-live values, checksums, etc.). In tunnel mode, the original IP packet is treated as the secure payload of an authentication header or the encapsulating security payload protocol—providing 100% security coverage of the original IP header—while a new IP header is prepended to the packet to actually forward it to either its ultimate destination or to the exit of the secure tunnel.

There is no opcode, flag or other explicit indication in either an authentication header or the encapsulating security payload protocol that specifies if the transport or tunnel mode is in use. It is simply a matter of decoding the next-header values in the relevant headers. If, for example, `Ah.nextHeader` holds a value that identifies a typical IP payload such as TCP as the authentication header's payload, then transport mode is in use (i.e., the authentication header is a payload of the original forwarding IP header). On the other hand, if `Ah.nextHeader` is set to a value that indicates that IPv4 or IPv6 is the next header, then tunnel mode is in use. The exact same rules apply to the encapsulating security payload protocol. In other words, transport mode is just simple, fundamental IP packet forwarding and tunnel mode is just simple, fundamental tunneling. There's nothing about either behavior that is security specific.

The use of transport mode by a security gateway—i.e., a router on the path between two IP endpoints—is allowed only when the IP source address (for packets originating at the security gateway) or the IP destination address (for packets terminating at the security gateway) belong to the security gateway system itself. In other words, the secure connection must originate or terminate at the security gateway router.

Generally, transport mode is associated with end-to-end secure communications between IP endpoints while tunnel mode is employed to offer secure communications between two security gateways.

Security Associations

IPsec makes use of security associations (SAs) to establish a series of one-way connections between participants in secure communications. For two endpoints to communicate with one another while taking advantage of the security features and services of IPsec, two reciprocal one-way connections must be established.

Security associations are identified by what's know as a security parameters index (SPI) in conjunction with the IPsec protocol type (authentication header or encapsulated security payload, each explained further below). Security associations are maintained in a database by an IP forwarding entity. This database uses the security parameters index, the IP destination address and (optionally) the IP source address as key to identify entries in the database.

A longest-prefix lookup is used to search against keys that are a concatenation of the following values from left to right: security parameters index, IP next-header field, IP destination address, IP source address. The longest and, hence, most restrictive or specific match is the one that is returned from the lookup operation. If no match is found, then the packet is discarded.

A successful lookup returns the various parameters required to validate or decrypt the associated packet. Among those parameters are: encryption algorithm selection, shared secret keys, etc.

Only those forwarding systems (e.g., IP routers) that are originating or terminating a secure connection need to perform security association lookups. For forwarding systems in the middle of a secure connection, it is sufficient to forward the packet in the normal manner using whatever headers and fields that are not encrypted.

Authentication Header (AH)

The authentication header serves a single purpose: deliver a cryptographically secure integrity check value along with the packet being protected. This check value ensures that the source of the packet has not been tampered with or obfuscated, that the contents of the packet's payload have not been modified, and that the packet is not part of a replay attack.

Described simply, the authentication header is inserted between an IP header and the IP header's payload. This is summarized in Figure 136.

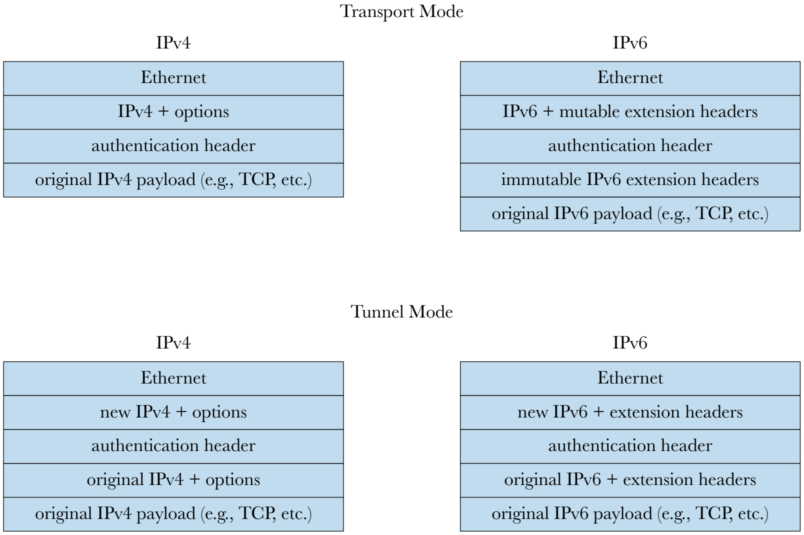

Transport Mode
IPv4
Ethernet
IPv4 + options
authentication header
original IPv4 payload (e.g., TCP, etc.)
IPv6
Ethernet
IPv6 + mutable extension headers
authentication header
immutable IPv6 extension headers
original IPv6 payload (e.g., TCP, etc.)
Tunnel Mode
IPv4
Ethernet
new IPv4 + options
authentication header
original IPv4 + options
original IPv4 payload (e.g., TCP, etc.)
IPv6
Ethernet
new IPv6 + extension headers
authentication header
original IPv6 + extension headers
original IPv6 payload (e.g., TCP, etc.)

Figure 136 Authentication Header Placement

The header stack-up for IPv4 is pretty straightforward. The IPv4 header (and all of its options) are followed immediately by the authentication header which is, in turn, followed by the original IPv4 payload. For IPv4 tunnel mode, simply place the authentication header and a new IPv4 header ahead of the original IPv4 header.

The header stack-up for IPv6 has to deal with IPv6 extension headers. These are considered headers in their own right instead of inseparable parts of the IPv6 header. The authentication header is, for all intents and purposes, yet another extension header (with some important differences that are described in the field definitions table). What makes IPv6 a bit tricky is that, in transport mode, the extension headers that are mutable (i.e., subject to being modified during forwarding) must precede the authentication header whereas the immutable headers are expected to follow the authentication header. The integrity check value in the authentication header (described in detail later) is expected to cover both the mutable and immutable IPv6 extension headers, while omitting the mutable fields from the mutable extension headers.

The structure of the authentication header is diagrammed in Figure 137.

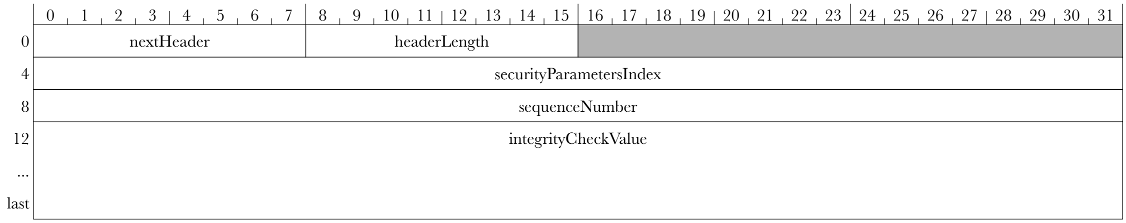


Figure 137 Authentication Header Structure Diagram

The fields of the authentication header are defined in Table 62.

Table 62 Authentication Header Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`nextHeader` (Next Header)	8 bits	0.0	Identifies the authentication header's payload type. If this value indicates that typical IP header payload is present (e.g., TCP), then transport mode is in use. If this value indicates that IPv4 or IPv6 is present, then tunnel mode is in use.
`headerLength` (Payload Length)	8 bits	0.8	Indicates the length of the authentication header. In the standard, this field is somewhat misleadingly named "payload length." In fact, this field indicates the length of the authentication header itself. The length of the authentication header is variable because the `integrityCheckValue` field is of variable length. The length of the header is expressed in 32-bit word units, minus 2. If, for example, `integrityCheckValue` is 96 bits wide (12 bytes) then `headerLength` is set to 4 (three 32-bit words for the fixed portion of the header plus three 32-bit words for the variable-length integrity check value, minus 2 yields 4). For IPv6, the length of the authentication header must be a multiple of 8 bytes in order to be consistent with IPv6 extension header requirements. This has two implications. First, `integrityCheckValue` may be followed by some number of pad bytes in order to fill out the header to a multiple of 8 bytes in length. Second, though IPv6 extension headers are supposed to indicate their length in units of 8 bytes (minus 2), this header indicates its length in units of 4 bytes. This means that any IPv6 forwarding entity that needs to parse beyond an authentication header must at least be aware that this type of header counts its size differently from all other IPv6 extension headers.

Field Name (std. name)	Width	Offset (B.b)	Definition
sequenceNumber (Sequence Number)	32 bits	8.0	A packet sequence number to detect replay attacks. This sequence number initialized to 0 when a secure association is established and is then incremented by 1 for each packet transmitted within a secure association. (The first packet transmitted in a newly-established secure association has an sequenceNumber value of 1.) With high-speed interfaces, a 32-bit number is entirely too narrow. For example, a 100 Gbps interface sending minimum-length Ethernet packets will roll over this counter in less than 30 seconds. (28.86, to be pedantic.) Hence, an extended, 64-bit sequence number is supported. However, the upper 32 bits of that sequence number is not conveyed by the authentication header. Instead, the upper 32 bits are agreed to and conveyed by the secure association management protocol. Though not included in the authentication header, the upper 32 bits of sequenceNumber are included in the integrityCheckValue computation. With the 64-bit extended sequence number in use, the counter rolls over using the parameters from the head of this paragraph.
integrityCheckValue (ICV)	varies	12.0	Authenticates a packet's contents and addressing. This value is derived by computing a secure hash value over the protected data. The protected data includes the outer IP header used for forwarding the packet (less fields that are normally subject to change), the authentication header itself (less integrityCheckValue) and the entire payload of the authentication header. The length of the integrity check value is dependent upon the algorithm used to check the value. The length of this field must be a multiple of 4 bytes (IPv4) or 8 bytes (IPv6). The length of the integrity check value is not a multiple of four or eight for all secure hash algorithms. Pad bytes are used to fill out this field to the required minimum length. Pad bytes always come after the integrity check value within the integrityCheckValue field. Pad bytes are included in the computation and checking of the integrity check value. Further padding may be necessary at the end of the packet in order to satisfy block size requirements of the secure hash algorithm that is in use. These pad bytes must always be zero-valued bytes. These pad bytes are *not*, however, included in the packet itself. They are only included in integrity check value computation and checking.

Field Name (std. name)	Width	Offset (B.b)	Definition
`securityParametersIndex` (SPI)	32 bits	4.0	An arbitrary value used to fetch security parameters. This value may be used by itself or in conjunction with the IP next-header value (i.e., authentication header or security encapsulation payload) to distinguish between the two protocols. This individual or combined value may then be further combined with IP destination and source address values and used in a longest-prefix match type lookup to retrieve the security parameters.

The following series of figures (Figures 138–141) show the coverage of the authentication header. All of the blue-shaded fields (including reserved fields) are included in the computation and validation of `Ah.integrityCheckValue`. `Ah.integrityCheckValue` itself (and any necessary padding) is set to zero during these computations.

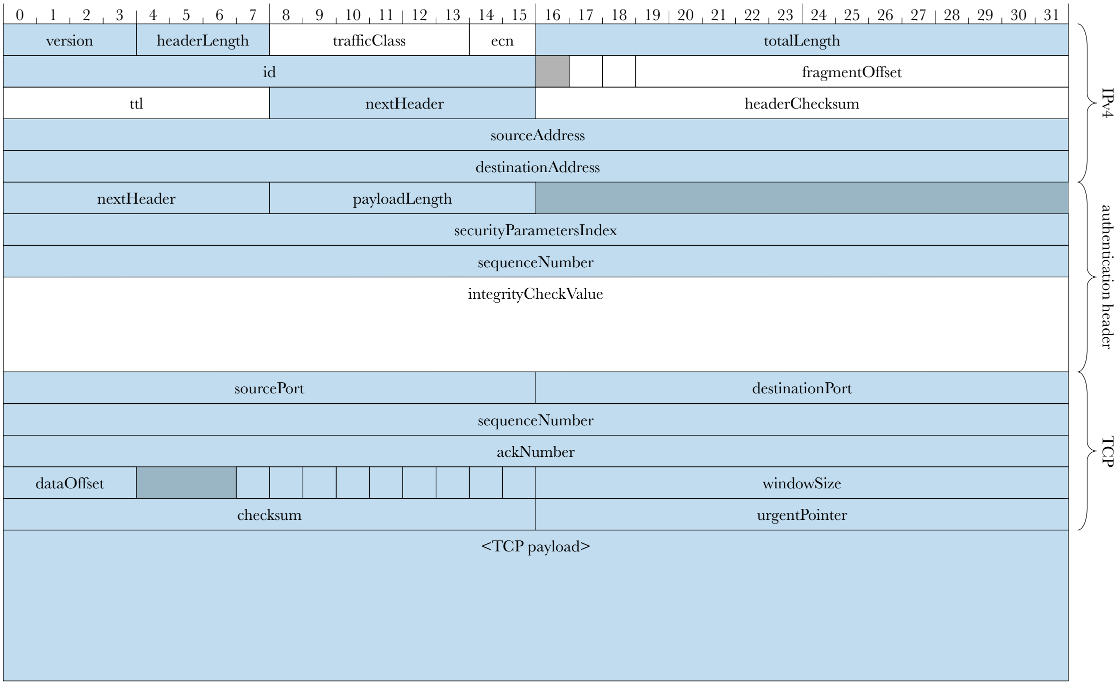


Figure 138 *Authentication Header Coverage: IPv4 Transport Mode*

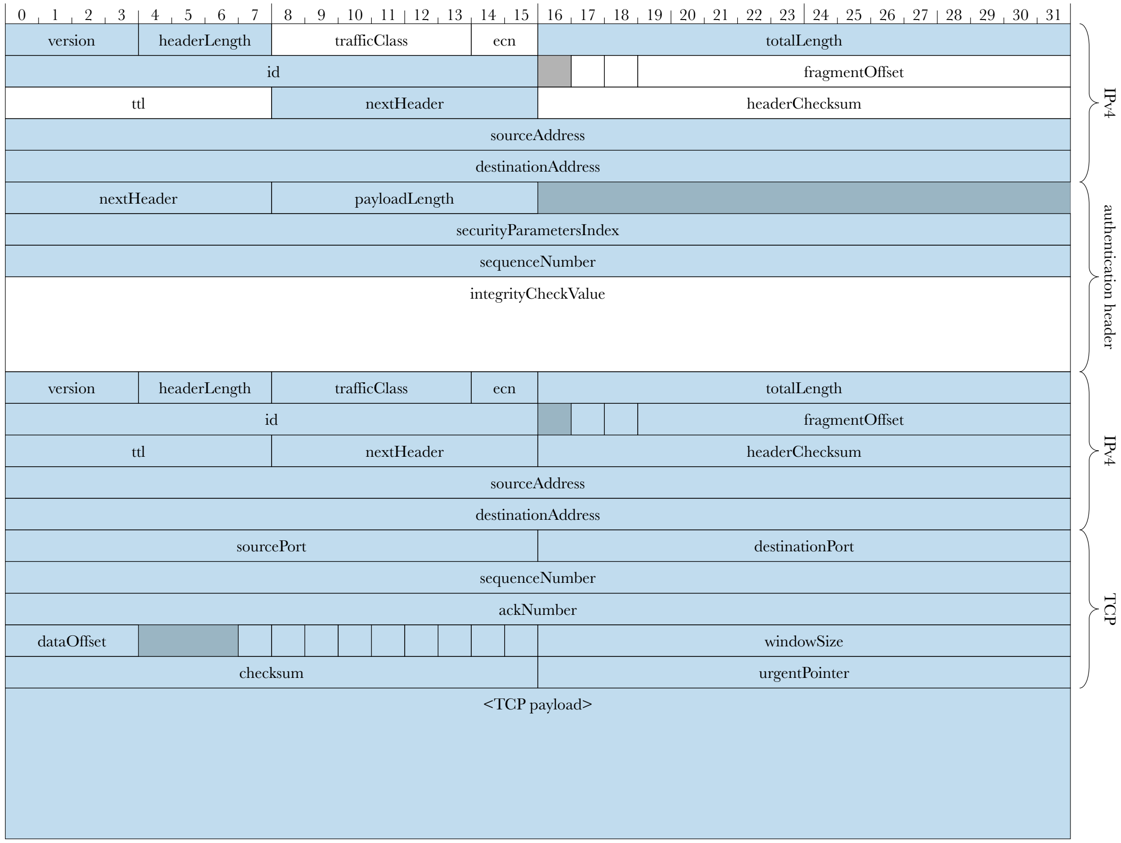


Figure 139 Authentication Header Coverage: IPv4 Tunnel Mode

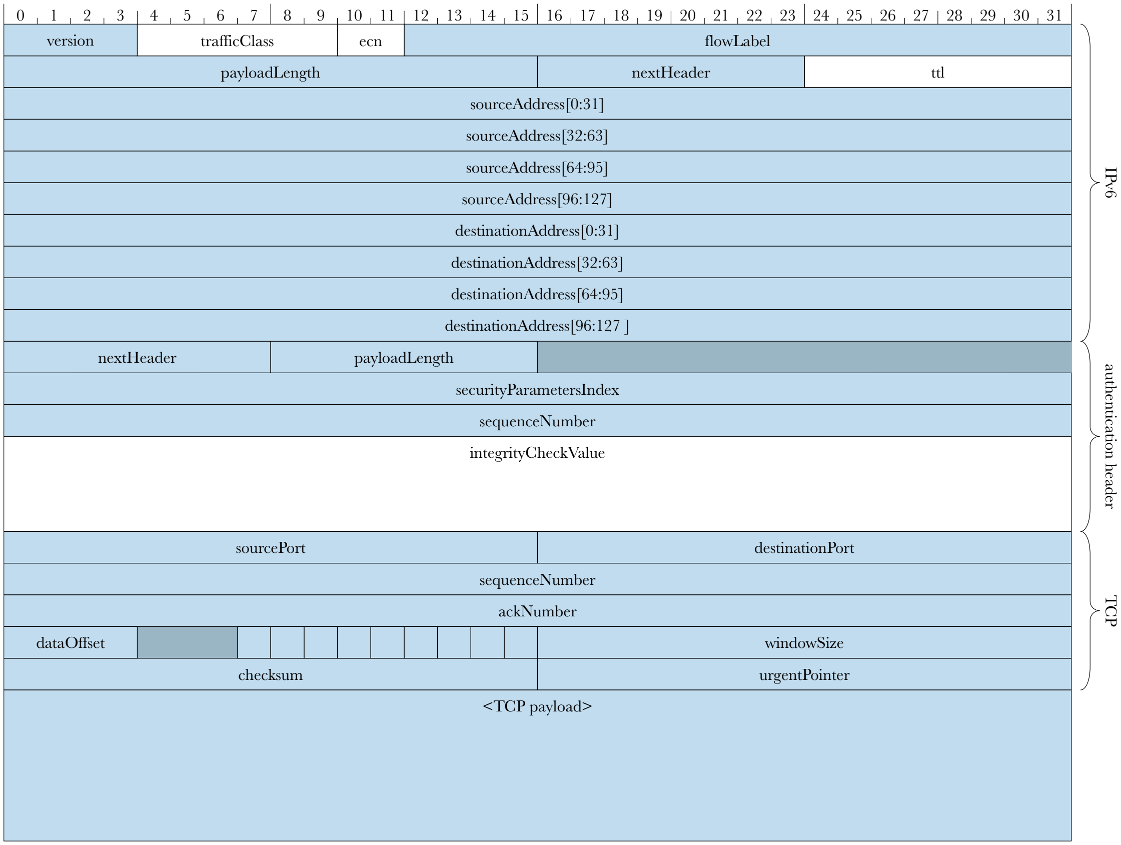


Figure 140 Authentication Header Coverage: IPv6 Transport Mode

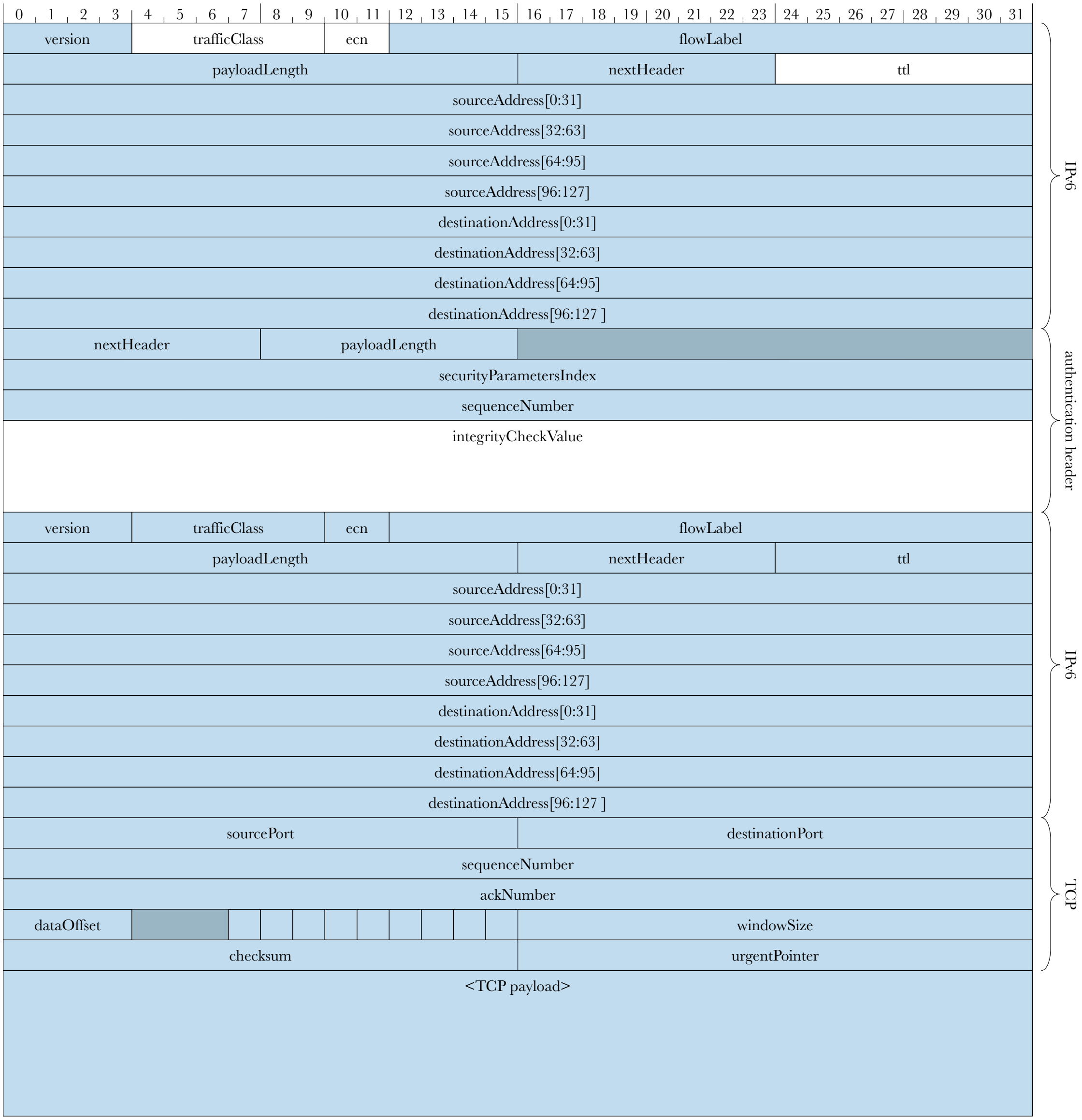


Figure 141 Authentication Header Coverage: IPv6 Tunnel Mode

Encapsulating Security Payload (ESP) Protocol

IPsec's encapsulating security payload (ESP, described in IETF RFC 4303) provides both authentication services and encryption services. ESP is far more complex than IPsec's authentication header (AH). This is so because ESP requires both a header component and a trailer component. In other words, the encrypted payload is surrounded by ESP components as shown in Figure 142.

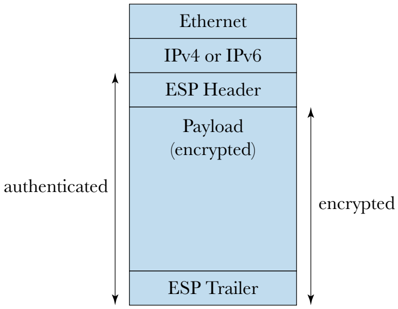


Figure 142 *IPsec Encapsulating Security Payload Packet Structure Diagram*

The ESP header provides the security parameters index (SPI) and sequence number information in the same manner as AH. The ESP trailer, on the other hand, provides padding, a next-header field and authentication data. The details of these headers are provided in Figure 143 and Table 63.

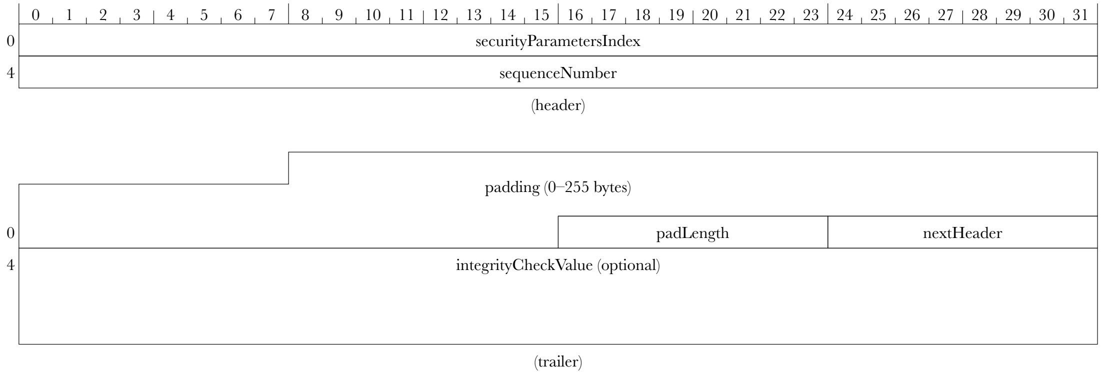


Figure 143 *IPsec Encapsulating Security Payload Header and Trailer Structure Diagram*

Table 63 IPsec Encapsulating Security Payload Header and Trailer Field Definitions

Field Name (std. name)	Width	Offset (B.b)	Definition
`securityParametersIndex` (SPI)	32 bits	0.0	An arbitrary value used to fetch security parameters. This value may be used by itself or in conjunction with the IP `nextHeader` value (i.e., authentication header or security encapsulation payload) to distinguish between the two protocols. This individual or combined value may then be further combined with IP destination and source address values and used in a longest-prefix match type lookup to retrieve the security parameters.
`sequenceNumber` (Sequence Number)	32 bits	4.0	A packet sequence number to detect replay attacks. This sequence number initialized to `0` when a secure association is established and is then incremented by 1 for each packet transmitted within a secure association. (The first packet transmitted in a newly-established secure association has a `sequenceNumber` value of `1`.) With high-speed interfaces, a 32-bit number is entirely too narrow. For example, a 100 Gbps interface sending minimum-length Ethernet packets will roll over this counter in less than 30 seconds. (28.86, to be pedantic.) Hence, an extended, 64-bit sequence number is supported. However, the upper 32 bits of that sequence number is not conveyed by the encapsulating security header. Instead, the upper 32 bits are agreed to and conveyed by the secure association management protocol. Though not included in the authentication header, the upper 32 bits of `sequenceNumber` are included in the `integrityCheckValue` computation. With the 64-bit extended sequence number in use, the counter rolls over using the parameters from the head of this paragraph.
`padding`	variable	variable	Pads payload data to a minimum multiple required by the encryption algorithm in use. Encryption algorithms often work on integral blocks of data. If the data to be encrypted is not a multiple of the algorithm's block size, then it must be padded out to fill in the gap.

Field Name (std. name)	Width	Offset (B.b)	Definition
`padLength` (Pad Length)	8 bits	0.16	Indicates the number of pad bytes. The length of the pad bytes must be sufficient to satisfy the encryption algorithm's requirements and to ensure that the overall length of the encrypted payload, including the `padLength` and `nextHeader` fields, is a multiple of 4 bytes. This is necessary because the optional integrity check value that follows `Esp.nextHeader` must align on a multiple of four bytes.
`nextHeader` (Next Header)	8 bits	0.24	Identifies the outermost header in the encrypted payload. This next-header value actually points back towards the head of the packet to identify the type of the encrypted payload that precedes the ESP trailer. For example, this value may be set to `6` to indicate TCP (transport mode), or to `4` or `41` to indicate IPv4 or IPv6, respectively (tunnel mode).
`integrityCheckValue` (ICV)	variable	4.0	Authenticates a packet's contents and addressing. This value is derived by computing a secure hash value over the protected data. The protected data includes the entire payload of the encapsulating security header. The length of the integrity check value is dependent upon the algorithm used to check the value. The length of this field must be a multiple of 4 bytes (IPv4) or 8 bytes (IPv6). The length of the integrity check value is not a multiple of 4 or 8 for all secure hash algorithms. Pad bytes are used to fill out this field to the required minimum length. Pad bytes always come after the integrity check value within the `integrityCheckValue` field. Pad bytes are included in the computation and checking of the integrity check value. Further padding may be necessary at the end of the packet in order to satisfy block size requirements of the secure hash algorithm that is in use. These pad bytes must always be zero-valued bytes. These pad bytes are *not*, however, included in the packet itself. They are only included in integrity check value computation and checking.

Since ESP supports both encryption and authentication, it is a strict superset of AH. Therefore, in common practice, ESP is generally favored over AH.

Denial of Service Defenses

The security methods and protocols described previously in this chapter have all focused on preventing unauthorized access to protected information or resources, or have ensured the privacy and/or integrity of data transmissions. However, these methods and protocols do very little to protect the network itself. A frequent cause of disruptions of services made available via the Internet are so-called “denial of service” (or DoS) attacks. The concept is absolutely simple: overwhelm a forwarding system or an endpoint with service request messages and that resource will be so busy trying to deal with all of these phony requests that legitimate requests face a very high probability of being dropped due to resource congestion. Often, attackers build huge armies of compromised zombie machines that unwittingly participate in simultaneous attacks. This is known as distributed denial of service (DDoS), where the total bandwidth of the attacking packets can easily exceed 1 Tbps. DDoS is particularly difficult to defend against because the rouge packets don’t all originate from a single IP address or even a single geographic region, making it pointless to try to filter these packets with a simple pattern match.

Access Control Lists

A common defense technique is to configure one or more rules in an access control list to detect and deny attacking packets. Finding patterns that are shared by the attacking packets—especially in a DDoS scenario—may be exceedingly difficult and is usually a painstaking manual process. Once an effective set of rules is in place, the forwarding system hardware should be effective at discarding attacking packets.

It is often necessary to examine the contents of the attacking packets well into their data payload to find some useful distinguishing characteristic since, when only the encapsulating and forwarding headers are examined, they may appear to be perfectly normal. Generally, forwarding systems and endpoints (i.e., servers) are ill-equipped for this kind of so-called deep packet inspection. Specialty firewall systems are generally required for such work.

Access control lists are discussed in further detail in *Chapter 18: Firewall Filters*.

Storm Controls

Sometimes, a network can be its own source of a denial of service attack. Ethernet bridging is particularly notorious in this regard. IEEE 802.1D transparent bridging (or its variants) relies on the rapid spanning tree protocol (RSTP, or its variants) to prune an arbitrary physical topology to a simple tree structure by selectively disabling links. The details of spanning tree are covered in *"Ethernet" on page 40*.

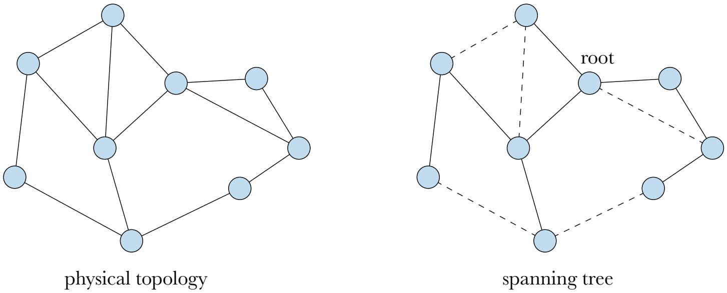


Figure 144 *Spanning Tree Created by Disabling Links*

However, all it takes is for a single Ethernet bridge that does not support the spanning tree protocol to be carelessly installed in the network, and a loop is formed that the spanning tree protocol can neither detect nor disable.

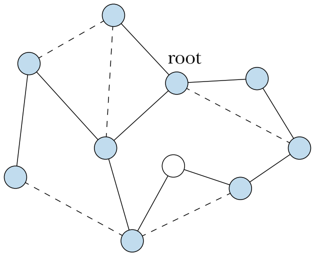


Figure 145 *Loop Created by Adding a Non-Spanning Tree Bridge*

Once a BUM packet (broadcast, unknown-unicast or multicast) is received by the non-spanning tree bridge, that packet will circulate around the loop and be flooded throughout the rest of the network until the loop is somehow broken. This is known as a broadcast storm. Broadcast storms force legitimate packets to compete for finite bandwidth and buffering resources, resulting is vast amounts of dropped packets.

Most forwarding systems that support Ethernet bridging include some form of broadcast storm detection and mitigation capability in the hardware. The operating principle of a broadcast storm control mechanism is quite simple. First, BUM packets are detected by examining `Mac.destinationAddress` directly and by checking for unsuccessful unicast destination address lookups. Any packet that falls into the BUM category is submitted to a dedicated rate meter. If the rate of BUM packets on a particular receive interface exceeds some threshold, then some fraction of those BUM packets are discarded. This will reduce the load on the network from the BUM packets so that non-BUM packets face less congestion. Of course, legitimate BUM packets will also get caught up by the storm control mechanism, but this is not easily avoided.

17 Searching

Searching for keys in a database is essential for many required behaviors of forwarding systems. Most obvious is the forwarding database, into which forwarding searches are performed. A typical forwarding search (or lookup) involves assembling a search argument using packet header data such as destination address or label values, metadata such as forwarding domains and receive interfaces, or, frequently, a combination of these elements. The assembled search argument is then used to find a matching key in the targeted database. There are two possible outcomes of a search: a hit or a miss. In the case of a hit, the so-called associated data that is stored along with the key in the database entry (or is at least pointed to by the matching entry) is returned as a byproduct of the search operation and is used to direct or influence the subsequent forwarding of a packet or other behavior of the forwarding system. In the case of a search miss, default associated data may be returned that is lookup-type or database specific, or the forwarding system may have automatic behaviors that are performed in the event of a database search miss.

There are a number of characteristics of the various search algorithms that make certain algorithms better suited for a particular application than others. The characteristics include:

- performance
- latency
- silicon or board area efficiency
- population density
- power efficiency
- algorithmic suitability

When considering performance and latency, it may seem as if one measure is simply the reciprocal of the other. In other words, if an algorithm can process one million lookups per second, then each lookup takes one millionth of a second. This, however, ignores the effects of pipelining. A number of common lookup algorithms require several steps to complete, and these steps may lend themselves to a practical pipeline design. A typical pipeline has some number of stages where each stage requires a fixed amount of time to complete its work and that time is the same for each of the stages. The performance of the pipeline is dictated by the number of operations that can be introduced into the pipeline per second. The latency is dictated by the duration of each stage's work and the number of stages. A pipeline is closely analogous to an assembly line in a factory; items being assembled move from station to station and specialized work is performed at each station.

The area of a search solution is dictated by two factors: the size of the required database (i.e., the number of database entries), and the efficiency with which each entry in the database may be accommodated in the database's storage elements. The size of the database is often dictated by the application (for example, forwarding protocol, network architecture, etc.). The area efficiency on a per-entry basis is a function of the search algorithm—certain algorithms require some degree of data redundancy or per-entry metadata—and the storage technology in use.

Population density doesn't refer to how full a table is. Rather, it refers to the ratio of the table size (i.e., its maximum number of entries) to the number space represented by the keys. For example, if a table has a total capacity of 1,024 entries and the key is 12-bits wide (4,096 possible values), then the table has a population density of 25% (25% of the possible values may fit into the table). Or, if a 1,024-entry table holds 32-bit keys (4 billion possible values), then the table has a population density of 1/4,000,000. Population densities greater than 100% indicate that the table is too large.

Power efficiency is becoming increasingly important. Consider a typical large-scale data center application. The silicon devices in the forwarding systems generate heat as they operate. This heat is a direct byproduct of the power that they consume. Silicon devices that consume more power require bigger power supplies. Power supplies are not 100% efficient, so they dissipate heat as well. The heat must be removed from the forwarding system's enclosure so that the components within don't overheat. This is typically done by using fans to force air through the system and out exhaust ports. The fans themselves require power, and generate heat of their own. Thus, a lot of hot air is expelled into the data center environment. This heat must be removed from the building. This is typically done using refrigeration-cycle cooling systems. These systems consume considerable amounts of power. The upshot of all of this is that the amount of power that is consumed by the components that actually do useful work—i.e., the silicon devices inside the forwarding systems—each have a substantial power multiplier penalty just to keep them within safe operating temperatures. Each watt saved within a piece of networking equipment may save one or more additional watts elsewhere. Different search technologies and algorithms have different power characteristics.

Finally, there's algorithmic suitability. For Ethernet bridging, an exact match search function is all that is required. However, for IPv4 or IPv6 routing, a longest-prefix match search function is required. Exact matches are generally quite simple, while longest-prefix matches are more complex. An algorithm that can perform a longest-prefix match can generally also perform exact matches, but not at the same levels of efficiency as an algorithm that only performs exact matches.

Table searching is vast field and worthy of a book all to itself. What follows is a brief survey of some fundamental search methods.

Indexed Reads

Calling an indexed read a search method is a bit of a stretch, but it deserves to be described since tables of this type are perhaps the most frequently encountered in forwarding system design. An indexed read is, essentially, a memory where the search argument is used directly as that memory's address input. In other words, the search argument points directly to the appropriate entry in the table without any manipulation of the search argument as in Figure 146.

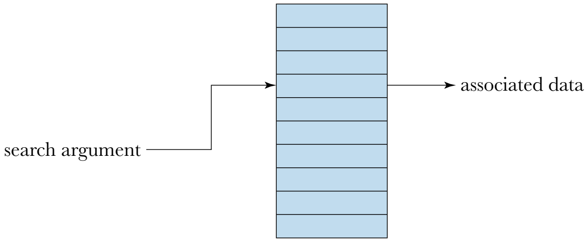


Figure 146 Indexed Read Table

This is, of course, the simplest possible table design. There's no need to store the key value in the database entries and there's no need for any kind of comparison logic because the search argument points directly to the desired entry.

In addition to being very simple, an indexed read table also has excellent performance and latency characteristics. It is only suitable for exact-match applications and it is ideally used where the population density is between, say, 50% and 100%.

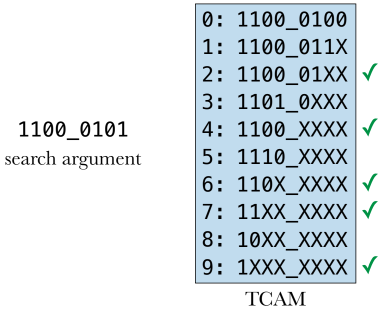
CAMs and TCAMs

CAMs (content-addressable memories) and TCAMs (ternary content-addressable memories) are specialized memories where every stored key in the table is compared with the submitted search argument in parallel.

In its most basic form, a CAM performs an exact match using binary keys. When a search argument is submitted, every row of the CAM's memory is read simultaneously and submitted to a comparison bus. The matching entry's location in the

CAM is produced as the CAM's output (i.e., data in, address out). Typically, the rows are prioritized such that matching rows with lower index numbers have higher priority than rows with higher index numbers. Therefore, if there are multiple keys that share the same value, the index of the key whose index value is lowest is returned as the match result. A binary CAM is only suitable for exact-match applications.

A TCAM extends the behavior of a CAM by encoding the keys with three-state logic: `0`, `1` and `X`. This greatly extends the utility of the CAM concept by enabling arbitrarily-masked matches. If there are bits in a search argument that are relevant for some key values and not relevant for others, the value `X` may be used in those bit positions in those keys where the match criteria is *don't-care*. Through careful key definition and sorting within the TCAM, longest-prefix matches may be easily performed. Consider Figure 147.



```
0: 1100_0100
1: 1100_011X
2: 1100_01XX ✓
3: 1101_0XXX
4: 1100_XXXX ✓
5: 1110_XXXX
6: 110X_XXXX ✓
7: 11XX_XXXX ✓
8: 10XX_XXXX
9: 1XXX_XXXX ✓
```

Figure 147 TCAM-Based Longest-Prefix Match

In Figure 147, the search argument matches all of the entries highlighted by the green check marks. The lowest-indexed match has the highest priority, and, since the keys are sorted such that those keys with the fewest masked bits (i.e., the longest prefixes) appear before those with more masked bits, the lowest-indexed match also corresponds with the longest-prefix match. In this example, that is the entry at index `2`.

While TCAMs are well-suited for longest-prefix match operations and arbitrarily-masked matches, they struggle a bit with arbitrary range matches, such as are commonly used with access control lists (ACLs) that include TCP or UDP port numbers in their rules. TCP and UDP port numbers are often expressed in ACL rules as numerical ranges such as 100 through 200, inclusive. Now, if the desired range was something like 96 through 127, a TCAM would be perfectly well suited since 96 (`0110_0000`$_2$) and 127 (`0111_1111`$_2$) exactly share a prefix. While, on the other hand, 100 (`0110_0100`$_2$) and 200 (`1100_1000`$_2$) do not share a simple prefix. To cover the range of 100 through 200 with a TCAM, a series of keys are required.

These keys are prefixes that fill in the range from 100 through 200 as follows:

- 100–103 (0110_01XX$_2$)
- 104–111 (0110_1XXX$_2$)
- 112–127 (0111_XXXX$_2$)
- 128–191 (10XX_XXXX$_2$)
- 192–199 (1100_0XXX$_2$)
- 200 (1100_1000$_2$)

Therefore, a single range requires six entries in a TCAM.

CAMs and TCAMs typically have excellent performance and latency characteristics owing to their parallel internal operation. They also have only moderately good storage density, which is hurt by the large storage cell sizes, but is helped by the simplicity of the storage structure (i.e., no unnecessary metadata).

The biggest challenge for TCAMs is their area and power costs, especially the more flexible TCAM. To store three-stage logic, two bits are required per key bit. And, the match logic in the TCAM must compare the search argument against thousands of keys simultaneously; this not only requires extensive logic and wiring area, it also requires a lot of power to activate all of those storage cells and comparison gates.

Because CAMs and TCAMs are so expensive on a per-bit basis, a database rarely stores each key's associated data within the CAM or TCAM elements. Instead, the search result from the CAM or TCAM is the index of the matching entry which is then used to perform an indexed read of an associated data memory. This increases the latency slightly, but does not affect performance.

Finally, a characteristic that CAMs and TCAMs share with most other search methods and technologies, is that they're suitable for very low population densities. It is not unreasonable to use TCAMs for 64-bit IPv6 longest-prefix match searches in tables of modest size.

Hashing

A hash is a one-way mathematical function that maps numbers from a large space to numbers in a small space. For an extremely simple example of a hash function, think of the remainder of an integer division operation. If, for example, numbers that range in value from 0 through 1,000,000 are divided by, say 23, the remainder of those division operations always fall into the range 0...22. A vital characteristic of a hash function is that it is repeatable. The same input value must always yield the same output value. This means that random or time-variant components must be excluded from a useful hash function. Ideally, a hash function provides an even distribution of results for a wide variety of input data sets. In other words, a histogram of the hash results is as close to flat as possible.

For the purposes of table searching, hashes are very powerful because it is very simple to map numbers from a potentially huge space (e.g., the 48-bit IEEE MAC address) to a much smaller space (e.g., a 14-bit hash result). The hash result may be used directly as an address to index into a memory that stores the database entries.

One of the characteristics of a hash that can be problematic is that hashes are imperfect. If a set of 1,024 randomly chosen 48-bit numbers is hashed to a set of 10-bit results, the odds of each input value mapping to a unique output value is vanishingly small. This is due to the birthday paradox. The birthday paradox states that, even though there are 365 possible birthdays within a year, the odds of two people in a group sharing a common birthday is 50% when the size of the group reaches just 23 people. Similarly, the odds of so-called hash collisions—the condition where two or more input values hash to the same output value—is extremely high even with fairly small data sets. This means that if keys are added to a forwarding database whose capacity is 365 entries, that the odds of encountering a hash collision is about 50–50 when adding the 23rd entry.

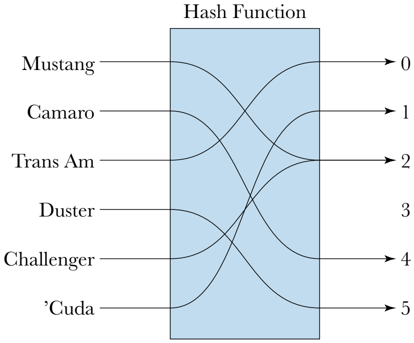


Figure 148 *Hash Collisions*

In Figure 148, a hash collision exists between the Mustang and Challenger inputs. An effective method to mitigate the effect of hash collisions is to store multiple keys per possible hash result, or "bucket" as they are commonly known. The idea is simple, if the example in Figure 148 allowed two entries per hash bucket, then the hash collision between the Mustang and Challenger inputs is easily handled. As the number of entries per bucket is increased, the number of buckets is decreased in order to maintain a constant overall table capacity. The benefits of multiple entries per bucket can be fairly intuitively visualized. Imagine taking a hash-based table that has a 16-entry capacity and morphing its aspect ratio so that, rather than 16 buckets of one entry per bucket, there's a single bucket with

16 entries. By doing so, we've effectively transformed the hash table into a CAM. And CAMs don't suffer from the effects of hash collisions. So, as the number of possible entries per bucket increases, the odds of unresolvable hash collisions decreases.

Another effective means for mitigating the effects of hash collisions is to use multiple hash tables with cuckoo[14] moves.

The idea is fairly straightforward. Two or more hash tables are maintained. Each uses a separate hash function (usually just a different hash coefficient). When performing a search, all of the tables are searched; a matching key being present in exactly zero or one of the tables. New keys are added to whichever table is less populated at the moment. If an attempt to add a key results in a hash collision (i.e., the intended bucket already holds a valid entry), then the existing entry is kicked out of its bucket and the new key takes its place. The key that was kicked out of its bucket is then moved to another table after being hashed by the new table's hash function. Here, of course, a new collision may occur that must be resolved in the same manner. Thus, a single attempt to add a new entry may result in a list of cuckoo moves. If each hash bucket can store more than one entry, then a list of cuckoo moves becomes a tree of cuckoo moves. As the population of the various hash tables continues to increase, the size of the cuckoo move lists or trees continues to increase, eventually becoming either impractically long or becoming circular (i.e., a subsequent move attempts to kick out the new entry that started the cuckoo move process). At that point, the hash table is declared full.

And that is, perhaps, the most significant downside of hash-based tables is that their capacity is non-deterministic. Achieving entry densities above, say, 85% can be challenging. So, when an equipment vendor claims that their hash-based forwarding database accommodates up to one million entries, the odds are that its capacity is actually closer to 850,000 (or less).

In summary, hashing has excellent performance and latency characteristics and good storage density, giving it good power and area characteristics. Hashing is suitable for exact-match searches where the population density is low.

Algorithmic Searching

All of the search methods described so far have had latency characteristics that are, essentially O(1), meaning that they have a constant time to completion regardless of the number of entries in the database or the width of the keys. In this section we'll explore more complex sequential algorithms.

[14] The cuckoo is species of bird with a most unusual child rearing method. The female cuckoo lays its eggs in the nest of a different species of bird whose eggs are very similar in appearance to that of the cuckoo's. A newly hatched cuckoo immediately sets about emptying the nest of its nest-mates—eggs and hatchlings alike. Thus, the non-cuckoo parent birds unwittingly raise the invasive cuckoo as if it were their own offspring; despite the vastly different appearance of a cuckoo chick compared to its own.

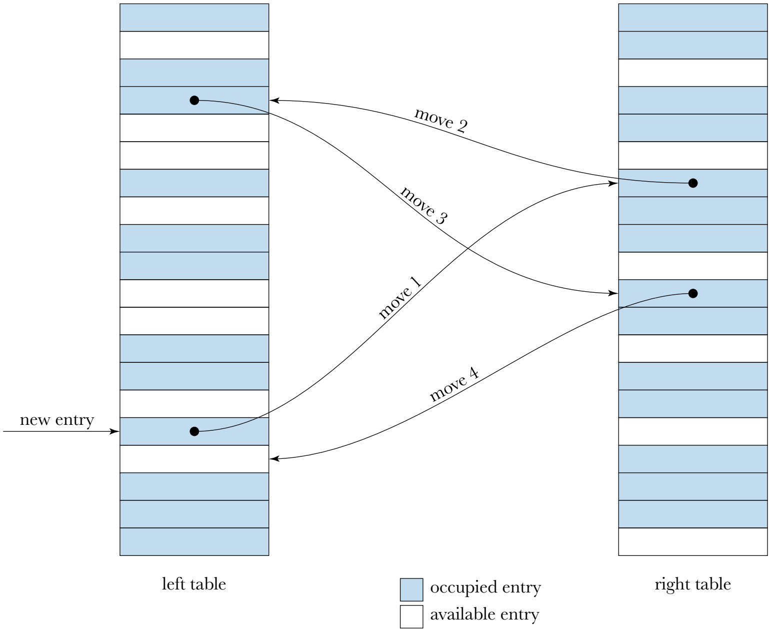


Figure 149 *Cuckoo Moves*

Trees

One of the first search algorithms most people learn is the binary search. The concept is exceedingly simple:

1. Start with a numerically-sorted table of keys; declare the list of keys to be the range of keys under consideration.
2. Identify the midpoint of the range.
3. Compare the search argument against the key found at that midpoint.
4. If the search argument matches the key value, the index of the matching location is the search result; terminate the search.
5. If the search argument is greater, consider only the range of keys that are numerically greater than the search argument and less than endpoint of the current range; continue with step 2.
6. Otherwise, consider only the range of keys that are less than the search argument and greater than the start point of the current range; continue with step 2.

As a binary search progresses, the range of keys under consideration is halved with each iteration. Thus, the search latency is approximately $O(\log_2 n)$, where n is the number of keys in the table. Meaning, of course, that the search latency is a function of the size of the table, or, more correctly, the number of valid keys in the table.

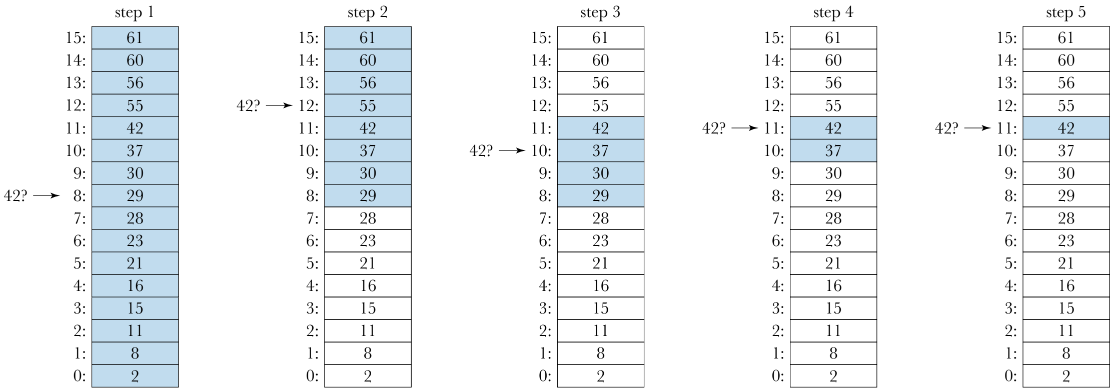


Figure 150 Binary Search

A concrete example of a binary search is shown in Figure 150. In this example, we have a numerically sorted list of keys and we're submitting a search argument of 42. The first table has a key count of 16. Dividing that by 2 yields an initial key pointer value of 8. We test our search argument (42) against the key stored at index 8 (29). Since 42 is greater than or equal to 29, we restrict the next phase of the search to the upper half of the list of keys. There are 8 keys in this portion of the table. Dividing 8 by 2 gives us 4. Thus, 4 is used as the offset from the base of the active region, pointing us at entry 12. The 55 at 12 is compared to 42. The less-than result means that we now consider only the lower half of the current region: just 4 keys. Thus, we apply an offset of 2 to our base of 8 to point at the key at index 10. The comparisons and halvings of the number of keys continues until we eventually arrive at our matching key. Of course, if our search argument had been 29, we would have found an exact match at the first step. Conversely, if our search argument had been 31, we would have reduced the size of the search region to just one key without finding a match; returning a "not found" search result.

If a binary search is performed entirely within a single physical table, then its performance scales in the same manner as its latency: as roughly the log of the table's size. However, a binary search actually lends itself to a pipelined implementation where the performance is O(1). The trick is to spread the keys across a series of tables where the first table holds just one key and each subsequent table holds twice as many keys as the one before. In each table, each entry represents the midpoint of the range of keys that corresponds to the stage of processing. For the first table, there's just one midpoint since its the midpoint of the total set of keys. For the second table, there are two midpoint keys: one for the upper keys from the total set of keys and one for the lower keys.

In the concrete example shown in Figure 151, we are once again using 42 as our search argument. Starting at the leftmost table, we compare 42 to the key at index 0: 29. The result is "greater than or equal," so we mark that first result as a 1 (it would be 0 if the comparison yielded "less than").

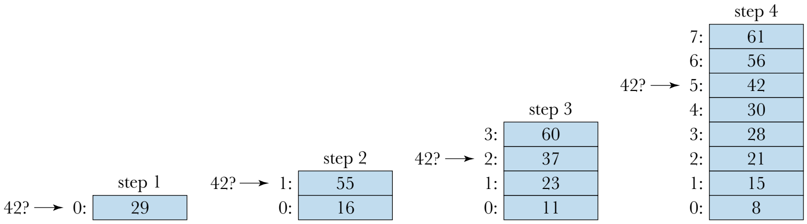


Figure 151 *Pipelined Binary Search*

Proceeding to the next table, we use the result from the previous test (the 1) as our index to fetch the 55 key. Here, the comparison with 55 yields a 0 result. The 0 is concatenated with the 1 from the previous step to gives us an index of 10_2 (2) for the next stage. Here, we compare our 42 with the key at index 2 (37), giving us a 1 result (for "greater than or equal"). Appending the new 1 result to the previous 10_2 result, gives us 101_2 (5) as the index for the next stage. Here, we compare our search argument (42) with the key at index 5 (42) and find an exact match; terminating the search with a successful result.

Sharp-eyed readers will note that the preceding pipeline binary search example had just 15 keys instead of 16 as in the prior example. For simple implementations, the total key capacity of a pipelined binary search is 2^n-1, where n is the number of stages.

Another instructive exercise is to improve upon a binary search's $O(\log_2 n)$ latency. This is easily done by increasing the radix of the tree from binary to, say, 4-ary. With a 4-ary search, a four-way decision is made at each stage of processing; meaning that each stage divides the size of the set of keys under consideration by 4 instead of 2. To make a four-way branch decision, three keys must be tested at each stage instead of one. A search argument may be in one of four states relative to a set of three numerically-sorted keys:

1. less than the first key;
2. greater than or equal to the first key and less than the second key;
3. greater than or equal to the second key and less than the third key;
4. greater than or equal to the third key.

By making a four-way decision at each stage instead of a two-way, the search latency improves from $O(\log_2 n)$ to $O(\log_4 n)$. Of course, higher radix values are possible. The limitation becomes the total width of the keys that must be fetched and compared with each stage of processing.

Though the binary search algorithm requires a fairly small amount of work to execute a search operation, adding new keys to the table can require a lot of work. This is so because the algorithm depends on the keys remaining numerically sorted. If a key with a low numerical value must be added to the table, it may be necessary to, in the worst case, shift every key to an adjacent location in order to open up a gap in the table to accommodate the new key. That is a fundamental challenge faced by all balanced-tree data structures.

One of the more remarkable characteristics of this form of tree-based search (binary or other) is that it is suitable not just for exact match searches, but also arbitrary range matches. To perform a range match, it is simply a matter of inserting two keys in the database for each range: a key that corresponds to the start of the range and a key that corresponds to the end of the range.

And, of course, a prefix is simply a special case of a range. For example, the IPv4 prefix 192.168.0.0/16 is the range 192.168.0.0 through 192.168.255.255. When searching for the longest prefix, what is really being searched for is the shortest range. Any set of keys that all share common prefix values of various lengths must be nested and not overlapped. This makes it unambiguous which length prefix is being matched when testing a search argument against a number of nested ranges.

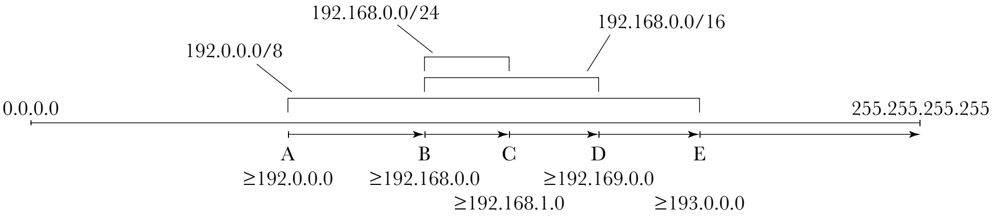


Figure 152 *Prefix Ranges*

Consider the example in Figure 152. Here, three nested ranges are shown (not to scale). If a search argument of 192.100.1.2 is submitted, it will match key A, indicating that the matching prefix is 192.0.0.0/8. Meanwhile, a search argument equal to 192.168.0.200 matches key B, indicating that the search argument falls into all three prefix ranges. However, since key B represents the shortest range (i.e., the longest prefix), then the matching prefix is 192.168.0.0/24. A search argument of 192.168.37.37 matches key C. Finally, a search argument equal to 192.200.0.1 matches key D. This is noteworthy because key A and key D both represent the exact same prefix: 192.0.0.0/8. Thus, a match with either A or D must return the same resulting associated data.

While much more complex than the simple hash-based algorithms previously described, tree-based searches have the benefits of being suitable for a variety of match types (exact, longest-prefix, arbitrary-range) and of offering deterministic key storage capacity. The search latency does increase with the number of keys and full-width keys must be stored and compared at every stage. In the next section, we'll explore a class of search algorithms that only examines a portion of the search argument at each stage.

Tries

Trie[15] structures originated in the dictionary search problem space. It was recognized that a lot of words share common prefixes in their spelling. Rather than storing (and testing) every letter of the sequence M-U-L-T-I-P-L several times when storing keys and testing search arguments against "multiple," "multiplication," and "multiply," simply store that common prefix just once and then branch to different key suffixes once reaching a point of divergence in spelling.

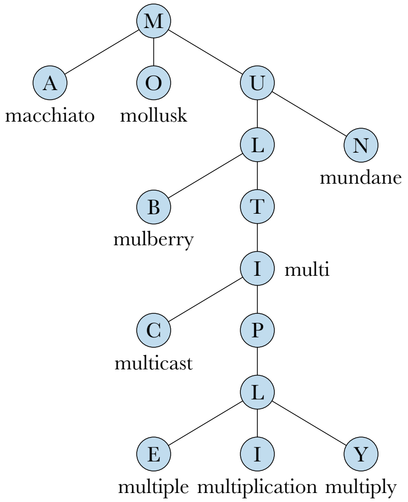


Figure 153 *Trie Structure*

In Figure 153, a handful of dictionary words are arranged in a typical trie structure. Letters from a search argument are considered one at a time, from left to right across the search word. The first letter (M) has three branch options based on the search word's second letter. If the second letter is A, then the only key in the dictionary that's a possible match is "macchiato." Following the U branch leads to a number of further keys. One characteristic of a trie is that each node along a path shares a common prefix, that prefix being longer and longer the further along the path that the key is located. For example, "mulberry" and "multicast" both share the "mul" prefix.

Another characteristic of a trie structure is the relative efficiency with which keys are stored. In the example above, nine words totaling 74 characters are stored using 15 single-character nodes. For longest-prefix matches, that's all the key storage that is required. For exact matches, the completions of each string must be stored.

[15] One of the most confounding aspects of a trie is its pronunciation. The name originated as the "trie" from the middle of "retrieval," implying it is pronounced like tree. Unfortunately, a number of authors have taken to pronouncing it like try in order to distinguish it in conversation from tree-based searches, contributing nothing but confusion.

For example, the C node above “multicast” must store the letters “ast” in order to exactly match a search argument against the key “multicast.”

Now, the key storage efficiency is offset to some extent by the need to explicitly point to next-nodes. With the binary search described previously, it is possible to algorithmically determine the location of the next node to test. With a trie, each node must encode the locations of the next nodes. When implementing a trie to search for a longest-prefix match of, say, an IP destination address, each node may hold just a single bit of the key. Thus, a simple two-way branch is possible for navigating to the next node. This, of course, means that two multi-bit node location values must be associated with each node.

Notice in the example above how certain nodes have just a single child node. For example, the T node’s only child is the I node. This means that the only valid sequence of letters in that part of the key is “TI.” This represents an opportunity to compress the structure to some extent by storing T and I together in a single node. This is known as a radix trie; the PATRICIA trie (Practical Algorithm to Retrieve Information Coded in Alphanumeric) being the canonical example of the form.

Tries have latency characteristics that are approximately O(m) where m is the number of bits in the keys. As for performance, tries do not lend themselves to pipelined implementations as neatly as do trees. A tree generally supports 2x or 4x table scales with each successive stage of processing. Tries generally store all of their nodes in a common, fungible memory system, mandating multiple read accesses of that memory system for each search.

Tries lend themselves beautifully to longest-prefix matches. In fact, the longest search process need not last any longer than the longest prefix in the structure, which may be far shorter than the search arguments. Tries can also perform exact matches. A trie can be applied to an arbitrary range match by replacing the equality comparison at each node with a magnitude comparison.

Others

There are, of course, many more search algorithms that have been described and I am confident that new ones will be developed for some time to come. Each algorithm has its own particular strengths and weaknesses. There is no such thing as an ideal, universal search algorithm. There are simply algorithms that are better suited for certain applications than other algorithms.

18 Firewall Filters

Sometimes, source and forwarding contexts, and destination information just isn't sufficient to adequately determine how (or whether) to forward a packet. This is where firewall filters—otherwise known as access control lists—come in.

Access Control Lists

Access control lists (ACLs) are an outgrowth of early work with ternary content-addressable memories (TCAMs). TCAMs were originally applied to the longest-prefix match problem for IPv4 forwarding. Later, multi-tuple lookups were used to try to reduce all of the forwarding processing to just a single step. In other words, if we care about packet attributes X and Y and Z when we forward packets of a particular type, let's just concatenate X and Y and Z into a single, multi-tuple search argument and submit that to a search within a TCAM, relying on the TCAM to return associated data that provides the required parameters for forwarding the packet. Unfortunately, for all but the smallest of networks, forwarding packets in this manner suffers from severe cross-product problems. Not only do you have to have an entry in your TCAM for each value of X that you are concerned with (let's presume that X stands in for a destination address value), you also have to have an entry for each value of Y in combination with each value of X that is of interest (that's X times Y) and every value of Z that's of interest (that's X times Y times Z). It doesn't take too many of those multiplications to reach database sizes that are wildly impractical.

While using ordinary TCAMs for so-called flow-based forwarding proved to be rather impractical, the method of performing multi-tuple lookups within a TCAM is very effective when applied to ACLs. ACLs may have thousands or tens of thousands of interesting combinations of values of fields rather than the tens or hundreds of millions of entries required for comprehensive flow-based forwarding.

An access control list, in its most fundamental form, is exactly what it says in the tin: a list that controls access. Think of it as the guy with the clipboard at the velvet rope outside of an exclusive nightclub: if you're not on the list, you'll be denied access.

In networking, an access control list is a sorted list of rules. The rules are tested in order, one after the other. Once a rule is found that matches the packet being tested, the processing of the list is stopped. This gives highest priority to those rules that are closer to the start of the list. Each rule is associated with one or more actions. The most fundamental action is: permit vs. deny. Other actions can include making changes to a packet's priority, forwarding domain, receive interface, and other metadata values. A matching rule's actions may also specify exactly how to forward a packet (e.g., transmit interface, required encapsulations, etc.). If you'll forgive a bit of a pun, there are no rules when it comes to designing rules or their associated actions.

Typically, the purpose of an ACL is to identify the kind of networking application associated with a packet. Examples of applications include viewing web pages, transferring files, watching streaming videos, etc. Network operators may want to apply rate limits to certain applications, shunt other applications to specific paths through a network, or simply prohibit (i.e., deny) other types of applications entirely.

Applications are typically identified by examining a packet's addressing, receive interface, forwarding domain and TCP or UDP (or similar) port numbers. For example, certain untrustworthy websites may all share a common IP address prefix and rely on HTTP (TCP port 80) to do their dirty work. By crafting an ACL rule that denies such traffic, a modicum of network security is realized.

The prioritizing behavior of a TCAM expresses itself in the design of a typical set of ACL rules in that the rules are expected to be evaluated in order, with the most narrowly-defined rules appearing first in the list and more loosely-defined rules appearing later. For example, a rule near the start of a list may permit packets that originate within an organization's network (i.e., locally-scoped IP source address) to access FTP servers from a particular public IP address. Meanwhile a later rule may deny packets that originate within an organization's network from accessing FTP servers, regardless of their IP address. Clearly, these two rules are inconsistent with one another: one permitting FTP while the other denies it. However, since the rules are prioritized and appear in a particular order, the resulting behavior is: "deny FTP packets unless they're addressed to this particular IP address."

Multi-Tuple Lookups

TCAMs represent just one possible means for implementing an ACL function in hardware. There are many others. What all of the methods have in common is that they must solve a multi-dimensional space intersection problem.

Consider a conventional range, longest-prefix or exact match. The solution space can be arranged along a number line as shown in Figure 154.

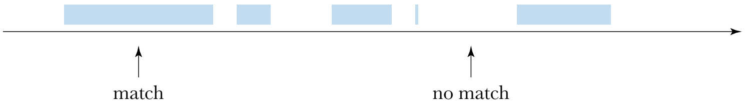


Figure 154 *Single-Dimension Match*

It is impossible to draw a single-dimension space, so rectangles are used in Figure 154 just for the sake of convenience. The heights of the rectangles convey no information. The behavior is easy to understand. Any particular search argument that consists of a single tuple (e.g., a single field from a header) must fall somewhere along the number line. The position where it falls either corresponds with the range or exact value specified by a key, or it doesn't. Hence, there's either a match, or there isn't.

Extending the problem to two dimensions yields Figure 155.

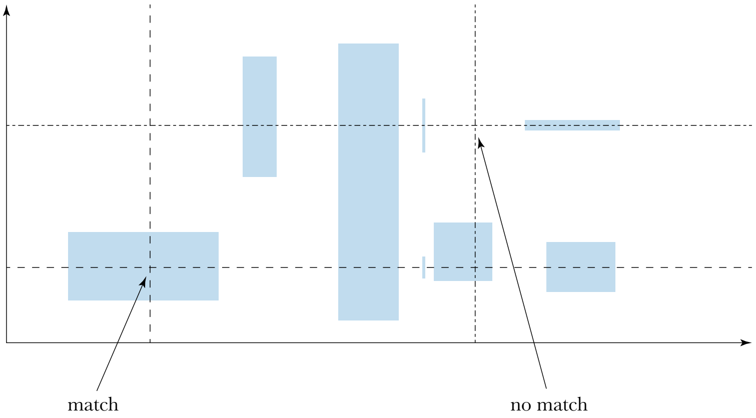


Figure 155 *Two-Dimension Match*

With the two-dimension match, two fields from either the same or different headers (or metadata) are used to build a multi-tuple search argument. If each of the two-tuples are submitted to separate searches along their respective number lines, the rectangles shown in Figure 155 correspond to mutual-match areas. The two intersecting dash-dash lines represent the two values associated with one two-tuple search argument whereas the dash-dot lines represent the two values associated with different values from the same two fields. The intersection of the dash-dash lines falls within a blue rectangle, meaning that both tuples of the search argument have matched relevant keys. Meanwhile, the dash-dot lines intersect away from all of the rectangles, meaning that the search failed to find a mutual match.

Graphical depictions beyond two dimensions exceed my skills as an illustrator, so you'll have to use your imagination. That being said, there is no real limit to the number of dimensions that may be considered when performing the multi-tuple matches associated with ACLs.

One way to solve the ACL matching problem without using TCAMs is to perform a series of individual lookups: one for each tuple. Each lookup returns a list of the descriptions or identifiers of the multi-dimensional space through which the single-dimensional search argument passes. This is repeated for all of the tuples. It is then a matter of finding the intersection of those lists to find the one (or more) multi-dimensional space which encapsulates all of the tuples. That space is associated with the highest-priority rule that uses that space in its specification.

Yet another method is to invert the problem and have each of the separate single-tuple lookups return a list of all of the ACL rules that have terms that match that specific tuple. It is then a matter of finding the intersection of the lists of rules to find those rules that match all of the tuples.

Actions

ACL actions are simply behaviors that are associated with a rule whose terms match the packet under consideration. The most fundamental action is: deny. The deny action simply means, “discard this packet.” The deny action is used when a packet is determined to be associated with a connection or application that is deemed to be in violation of usage guidelines or may represent some kind of malicious behavior.

Other actions may permit a packet to be forwarded, but may update some aspect of the packet's metadata or actual header field values. For example, a matching ACL rule may raise or lower a packet's priority. The effects of that priority change may be purely local (meaning that it is just the packet's metadata and the system's queuing behavior that are affected), or may be long-lived (meaning that the packet's priority-encoding fields are updated so that all subsequent forwarding systems will grant the packet the same priority treatment). Other actions may specify that a packet be copied and forwarded to a specialized traffic monitoring function for statistical or forensic analysis.

ACLs Relative to Other Packet Processing

A forwarding system may have thousands of ACLs configured. Only a small number of those ACLs may be associated with any particular packet. ACLs may be associated with a packet's receive interface, transmit interface or with any of the forwarding domains or logical interfaces that a packet adopts as it exits and enters various tunnels. Receive interfaces and tunnel terminations (i.e., tunnel exits) are encountered during ingress processing whereas transmit interfaces and tunnel originations (i.e., tunnel entrances) are encountered during egress processing. If ACLs may be associated with aspects of both ingress and egress processing, where, exactly, should ACLs be processed?

Strong arguments can be made for three points along a typical packet forwarding series of events:

1. Between tunnel termination and destination determination;
2. Between destination determination and tunnel origination;
3. Between tunnel origination and the transmit interface.

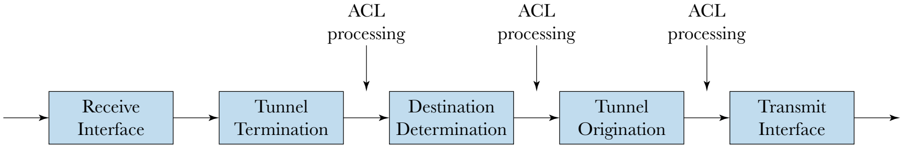


Figure 156 *ACL Processing Placement*

In the first position, the receive interface and all of the various forwarding domains that have been associated with the packet are known, so it is a simple matter of using those pieces of receive context information to determine which (if any) ACLs apply to the current packet. This position is also beneficial because any of the actions performed by the ACL function that update the packet's metadata will be reflected in the outcome of destination determination. For example, if an ACL action modifies a packet's forwarding domain, that new forwarding domain value may be used by the destination determination process to affect the outcome.

The one major shortcoming of the first ACL processing position is that the packet's transmit interface is not yet known, so it is impossible to check the packet against any ACL associated with transmit interfaces. That's where the second ACL processing position becomes interesting. It is positioned after destination determination, so any actions that affect a packet's forwarding behavior have no value, but ACLs that are associated with transmit interfaces may be applied to the packets.

In large forwarding systems that may have multiple packet processing functions interconnected by some form of fabric, a cross-product scaling problem occurs when egress-context processing is performed during ingress processing. Access control lists whose rules specify things such as, "packets matching these criteria must not be transmitted via interface X," are, of course, transmit context specific. Each transmit interface may have its own set of ACLs. If an ACL function in the second position (which is on the ingress side of a forwarding system's fabric), must implement ACL behaviors that are associated with transmit interfaces, then it must have a set of ACLs for each transmit interface. In large systems, this can be prohibitively expensive. Hence, the benefits of ACL processing at the third position.

Of course, the same cross-product scaling issues exist at the third ACL processing position if those ACL functions are expected to implement ACLs that are associated with receive interfaces. In practice, it is common for forwarding systems to have the ability to execute ACL behaviors at all three of the points shown in Figure 156 on page 319.

19 Routing Protocols

With the exception of the discussion on transparent bridging's source address learning behavior, little has been said about out how forwarding databases get their contents. How exactly does a group of largely independent forwarding systems cooperate to determine the structure of the network to which they are attached and how are optimal paths through that network computed? This is where routing protocols come in.

While the so-call forwarding plane of a forwarding system is responsible for the packet-by-packet operation of a network, the routing protocols running in a forwarding system's control plane are responsible for the moment-to-moment operation of a network: determining a network's topology and reacting to changes as links or systems fail, are removed from a network or are added to a network, and building and maintaining the contents of the forwarding databases to reflect a network's current topology and best forwarding choice for each destination.

The discussions that follow are arranged in an order that show the progression from the early days of the Internet, where simple and effective was good enough, to today's Internet where scale and stability are vital.

Routing Information Protocol (RIP)

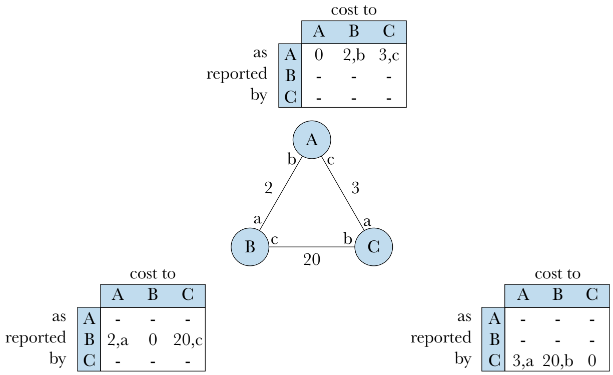
The routing information protocol is the original routing protocol for the Internet. RIP is known as a distance-vector routing protocol. What this means is that it attempts to minimize the number of hops when forwarding packets from a source endpoint to a destination endpoint. RIP supports a maximum of 15 hops. If a route to a destination is advertised as requiring 16 hops, it is interpreted as being infinite and, hence, unreachable via the path that is advertising that distance.

RIP operates in a distributed fashion, therefore there is no centralized intelligence or controller that coordinates the behavior of the protocol or the network. Each node in a network running RIP advertises the distance from it to all of the other nodes in the network. The distance is measured in units of hops. Each router along a path from the current node to the advertised destination node counts as a hop.

In virtually all networks of any significant scale and complexity, there are multiple possible paths from any node to any other node. It is the job of the distance-vector algorithm to find the shortest path from among all of the possible paths.

Distance-Vector Algorithm

The following series of figures (Figures 157–162) and their accompanying description walk through a simple example that illustrates the fundamentals of the distance-vector algorithm.



cost to

as reported by	A	B	C
A	0	2,b	3,c
B	-	-	-
C	-	-	-

cost to

as reported by	A	B	C
A	-	-	-
B	2,a	0	20,c
C	-	-	-

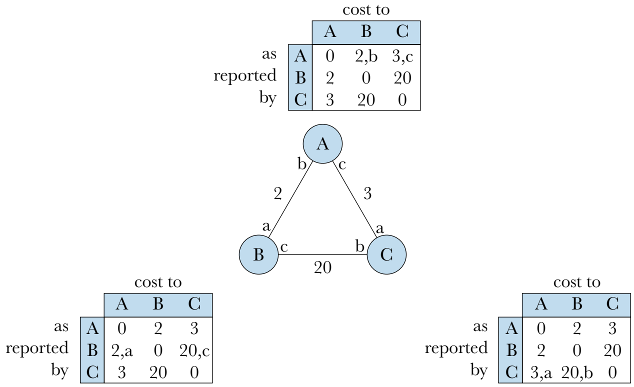
cost to

as reported by	A	B	C
A	-	-	-
B	-	-	-
C	3,a	20,b	0

Figure 157 Distance-Vector Algorithm Example, Step 1

In the first step of processing, each router determines the distance to each of its directly-accessible neighbor nodes. For example, router A's table shows that, according to its own determination, it has a distance of 2 to router B via interface b and 3 to router C via interface c. Of course, the cost to itself is zero. Similar determinations are made by routers B and C.

Once all of the routers have made these simple determinations, they share their distance-vector information with all of the routers that they're aware of. This is shown in Figure 158.



cost to

as reported by	A	B	C
A	0	2,b	3,c
B	2	0	20
C	3	20	0

cost to

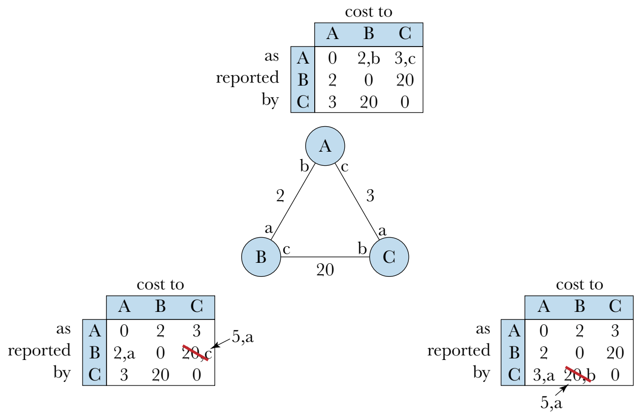
as reported by	A	B	C
A	0	2	3
B	2,a	0	20,c
C	3	20	0

cost to

as reported by	A	B	C
A	0	2	3
B	2	0	20
C	3,a	20,b	0

Figure 158 Distance-Vector Algorithm Example, Step 2

At this point, all of the routers have received distance vector information from all of the other routers. Each router now goes through and optimizes its distance computations. Router B, for example, initially assumes that its best route to C is via interface c at distance of 20. However, A is reporting that it is just 3 hops from C, and B knows that it can get to A in 2 hops. The total distance to router C via router A is 5, which is less than its current working value of 20. Hence, router B updates its forwarding database to favor interface a (with a distance of 5) when forwarding packets to C. The updates to the distance-vector tables are shown in Figure 159.



cost to

as reported by	A	B	C
A	0	2,b	3,c
B	2	0	20
C	3	20	0

cost to

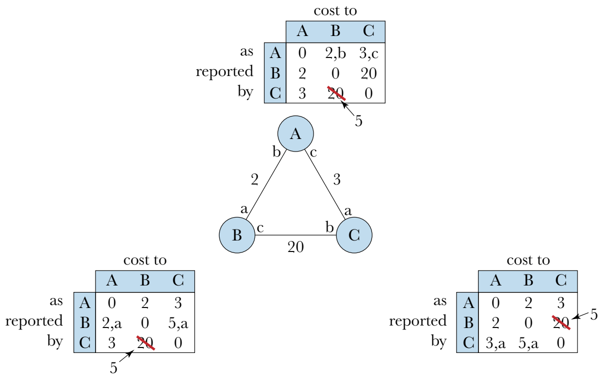
as reported by	A	B	C
A	0	2	3
B	2,a	0	~~20,c~~ 5,a
C	3	20	0

cost to

as reported by	A	B	C
A	0	2	3
B	2	0	20
C	3,a	~~20,b~~ 5,a	0

Figure 159 *Distance-Vector Algorithm Example, Step 3*

Notice that router A did not need to make any changes to its distance-vector table because it already had the shortest paths in its forwarding database. Finally, the newly computed cost values are distributed to all of the other nodes so that all of the routers have a consistent view of the network. This is shown in Figure 160.



cost to

as reported by	A	B	C
A	0	2,b	3,c
B	2	0	20
C	3	~~20~~ 5	0

cost to

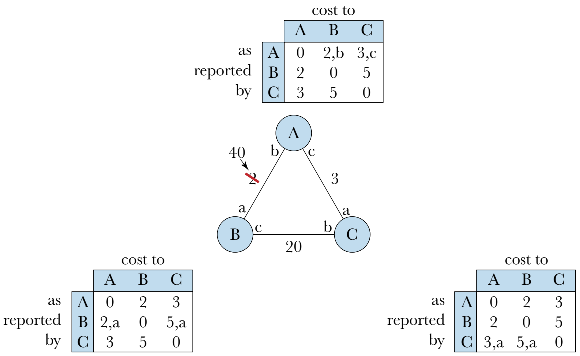
as reported by	A	B	C
A	0	2	3
B	2,a	0	5,a
C	3	~~20~~ 5	0

cost to

as reported by	A	B	C
A	0	2	3
B	2	0	~~20~~ 5
C	3,a	5,a	0

Figure 160 *Distance-Vector Algorithm Example, Step 4*

One of the weaknesses of the distance-vector algorithm upon which RIP is based is its slow convergence times when changes to the network topology occur. Let's consider what happens when the hop count from router A to router B increases from 2 to 40, as shown in Figure 161.



cost to

as reported by	A	B	C
A	0	2,b	3,c
B	2	0	5
C	3	5	0

cost to

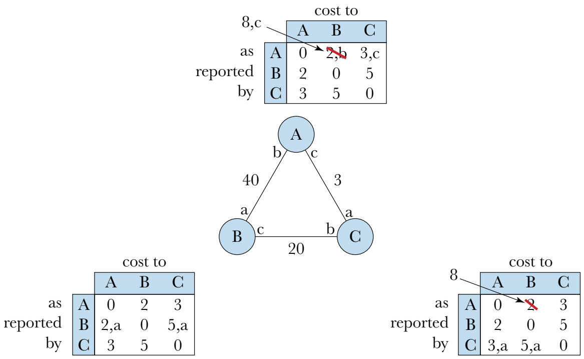
as reported by	A	B	C
A	0	2	3
B	2,a	0	5,a
C	3	5	0

cost to

as reported by	A	B	C
A	0	2	3
B	2	0	5
C	3,a	5,a	0

Figure 161 Distance-Vector Algorithm Example, Step 5

With the increase in the distance from A to B from 2 to 40, router A must update its cost table to reflect this change.



cost to

as reported by	A	B	C
A	0	~~2,b~~ 8,c	3,c
B	2	0	5
C	3	5	0

cost to

as reported by	A	B	C
A	0	2	3
B	2,a	0	5,a
C	3	5	0

cost to

as reported by	A	B	C
A	0	~~2~~ 8	3
B	2	0	5
C	3,a	5,a	0

Figure 162 Distance-Vector Algorithm Example, Step 6

(To keep the example simple, let's just concentrate on the interactions between routers A and C.)

In Figure 162, router A is shown updating its cost from itself to B from 2 to 8 instead of from 2 to 40. Why doesn't A adopt the new direct distance cost from A to B of 40? The reason for this is, according to the information in router A's table, A can get to C with a cost of 3 and that C can get to B with a cost of 5. Hence, the total cost from A to B is 8 via c. This value is also propagated to router C.

Router C reacts to this update by computing new lowest-cost paths. It determines that it can get to router A with a cost of 3 and that router A can get to B with a cost of 8. Therefore, router C's new cost for getting to B is 11. Of course, this new cost is propagated to router A, and the cycle continues until, eventually, the computed cost from C to B via A exceeds 20 and router C realizes that its direct connection to B is the shortest path and router A realizes that its shortest path to B is via C with a cost of 23.

This slow, incremental convergence is known as the "count to infinity" problem. The solution to the count to infinity problem is called "poison reverse." If a router—router A in our example—must advertise an indirect route to a destination (e.g., reaching directly-attached B via the route: C-to-B), then A advertises its distance to B as infinity, instead of 8. This causes C to immediately adopt the direct route to B with a cost of 20, and the count to infinity cycle is avoided.

Perhaps the most attractive aspect of RIP is its simplicity. It requires no configuration since all of the costs are simply the directly-measured hop counts from one router to the next. RIP will always come up with short, efficient routes between any accessible pair of routers.

Unfortunately, RIP can be very slow to converge and it doesn't scale well to large networks—it is limited to a maximum of 15 hops. Hence, as the Internet grew, replacements for RIP were sought and, now, RIP is largely obsolete. One of the first new routing protocols that was developed to replace RIP was the open shortest path first (OSPF) protocol.

Open Shortest Path First (OSPF)

In networking, one of the most effective and most commonly-used techniques for increasing the scale of a network is to introduce hierarchy. As the Internet grew, this is exactly what was done. This hierarchy was introduced in the form of Autonomous Systems (ASs). An autonomous system is defined by IETF RFC 1930 as "a connected group of one or more IP prefixes run by one or more network operators which has a single and clearly defined routing policy." An autonomous system may (and usually does) span multiple IP routers. A single IP router may support multiple autonomous systems. The important criterion is that the collection of IP prefixes are administered under a single routing policy; meaning that the routers within an autonomous system exchange reachability and routing information with one another in accordance with the relevant routing policy. Autonomous systems are identified by their autonomous system number (ASN): a globally unique number assigned by the Internet Assigned Numbers Authority (IANA).

The OSPF protocol operates exclusively within an autonomous system. OSPF is designated as an interior gateway protocol (IGP) owing to its domain residing entirely within an autonomous system. Exterior gateway protocols, by contrast, serve to establish routes and operate between autonomous systems.

OSPF is a link-state protocol as compared to the distance-vector behavior of RIP. In a nutshell, link-state routing relies on every node maintaining its own complete map of the network (i.e., the autonomous system) and independently determining the best paths to every other node in the network. The only routing information that is shared between routers pertains to reachability and costs to neighbor nodes. Reachability information (i.e., the identity of the node providing the information and a list of all of its directly attached nodes) is shared with all of those directly-attached nodes. Those nodes, in turn, forward the information to all of their directly-attached neighbors (except, of course, to the neighbor that supplied the information so as to avoid loops). Eventually, every node in the network has received neighbor connectivity information from all of the other nodes. When building its complete map of the network, a node only considers a link between two nodes to be valid if both ends of the link have reported each other to be neighbors. Once a node has assembled its network map from all of the individual reports, it's then just a matter of determining the best paths through the network so that when any node needs to forward a packet to any particular other node, it can choose the packet's best next hop while being confident that the chosen next hop router will, in turn, choose its best next hop.

Each node operates independently in using its network map information to determine the best paths through the network. Each must compute what is known as a shortest-path tree. Meaning, a node doesn't need to compute the best paths from every node to every other node, it must only compute the paths from itself to every other node; a tree, in other words. This independence of action also means that each node may use a different algorithm to find the shortest paths without compromising interoperability. Generally, some variant of Dijkstra's algorithm is used.

Dijkstra's Algorithm

Figure 163 shows a simplified map of the freeways of the San Francisco Bay Area. This map will serve as a useful example for how Dijkstra's algorithm quickly finds the best route from a starting point to an ending point.

The lettered black dots represent freeway interchanges and are analogous to network routers. The gray lines between the black dots are, of course the freeways, and the numbers represent the travel time from interchange to interchange. The travel times are a function of distance and speed. Some of the paths that appear physically shorter than others may have longer travel times due to congestion. For this example, we'll summarize travel distances and times simply as costs.

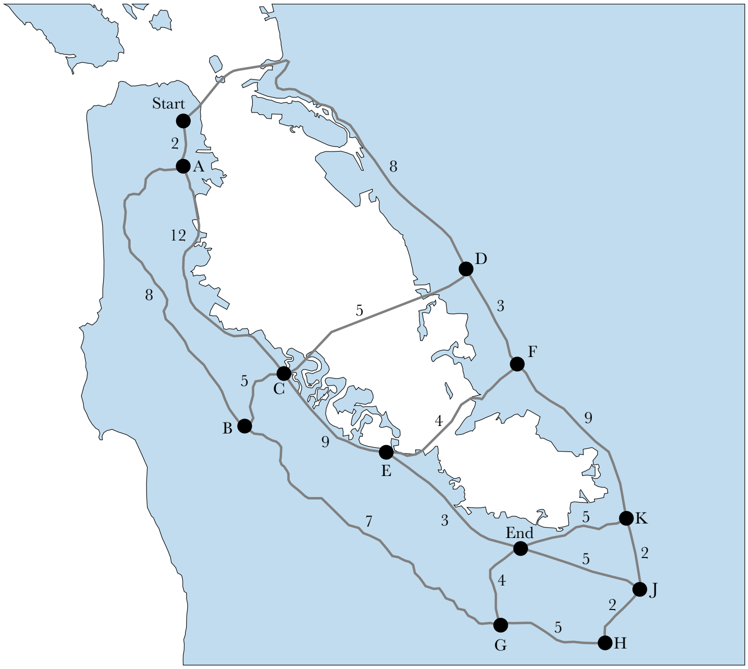
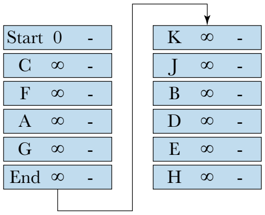


Figure 163 *Dijkstra's Algorithm Example Map*

The first thing that is done according to Dijkstra's algorithm is to build a priority list of all of the nodes, sorted by the known total cost of the best path to each node. The Start node has a map of all of the nodes and their node-to-node costs, but it hasn't yet determined the costs of the best paths. Hence, those costs are initially set to infinity; except the cost to Start, which is known in advance to be zero. Each entry in the list of nodes holds the name of the node, its best-path cost, and the immediate upstream node in the best path to the current node. The initial list, or first step, is shown in Figure 164.



Start	0	-
C	∞	-
F	∞	-
A	∞	-
G	∞	-
End	∞	-
K	∞	-
J	∞	-
B	∞	-
D	∞	-
E	∞	-
H	∞	-

Figure 164 *Dijkstra's Algorithm Example, Step 1*

In the initial state, all of the costs are set to infinity and there are no route records. The Start node is at the top of the priority queue because it has the lowest cost: zero. The remaining nodes are sorted arbitrarily in the queue.

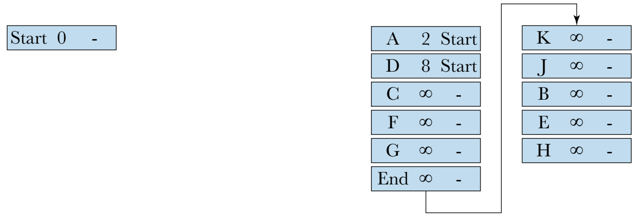


Figure 165 Dijkstra’s Algorithm Example, Step 2

In Step 2, the costs to the nodes that are directly attached to Start are checked. They are A and D and they have costs of 2 and 8, respectively. This determination accomplishes two things. First, it completes our work with Start, so it is moved to a “completed” list. Second, now that we have costs to A and D, we can re-sort the priority queue. Notice also that A and D are marked with Start as the node that immediately preceded them. This is their route record. Since A is now at the top of the queue, we look at the nodes to which it is directly attached (excepting Start, because it is in the completed list).

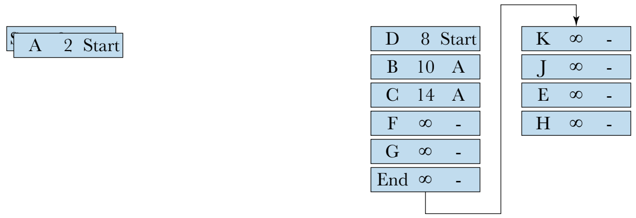


Figure 166 Dijkstra’s Algorithm Example, Step 3

In Step 3, we examine the exit routes from A and determine the total costs to nodes B and C—A’s directly attached nodes. Since we’re looking for the total cost, the costs to B and C through A must include the cost from Start to A. These new costs allow us to re-sort the queue and to move A to the completed list since we’ve examined all of the paths that lead out of A. Notice that D is now at the top of the queue. Thus, that is the next node to be examined. Remember, we want to open the shortest (i.e., lowest cost) paths first.

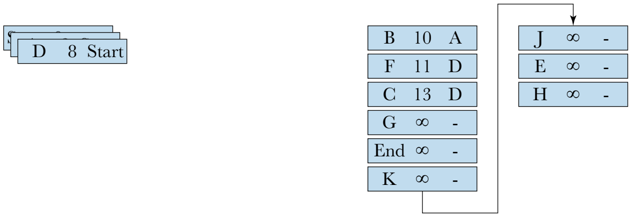


Figure 167 *Dijkstra's Algorithm Example, Step 4*

An interesting thing happened in Step 4. A lower-cost route to C was found through D (total cost 13) compared to the original measurement of 14 through A, hence its backward-facing pointer is changed from A to D. The entries for F and C are updated accordingly and the queue is re-sorted. Also, D is moved to the completed list. Next, we examine the routes leaving B, the node at the top of the priority queue.

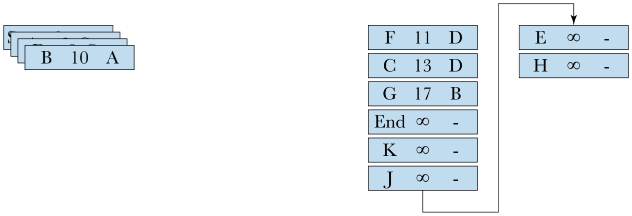


Figure 168 *Dijkstra's Algorithm Example, Step 5*

In Step 5, we've completed work on B and determined the total cost to G. Though C is also directly connected to B, its entry wasn't updated because its existing total cost is lower than the total cost through B.

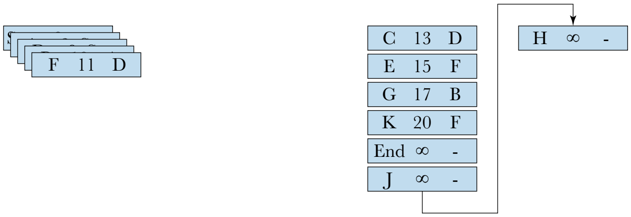


Figure 169 *Dijkstra's Algorithm Example, Step 6*

In processing F in Step 6, we determine the total costs to E and K. C has the lowest total cost, so that's up next.

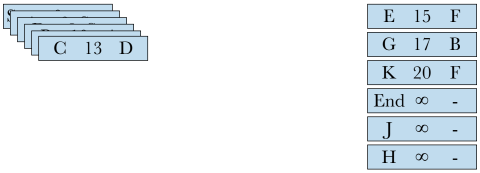


Figure 170 Dijkstra's Algorithm Example, Step 7

C is directly connect to A, B, D and E. A, B and D are on the completed list, so only E is of interest. However, the total cost to E through C is higher than the E's current cost measurement, so no changes are made except that C is moved to the completed list. Next up: E.

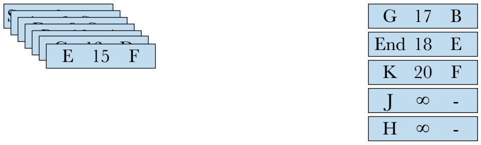


Figure 171 Dijkstra's Algorithm Example, Step 8

In Step 8, we've measured the cost to the End node for the first time since E is connected to End. However, its total cost doesn't bring End to the top of the priority queue, so we're not quite done since it is possible that a lower-cost route still exists.

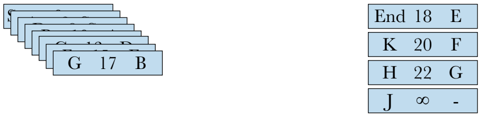


Figure 172 Dijkstra's Algorithm Example, Step 9

In Step 9, the End node has percolated to the top of the priority queue. Once that happens, there is no point in examining any of the other nodes or routes since they are all guaranteed to have a higher cost. The final step is to simply work backwards from the End node to determine the path of the shortest route. That is illustrated in the final step, Figure 173. Notice how End points to E, E points to F, and so on.

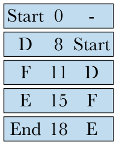


Figure 173 Dijkstra's Algorithm Example, Step 10

The path list shown in Figure 173 is the route shown in the map in Figure 174.

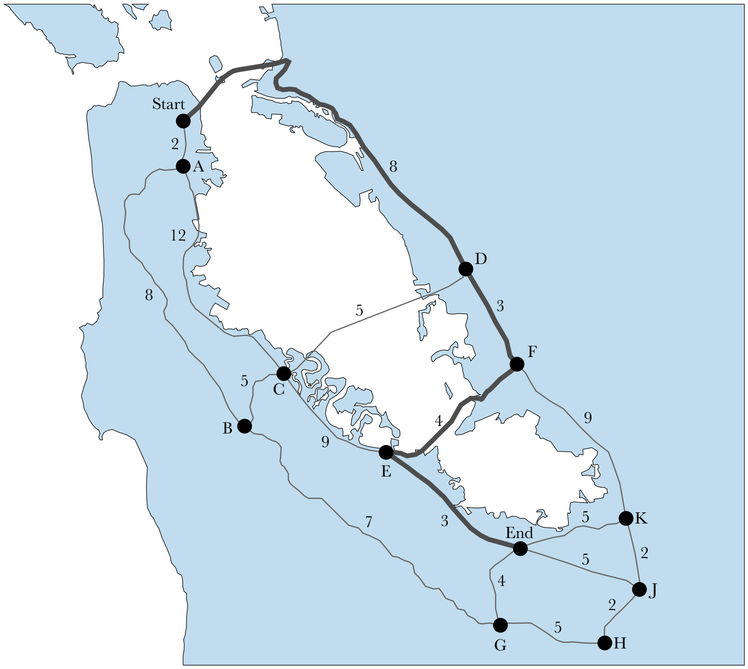


Figure 174 *Dijkstra's Algorithm Example Map Best Route*

Now, even though the Start node knows the entire path for the shortest route to the End node, it only uses the path to D (i.e., the next hop) in its forwarding database instead of storing the entire list of nodes. It is possible to forward the packet only by its immediate next hop because all of the other nodes (i.e., routers) in the network have come to the same answers and have set up their forwarding database to forward to what is, from their perspective, the next hop on the best path to the End node.

OSPF's Forwarding Protocol Dependency

OSPF was invented before the advent of IPv6 and was, therefore, intimately tied to IPv4. The OSPF messages are carried directly on top of IPv4 headers without using an intermediate layer such as UDP. Also, the OSPF messages are IPv4-specific (e.g., 32-bit addressing, etc.). Thus, when IPv6 was developed, a new version of OSPF had to also be developed. As a consequence, there are two separate IETF RFCs for OSPF: RFC 2328 for IPv4 and RFC 5340 for IPv6.

Given the close ties between OSPF and the version of IP upon which it runs, updating the protocol to support a new forwarding protocol was a major undertaking. IS-IS, on the other hand is a Layer 2 protocol and is agnostic of the kind of forwarding protocol to which it is applied.

Intermediate System to Intermediate System (IS-IS)

At first glance, IS-IS (pronounced “eye ess, eye ess”) is largely the same as OSPF. Both are link-state protocols that use Dijkstra’s algorithm (or a variant thereof). Both operate within Autonomous Systems. And both are classified as interior gateway protocols. However, the differences are certainly noteworthy.

IS-IS was standardized by ISO/IEC (as 10589:2002) as opposed to being standardized by the IETF as most Internet-related protocols are. Because it was standardized outside of the primary Internet standards organization, it takes great pains to be protocol agnostic. Through type-length-value (TLV) structures it can perform routing for virtually any routable forwarding protocol. It can also convey its messages by virtually any Layer 2, connectionless forwarding protocol.

IS-IS operates at Layer 2; meaning, the IS-IS messages are carried directly on top of an Ethernet packet. These are the old-school 802.3 Ethernet packets rather than the more common Ethernet v2 packets. IS-IS is identified as an Ethernet packet’s payload by the LSAP value `0xfefe`. Because IS-IS operates over Ethernet (or any other Layer 2 protocol such as ATM), it cannot be routed and an IP router will not forward IS-IS messages. This helps to enforce the hop-by-hop nature of its behavior and protects it from certain kinds of route spoofing.

IS-IS introduces some hierarchy within an Autonomous System through the definition of “areas.” An IS-IS router can be designated to operate at one of three levels:

- Level 1: intra-area;
- Level 2: inter-area;
- Level 1-2: both intra-area and inter-area.

Level 1 routers can exchange information with other Level 1 routers that operate within the same area. Level 2 routers can exchange information with all Level 2 routers within the same Autonomous System. The Level 1-2 routers act as gateways between the various areas (i.e., Level 1 domains) and Level 2. Hence, level 1-2 routers operate at both Level 1 and Level 2. This strict delineation of relationships between routers makes IS-IS simpler in comparison with OSPF.

IS-IS is “less chatty” than OSPF. Meaning, it requires fewer messages be exchanged between routers. This makes it more suitable for particularly large networks.

And a final little tidbit: IS-IS is also used for IEEE 802.1aq Shortest Path Bridging (SPB).

Border Gateway Protocol (BGP)

BGP is what's known as an exterior gateway protocol. What that means is that it is intended to perform the routing between Autonomous Systems. In other words, it is the protocol that glues together the worldwide Internet. The current version of BGP (version 4) is specified by IETF RFC 4271 and has been in use on the Internet since 1994. While BGP is strongly associated with routing between autonomous systems, it is not limited to that role; BGP can also be used within an autonomous system as an interior gateway protocol. In this role, it is often referred to as iBGP. To avoid confusion, BGP operating as an exterior gateway protocol is referred to as eBGP.

BGP is a path-vector protocol as opposed to a link-state protocol. A path-vector protocol reports the complete path taken from an advertising BGP router to the current BGP router. How this is done is quite simple. When a BGP router advertises the IP address prefixes (i.e., subnets) that are reachable within it, it sends that advertisement message to all of the other BGP routers to which it is directly connected. These directly-connected BGP routers append their Autonomous System Numbers (ASNs) to the message and then forward the advertisement on to their directly attached BGP routers. Eventually, the current router—the one whose perspective of the network that we are considering—receives a copy of the advertisement from that original BGP router that had something to report from all possible paths through the network. Hence, the current router has information on all possible paths from itself to a particular IP subnet. The best path is the one with the fewest hops.

A distance-vector protocol (as used by RIP) bears some similarity to a path-vector protocol in that the number of hops is used to rank all of the possible paths. The key difference is that a distance-vector protocol reports each node's neighbors and the cost to reach them whereas a path-vector protocol reports all of the paths to each possible BGP router.

Another unique characteristic of BGP is that TCP connections (port `179`) are used to exchange information between BGP routers. Since each TCP connection is a point-to-point connection, a full mesh of TCP connections is required. In other words, a network of 500 BGP routers requires each router to maintain 499 TCP connections. These connections, plus the very large and detailed route database means that BGP routers must have significant compute and storage capacities.

BGP supports dual-homed attachments. If a customer's BGP edge router is attached to two service providers' networks, it will not advertise routes that connect those two service providers networks together. Therefore, both service providers can communicate with that customer's network, but the service providers cannot communicate with one another via that customer; they must find a different route to do so.

One of the side effects of the large route databases and manually configured TCP connections between BGP routers is that there are lot of opportunities for fine tuning the behavior of the network through administratively-assigned policies. These policies may be related to peering agreements between service providers, to the reliability and availability along certain paths, or whatever strikes the fancy and captures the imagination of the network administrators.

20 Forwarding System Architecture

Modern forwarding system architectures generally follow a common pattern where the system is neatly divided into three mostly-independent subsystems that are often referred to as planes: the control plane, the data plane, and the forwarding plane. The control plane is generally implemented entirely in software and is responsible for running the routing protocols and managing the contents of the forwarding databases and other system processes. The data plane is responsible for communicating with the Layer 2 media access controllers (Ethernet, usually), buffering and queuing packets, and delivering packets from device to device in multi-device systems. Forwarding planes come in many forms. A purely software-based forwarding system may implement both the data plane and forwarding plane in software or it may have specialized hardware for managing buffering and queuing while relying on software for algorithm-intensive packet processing. The next step up in performance and efficiency is a sea of specialized micro-coded engines. These engines have instruction sets that are optimized for the kinds of bit vector extractions and manipulations that are typical in packet processing applications. For the ultimate in packet processing performance and efficiency, a pipeline is typically used. However, before delving into the details of these various options, it is helpful to take a look at how a forwarding system can be broken down into its most essential elements. Figure 175 shows these elements and illustrates their relationships.

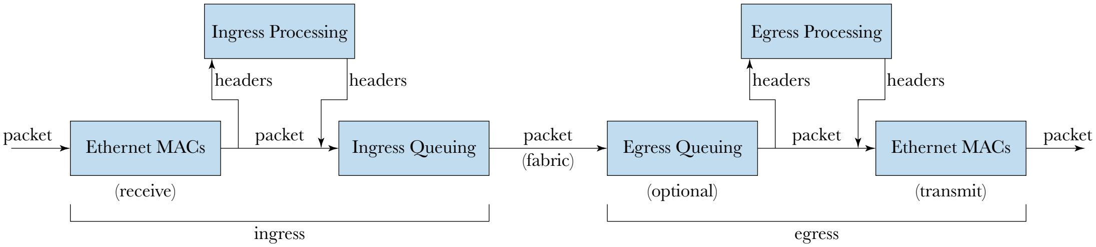


Figure 175 *Forwarding System Architecture Building Blocks*

From left to right in Figure 175, a packet is received from an Ethernet MAC and is split to follow two distinct paths. The packet's headers are delivered to ingress packet processing while the remainder of the packet (the payload bytes) are delivered to the ingress queuing system. Once ingress packet processing is done with the headers, those headers (which may be modified) along with some pertinent metadata are delivered to the ingress queuing system where they rejoin the packet's payload bytes and are enqueued to await an opportunity to be transferred to egress processing.

The fabric connection between the ingress and egress can either be an actual chip-to-chip packet- or cell-based fabric system, or it may simply be a convenient way to delineate the ingress and egress realms of a single-chip forwarding system. In a single-chip system, ingress and egress queuing may be combined.

Once across the fabric, the packet enters the egress queuing system. This system is optional for certain data path architectures. As in the ingress side of things, the packet's header and payload take separate paths to the Ethernet MACs, with the headers making a detour through egress packet processing.

There are, of course many variations of the theme described above, but Figure 175 serves as a useful template and as a basis for discussion. The remainder of this chapter is broken down into two main sections: Forwarding Plane and Data Plane.

Forwarding Plane

Regardless of the protocols involved or the details of the architecture of a forwarding system, the operations of the forwarding plane (i.e., packet processing) can be broken down into three main phases:

- Tunnel decapsulation
- Forwarding
- Tunnel encapsulation

Most packet forwarding scenarios involve the termination of tunnels. The tunnels that must be terminated are addressed to the current forwarding system by the header that corresponds to the to-be-terminated tunnel. The information in each terminated tunnel is used to update a packet's forwarding domain and receive interface metadata: updates that can influence the further work that must be done. All of the encapsulating headers that precede the packet's forwarding header must be processed before the forwarding header may be processed.

The forwarding phase examines the first header (from the start of the packet) that does not correspond to a terminating tunnel (i.e., it isn't addressed to the current forwarding system). It is the forwarding phase that determines how, and to where, a packet must be forwarded. In a multi-chip forwarding system, that determination also yields the identity of the chip that is associated with the packet's intended transmit port. Thus, the packet must be addressed to that chip when it crosses the chip-to-chip fabric.

Since tunnel decapsulation must be completed prior to forwarding, and forwarding must be completed prior to reaching the fabric, it makes sense to group tunnel decapsulation and forwarding together on the ingress side of the forwarding system. This leaves tunnel encapsulation to reside in egress processing.

Placing tunnel encapsulation in the egress side of a forwarding system isn't just a default or arbitrary choice. It is actually quite beneficial to perform egress processing close to the transmit interfaces. Consider ECMP (equal-cost, multi-path). With ECMP, the same destination is reachable via multiple paths. This means that a series of packets addressed to, say, the same IPv6 destination address may be transmitted by a number of different physical ports residing on multiple chips in a multi-chips system. Each path and transmit port may have its own particular tunnel encapsulation requirements. In other words, a packet being transmitted by one port may just require a simple Ethernet + VLAN encapsulation whereas another may required Ethernet + VLAN + MPLS. By placing the tunnel encapsulation processing and its associated databases close to the transmit ports, those databases may be optimized to only hold entries that pertain to the transmit interfaces associated with the egress chip. This, of course, requires that tunnel encapsulation operations be located in the egress side of a forwarding system.

With that brief introduction out of the way, let's dive into some of the details.

Code vs. Pipelines

The trade-off between code (i.e., microprocessors or micro-coded engines) and pipelines is that of flexibility vs. efficiency. With sufficient code storage resources, a code-based packet processing function is infinitely flexible. Pretty much any packet handling algorithm that can be imagined can be implemented. Unfortunately, that flexibility comes at a price.

The efficiency of a piece of silicon is a function of the number of data movements that are required to accomplish a net amount of work. With a code-based packet processing implementation, every operation involves the movement of instructions from some kind of instruction storage to the execution unit, and most instructions require the movement of data to or from storage, or to or from (or to *and* from) some kind of arithmetic logic unit. All of this data movement requires a lot of active logic gates and wires, which increases the area and power penalties relative to a pipeline architecture.

A pipeline, on the other hand, does not need to fetch instructions since the expected behavior is hardwired (in the most extreme cases) into each stage of the pipeline itself. A pipeline can be thought of as being like an assembly line. An assembly line has a number of stations along some form of conveyor. The item being worked on (i.e., a packet's set of headers) is moved from station to station, and each station performs work that is specific to that station; each has easy and ready access to whatever tools or resources that are required. A code-based implementation is more akin to a solitary craftsman toiling away and performing every operation required to get the job done. Performance can be scaled by having a lot of craftsmen working in parallel on separate instances of the job at hand, but that model can never achieve the efficiency of an assembly line.

Of course, an assembly line that is designed to build cars may not be well suited to building, say, mobile phones. Therein lies the principle shortcoming of pipelines: inflexibility. However, pipelines can be designed with a high degree of configurability (within certain limits).

Another interesting characteristic of the relative efficiencies of pipelines versus processors is how their performance varies with the amount of work that must be performed per packet. This is illustrated in Figure 176.

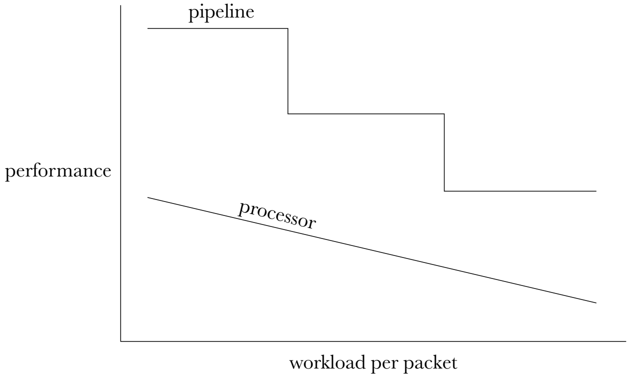


Figure 176 Incremental Performance of Pipelines vs. Processors

What's happening in Figure 176 is that the processor's performance is degrading continuously as more and more features and options are enabled on a per-packet basis. This should be intuitive since a processor must execute more instructions to get more work done, and there's some unit of time consumed for each of those instructions. A pipeline, on the other hand, can tend to have a certain time to completion for the simplest possible packet forwarding scenario that can be maintained as more and more features and options are enabled. However, at some point an additional feature is going to require that a packet perform two operations for which the pipeline is designed to handle just one. This will require some kind of looping or recirculation (i.e., sending the packet through some or all of the pipeline a second time). Hence the step function of decreasing performance as more and more features and options are enabled. Despite the quantum changes in pipeline behavior, it achieves higher performance and is more efficient than a code-based implementation for any particular workload.

Tunnel Decapsulation

Packets are best processed layer by layer. Some shortcuts may be possible here and there, but a methodical approach is at least a good conceptual starting point. Generally, each header—as encountered in sequence from the start of the packet—identifies the subsequent header. Thus, if the outermost header type is known, that header can identify the next header and so on. This is the basis of header parsing. Once a header's type is known, its contents (i.e., its fields) may be extracted and examined.

Headers fall into two broad categories: forwarding and non-forwarding. Examples of forwarding headers include Ethernet, IPv6, and MPLS. Examples of non-forwarding headers include VXLAN and GRE. Tunnels must be associated with a forwarding header and may be supplemented by a non-forwarding header. For a forwarding system to terminate a tunnel, that tunnel's header must be addressed to the forwarding system. In other words, the header's destination addressing information must match the address of one of the forwarding entity instances of the corresponding type. For example, to terminate an Ethernet tunnel, an Ethernet header's `destinationAddress` value must match the MAC address of one of the instances of an Ethernet forwarding entity.

When a successful address match is made, information from the tunnel's header is used to update certain metadata attributes of the packet. For example, the header's destination address value in combination with the packet's current forwarding domain serves to identify a specific forwarding entity. In concrete terms, what that means is that the tunnel header's destination address and forwarding domain serve to define the packet's new forwarding domain for use with the next header in sequence. Similarly, the tunnel header's source address information serves to update the packet's receive interface identity. This is particularly useful when terminating a tunnel that exposes an Ethernet header as the packet's forwarding header since Ethernet bridging demands that a receive interface be associated with each learned MAC `sourceAddress` value.

Once a tunnel's header has been terminated, it may be safely removed from the packet. This process is known as decapsulation. One important caveat here, though: the tunnel headers may provide useful search argument material for access control list (ACL) processing.

Forwarding

At some point in the header processing sequence, a header will have a destination address value that does not correspond with any addresses configured for any of a forwarding system's forwarding entities. In other words, the destination address does not indicate that a tunnel must be terminated within the current forwarding system. This header is the packet's forwarding header and it informs the current forwarding system of the packet's destination. The destination encoded in the forwarding header may not be the packet's ultimate destination; it may just be the remote exit point of some tunnel. But, as far as the current forwarding system is concerned, the forwarding header is to be used as the basis for making a forwarding decision.

A forwarding decision usually entails two distinct operations. The first is to resolve the addressing information to a destination identifier that is meaningful within the context of the forwarding entity that is forwarding the packet. The second operation is to use that destination identity to determine what the packet's next hop is and how to get there.

Destination Resolution

A packet's forwarding destination is typically resolved by combining the forwarding header's destination address value with the packet's current forwarding domain to form a contextually-unique search argument. This search argument is submitted for a search into the forwarding system's forwarding database. A search that is successful (i.e., a matching key is found) returns a mix of destination-specific information and, optionally, next-hop-specific information.

Destination-specific information may include any number of instructions or attributes associated with the matching forwarding database key that can be used to update aspects of the packet. These aspects may include quality-of-service values, time-to-live handling and more. At a minimum, a successful search simply returns an index value that corresponds to the matching key.

Failed searches (i.e., no matching key found) are also interesting. Different forwarding protocols have different expectations for how to handle a failed destination resolution search. Ethernet bridging, for example, stipulates that the packet must be flooded to all available transmit interfaces within the current forwarding domain, while IP generally allows for a default route. A failed search may also return next-hop-specific information.

Next Hop Determination

Any forwarding entity that is forwarding a packet that is a payload of some kind of tunnel-encapsulation packet isn't necessarily forwarding the packet to its ultimate destination (i.e., the endpoint or tunnel exit identified by the forwarding header's destination address value). Instead, that forwarding system is simply striving to get the packet to the next hop on the packet's path to its ultimate destination.

The number of possible ultimate destinations may number in the billions. Meanwhile, the number of next hops to which a packet may be forwarded may number in the dozens or hundreds. Thus, there is a lot of sharing going on; a lot of separate destinations may share a common next hop.

The next-hop-specific information returned from a successful search operation may specify such things as the kinds of tunnel encapsulations that are required and, at a minimum, the transmit interface to use to transmit the packet toward its next hop.

Next hop determination must account for equal-cost, multi-path (ECMP) requirements. ECMP inverts the observation that a lot of destinations may share a common next hop. ECMP implies that a single destination may have multiple, equally-suitable next hops, only one of which is chosen for each packet.

Link aggregate group (LAG) behavior must also be considered during next hop computations. Since LAGs operate at the link layer and it is expected that all of the links in a LAG terminate at the same next hop, the actual next-hop determination is the same regardless of the link choice made within the LAG. However, since a LAG's links may be spread across multiple chips in a large forwarding system, it is incumbent upon ingress processing to resolve the LAG selection before the packet is forwarded to the egress chip. Thus, ECMP and LAG path selections are, ideally, both resolved by actions within the same general processing function.

Once a packet's next hop has been identified and the packet's transmit interface has been chosen, tunnel encapsulation processing must follow a series of inevitable steps to prepare the packet for transmission.

Tunnel Encapsulation

Tunnel encapsulation is, essentially, the reciprocal of tunnel decapsulation. Specifically, new headers are added to the packet for each tunnel that the packet must enter at the current forwarding system. Unlike tunnel decapsulation, tunnel encapsulation does not depend upon querying large databases using the addressing information contained within headers. Instead, the next-hop determination made according the description above provides the encapsulation process with the starting point that tunnel encapsulation requires.

In the egress part of the packet processing process, what is encountered is a series of converging tree structures. Meaning, while there is a sequencing that is required to lead from one encapsulating header to the next, the sequencing is generally unconditional. A number of initial starting points may share a common next node in the tree. Several sibling nodes at the same level of hierarchy may, in turn, share a common next node. And so on. For the tunnel encapsulation process, it is a simple matter of following this converging path through a tree structure.

Each new encapsulating header must be populated with addressing and other information. The destination address information is directly related to the identity of the next hop determined during ingress processing. So, as the converging tree structure is followed, each node's identity is related in a many-to-one fashion (i.e., many nodes sharing a common address value) with a destination address value that is appropriate for the header type and its place in the sequence of headers.

Source address values for each header must generally point to the current forwarding entity. Again, the current node in the tree structure is related in a many-to-one fashion with a source address value.

Of course, other aspects of each new encapsulating header must be managed correctly. Quality of service markings and time-to-live values must be set according to rules that are appropriate for each tunnel being entered. Each packet has a traffic class value as part of its metadata that indicates the priority level of the packet. However, each tunnel instance may have its own encoding requirement for mapping an abstract traffic class value to a concrete priority code point in a header. Similarly, tunnels may manage their time-to-live values in either a transparent or opaque manner. This means that the identity of each tunnel (as inferred from the converging tree node that corresponds with the tunnel) is associated with the rules for time-to-live value management. Generally, for opaque tunnels, the time-to-live value of the payload is ignored and the new tunnel header's time-to-live value is set to its maximum allowed value, and, for transparent tunnels, the time-to-live value of the new tunnel header is set equal to the time-to-live value of the tunnel's payload header. (It is presumed that the forwarding header's time-to-live value is decremented prior to the start of the tunnel encapsulation process.)

Data Plane

The data plane of a forwarding system connects the outside world (i.e., the system's network interfaces) to the forwarding and control planes, and it interconnects the various separate chips that are used to build a forwarding system.

Packets vs. Cells

The packet forwarding chips (often referred to as PFEs, for packet forwarding engines) typically fall into one of two broad data plane categories. These categories describe the means by which PFEs within a single multi-chip forwarding system communicate with one another across an interconnection fabric. There are packet-based PFEs and cell-based PFEs.

With a packet-based PFE, all of its external interfaces are packet interfaces such as Ethernet. Thus, when building a multi-chip system, these packet-based interfaces are used to interconnect the chips. The benefit of this approach is that a standards-based interface is used, so there's one less thing to invent and make work. Packet-based fabrics also have generally good latency characteristics because the transfer of a packet from one chip to another across the fabric is not negotiated in advance; the packet is simply transmitted from an ingress PFE to a egress PFE in a best-effort manner.

Perhaps the single greatest benefit of the all-packet chips is that they are perfectly suited to building forwarding systems whose total bandwidth and port capacity exactly matches that of the chip being used: the so-called single-chip system. If, for example, you have a 3.2 Tbps chip, it is a simple matter to build a 3.2 Tbps system. Just take that chip and wrap some sheet metal around it and, give or take a little bit, you're done.

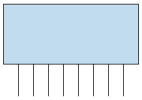


Figure 177 *Single-Chip System*

However, 3.2 Tbps is not a terribly ambitious goal. What if you want to make a 6.4 Tbps system? Well you can't simply take two 3.2 Tbps devices and place them in an enclosure; you'd end up with two sets of ports that can't exchange packets with one another.

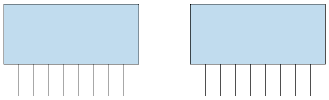


Figure 178 *System With Two Isolated Chips*

If you were to naïvely use half of each chip's ports to provide a non-blocking interconnect between the two devices, you'd end up exactly where you started: with a total of 3.2 Tbps of bandwidth on the front panel of your system.

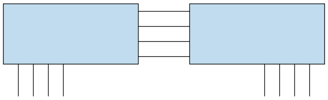


Figure 179 *System With Two Naïvely-Interconnected Chips*

Indeed, what's required is to use six 3.2 Tbps chips to achieve 6.4 Tbps of fully connected, non-blocking interconnection.

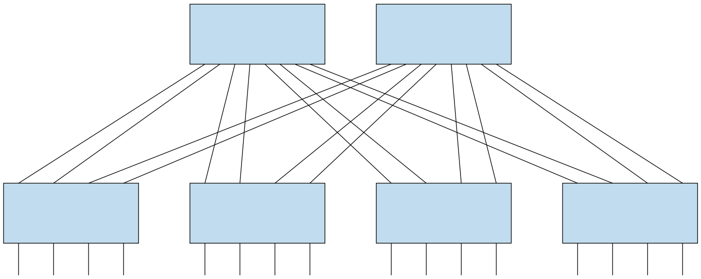


Figure 180 *Proper Multi-Chip System*

The style of interconnecting the devices shown in Figure 179 is known as a Clos fabric. Clos fabrics are extremely common and are used in a wide variety of applications. Clos fabrics have the remarkable characteristic that they provide completely non-blocking, any-to-any interconnect with the fewest number of wires. However, for a Clos fabric to fully deliver this behavior, all of the available interconnection paths must be fully and equally utilized. And this is where systems built with packet-based chips begin to suffer.

The reason packet-based chips are less than ideal when building multi-chip systems is that packets come in a huge range of sizes: from 64 bytes to over 9,000 bytes. When a packet-based ingress PFE attempts to make full use of its Clos fabric by transmitting packets onto whichever interface is the next-available interface (to ensure that each interface was always busy), packets will arrive at the egress PFE wildly out of order. This mis-ordering of the packets is caused by the skew between the packets due to their following various paths across the fabric. That skew is the byproduct of the high dynamic range of packet lengths.

According to the Forwarding Entity Axiom (see page 12), packets within a flow must be delivered in order. Hence, if the packets arrive at an egress PFE out of order, they must be placed back in order by the egress PFE. The greater the skew between the shortest forwarding time and the longest dictates the scale of the re-ordering function. Greater scale implies more storage resources for packet data and more state storage for the partially re-ordered flows. Because of this scale problem, packet re-ordering is generally not done. Instead, ingress PFEs strive to ensure that packets that belong to a common flow all follow the same path across the Clos fabric, ensuring that per-flow packet order is maintained.

ECMP-like behavior is used by packet-based ingress PFEs to distribute packets across the available fabric paths in a flow-aware manner. There are, of course, a couple of shortcomings with this approach. First, hash functions are not perfect. They cannot guarantee even distribution of outputs for any particular set of inputs. A histogram of hash function outputs is never a straight line. These hash collisions (i.e., multiple inputs yielding the same output) cause multiple flows to map to the same path. And, because this is done unevenly, some paths are assigned more flows than others. Second, not all flows are created equal; some flows are busy while others are quiet. These two shortcomings combine to yield fairly uneven bandwidth distribution across the available paths. And, once a single path is oversubscribed, congestion can build up that affects the entire network. So, it is not uncommon to design packet-based Clos interconnect fabric networks such that they are overbuilt to avoid congestion or to presume that, overall, the system's bandwidth demand will never be that high.

Breaking packets up into cells solves the problems just described. Because the cells are generally small and of fixed length, the dynamic range of forwarding latencies between PFEs is also small. This makes it practical to "spray" the cells across all available paths through the fabric and then re-order and reassemble the cells into packets in the egress PFE. With a cell-based Clos fabric, all of the paths are always equally utilized (or nearly so). This means that a cell-based Clos fabric operates at very high efficiency, regardless of the characteristics of the flows being forwarded. This, in turn, yields deterministic performance.

Output Queues

In an idealized model of a multi-chip forwarding system, all of the ingress PFEs could forward the entirety of their received bandwidth to a single egress PFE where it is safely buffered and queued to await transmission without experiencing any congestion due to oversubscription of the fabric. In other words, the fabric is scaled to handle b(n - 1) in total bandwidth where b is the bandwidth per PFE and n is the number of PFEs (it's n - 1 because it is presumed that packets addressed to a port on their own receive PFE do not need to cross the fabric).

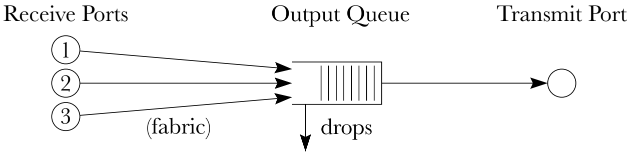


Figure 181 *Ideal Output Queue Model*

In this model, packets are shuttled across the fabric as quickly as they are received and are then buffered and queued for each transmit port (ignoring, here, multiple priorities per port) in the order in which they were received. This provides completely fair treatment of all of the received packets regardless of which port or PFE received the packets.

Such a system is, of course, wildly impractical. Instead, systems are built using either combined input-output queues or virtual output queues.

Combined Input-Output Queues

In a combined input-output queue (CIOQ) system, both the ingress PFE and egress PFE have substantial buffering capacity for packets while they await transmission onto a network by the egress PFE. This means that the packet is stored and retrieved twice: once in the ingress PFE and once in the egress PFE. The ingress PFE may have, say, one fabric-facing queue for each egress PFE. Thus, all of the packets that must be transmitted by the several ports of a particular egress PFE are enqueued into the same ingress queue, even though they may, ultimately, be transmitted by any of the egress PFE's physical transmit ports, each of which may be operating at its own bit rate.

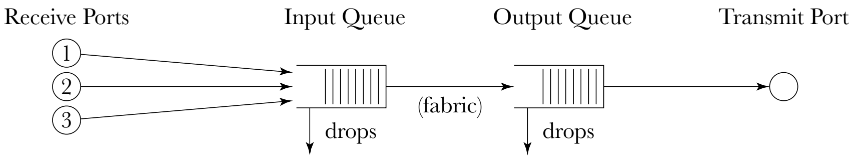


Figure 182 *Combined Input-Output Queue Model*

Though the egress PFE's packet buffering resources may be substantial, they are not infinite. It is possible for an egress PFE's packet buffering resources to become congested when the number of packets arriving from a number of ingress PFEs exceeds the rate at which those packets can be transmitted by some of the egress PFE's slower transmit ports. When congestion occurs in the egress PFE, it signals the congestion to the various ingress PFEs to prevent them from overrunning the egress PFE's buffer space and, consequently, suffering packet loss. Unfortunately, just a few slow transmit ports (or even one) can cause enough congestion in an egress PFE to prevent all new packets from arriving at that PFE, even those packets that are not addressed to the slow, congestion-causing ports. In other words, congestion on one path leads to the interference of another path. This is known as head of line blocking. For a discussion of head of line blocking, see *"Head of Line Blocking" on page 201*.

Generally, head of line blocking is mitigated (but not entirely avoided) in CIOQ systems by building a certain amount of speed-up in the fabric that interconnects the PFEs. With speed-up available, it is possible to move the packets from multiple ingress PFEs to a single egress PFE simultaneously without having to slow down those ingress PFEs. Of course, if this scaling up of the fabric is taken to its limit in order to achieve ideal behavior, then you've arrived at the output queuing model described in the previous section. Which, as has been stated, is wildly impractical.

One of the tremendous benefits of the CIOQ architecture is that the egress PFEs have substantial buffering. This means that the packets received from the fabric by an egress PFE may be subjected to sufficient packet processing to classify them by flows in some richly detailed fashion and to enqueue each of those flows into separate queues. Once the packets are thusly enqueued, it becomes practical to apply complex transmission scheduling policies that respect flows on a per-customer, per-priority or per-whatever basis.

Of course, that second, egress-only packet buffer is not free. It requires memory resources and every storage and retrieval operation consumes power. Ideally, a packet is buffered just once and the fabric does not have to have generous speed-up built into it. This is where virtual output queues come in.

Virtual Output Queues

In a virtual output queue (VOQ) system, all of the buffering and queuing is performed on the ingress PFEs where there is one physical queue in each ingress PFE for every port and priority on all of the egress PFEs. For example, a system with 16 PFEs where each PFE had 32 ports and 8 priorities per port, each ingress PFE must have 16 × 32 × 8 = 4,096 queues. To simplify the remainder of this discussion, only PFEs and ports will be considered and not priorities.

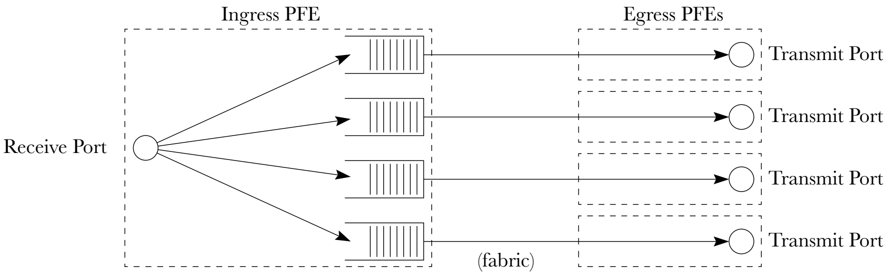


Figure 183 Virtual Output Queue Model (Common Understanding)

The benefit of having all of those queues in each ingress PFE is that it completely eliminates the head of line blocking problem. If a particular transmit port becomes overwhelmed with pending packets and must exert some kind of flow control on the ingress virtual output queues that are feeding into it, all of the other queues on every ingress PFE may continue to forward packets to the unaffected ports. Also, unrelated packets are only buffered once in a large memory system. This returns significant cost, complexity and power savings dividends.

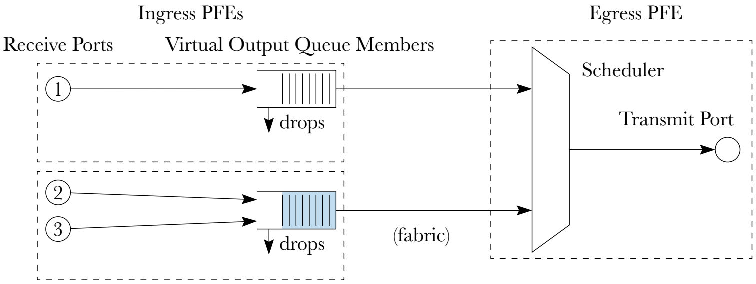


Figure 184 Virtual Output Queue Model (Single Transmit Port View)

Unfortunately, VOQs can behave unfairly in the face of congestion. To understand the nature of the problem, it is important to understand that each of the ingress

PFEs maintains a physical queue that is associated with a common transmit port (see preceeding Figure 184). So, if there are 16 ingress PFEs, there are 16 queues associated with a single transmit port. Each of these independent queues is a member of the same virtual output queue. They are, collectively, the output queue for that port. But, these VOQ members are operating independently, and that independence can lead to unfairness.

Consider a scenario where there is standing queue in one of the VOQ members due to some previous, temporary congestion. The queue is in a "standing" state because the receive and enqueue rate into the VOQ member (from port 2 in Figure 184) exactly matches the dequeue and transmission rate, so the prior congestion can never drain out. Now, at some point, a packet is received by a PFE whose VOQ member for the same VOQ is empty (port 1 in Figure 184). This newly-received packet immediately appears at the head of its previously-empty queue and is allowed to compete with the nearly full VOQ member of the other ingress PFE. If a simple round robin scheduling is used by the egress PFE, then that newly-received packet is allowed to cross the fabric almost immediately, and certainly far ahead of the bulk of the packets in the standing queue. In a true output-queue system, that newly-received packet is enqueued at the back of the standing queue and forced to wait its turn in line, as packets received via port 3 must do. The consequence of this is that two receive ports—1 and 3—may exhibit very different forwarding delay characteristics. With ideal output queue behavior, packets received via port 1 have identical delay characteristics as packets received via port 3.

There are a number of other such unfairness issues, but the one described above illustrates the point. The egress PFE that is granting access to the fabric for the various VOQ members must take the relative depths of the queues into account so that packets enqueued into shallow queues are forced to wait some amount of time before they are granted permission to cross the fabric, thus restoring ideal output queue behavior.

21 Conclusion

A reasonable argument can be made that a book like this is an impossible or pointless undertaking. After all, the topic is vast, the details innumerable and the rules ever-changing. Even just drawing a rough circle around the constellation of topics to include in the book presents a significant challenge. The raw data—the various protocols, headers, behaviors and the like—are all readily available via numerous standards documents; bolstering the case that there's not a lot of room for original writing in this space. Further, this book doesn't provide any ready-to-go solutions in the form of concrete algorithms, data structures or even detailed block diagrams of the internals of packet processing silicon. You certainly can't use this book as a sole reference when sitting down to design a network forwarding system simply because no single book could possibly anticipate and present every last detail of every consideration and aspect of a fully functioning hardware design.

However, I believe—and I hope you agree—that collecting, organizing and presenting the fundamentals of networking hardware's considerations and concepts in a consistent, cogent manner is a valuable and useful exercise. Yes, you could read hundreds of standards documents and research papers, and interview dozens of grizzled senior engineers and architects for their perspectives and wisdom and come to the same conclusions that I have presented in this book; and I encourage you to do those things. But it is my fervent hope that the chapters of this book accelerate that process and bring the complex and sometimes murky details of network structures and behaviors into sharp focus.

Ultimately, this book is about revealing and highlighting the patterns in networking. These patterns are rarely explicitly stated or described by the standards documents that incorporate them into their designs. These patterns have emerged and have been clarified over time as various experimental networking algorithms, protocols and methods have been tried and tested; with those that align with the natural patterns of networking enjoying success and those that stray from those patterns fading away. Chief among these patterns are:

- *The Forwarding Entity Axiom*

  A simple forwarding entity axiom applies to all forwarding protocols. This forwarding entity axiom states that: A forwarding entity always forwards packets in per-flow order to zero, one or more of a forwarding entity's own transmit interfaces and never forwards a packet to the packet's own receive interface. This simple description applies not only to the forwarding entities that we readily recognize as a discrete networking component such as a small, 5-port Ethernet bridge but also applies to vast EVPN networks that are built from hundreds or thousands of sophisticated chassis-scale routers. Viewing such complex networks through the lens of this axiom brings a significant clarity to understanding the functional and operational requirements of the systems used to build them.

- *Tunneling*

  It is a fundamental truth of engineering that abstraction, reduction, and simplification lead to powerful insights and results. As a youthful and curious engineer, my realization that tunnels are the building blocks of even the simplest of networks led to a clarity of thinking about network architectures and packet processing hardware architectures; facilitating the development of sophisticated and powerful silicon devices. Finding common patterns of behavior from one protocol to the next enables the sharing of resources and structures within a hardware-based forwarding system, leading to significant improvements in efficiency, performance and capabilities.

- *Virtualization*

  Virtualization complements tunneling by allowing multiple peer virtual networks to occupy a single tunnel without risk of accidental crossover from one virtual network to another, thus enabling such modern networking architectures as massively-scaled, multi-tenant data centers. This separation of one tenant from another also facilitates the easy remapping of a tenant from one set of physical resources to another, a vital capability as virtual machines are moved from server to server in a never ending process of performance and cost optimization.

Variations and combinations of these patterns underlie the vast majority of the concepts presented in this book. As new protocols and network architectures are developed, they will undoubtedly also make use of these patterns. If you, dear reader, make these patterns a part of your thinking and vocabulary you will be well equipped to make use of these patterns to quickly and easily adapt to future developments or, indeed, invent the future yourself.