

CS 168 Course Notes

By Peyrin Kao, based on lectures by Sylvia Ratnasamy, Rob Shakir, and others.

These are the course notes for CS 168: Computer Security at UC Berkeley.

Here is the official course description:

This course is an introduction to the Internet architecture. We will focus on the concepts and fundamental design principles that have contributed to the Internet's scalability and robustness and survey the various protocols and algorithms used within this architecture. Topics include layering, addressing, intradomain routing, interdomain routing, reliable delivery, congestion control, and the core protocols (e.g., TCP, UDP, IP, DNS, and HTTP) and network technologies (e.g., Ethernet, wireless).

Disclaimer: Beta

These notes have not been proofread. They likely contain errors.

If you're a CS 168 student at Berkeley, in any case of dispute, the official course lectures are the correct source of truth.

PDF Version

These notes are available in HTML form at <https://textbook.cs168.io>.

This PDF version is not always up-to-date. It was last updated on November 5, 2024. The HTML version is generally more up-to-date.

Corrections

As of the Fall 2024 semester, this textbook is still being actively maintained and updated.

If you see any parts that need to be corrected, please open a Github issue at <https://github.com/cs168-berkeley/textbook/issues>.

Source and Changelog

The source for the textbook and a log of all changes is available on Github at <https://github.com/cs168-berkeley/textbook>.

License

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Introduction to the Internet

What is the Internet?

The Internet is ubiquitous as a tool for transferring data between devices around the world. In this class, we'll be focusing on the infrastructure (both hardware and software) that supports this.

The Internet and the World Wide Web are not the same thing. You can think of the web as applications built on top of the Internet (e.g. Facebook, Twitter) that you can access through a web browser (e.g. Firefox, Chrome). Other applications besides the web can also use Internet infrastructure, too. Examples of non-web applications are Zoom or online games, or even Internet-of-things (IoT) devices like a sensor in your refrigerator or car.

Why is the Internet Interesting?

The Internet is not a new type of network technology (e.g. electrical wires already existed), but is instead about a completely new problem of tying together different, existing networks. Solving this problem required a new design paradigm that influenced other computer science fields.

Networking is a relatively new addition to the field of computer science. The Internet introduced many new challenges that are different from many traditional computer science fields. For example, unlike theory fields, we don't have a formal model of the Internet. Unlike hardware fields, we don't have a measurable benchmark for performance.

Unlike previous classes you might have taken, it's no longer enough to write code that simply works. The code you write has to scale to billions of users. The code you write also has to align with the business relationships of different operators (otherwise, they might not agree to run your code).

Code that works for a lightweight application (e.g. your home computer) might not work for a heavy-duty server. Code that works today might not work tomorrow, when different computers join and leave the network.

The design of the Internet has influenced the way in which we architect modern systems (e.g. reasoning about goals, constraints, and trade-offs in the design). Network architecture is more about thinking about designs, and less about proving theorems or writing code. It's more about considering tradeoffs, and less about meeting specific benchmarks. It's more about designing systems that are practical, and less about finding the optimal design. The Internet is not optimal, but has successfully balanced a wide range of goals.

The Internet is Federated

The Internet is a federated system, and requires interoperability between operators. In other words, each operator (ISP) acts independently, but every operator has to cooperate in order to connect the entire world. In other words, all the ISPs in the world need to agree on some common protocol(s) in order to achieve global connectivity.

Federation introduces several challenges. Competing entities (e.g. rival companies) are forced to cooperate, even though competitors might not want to share confidential information with each other. When designing

protocols, we have to consider real-life business incentives in addition to technical considerations.

Federation also complicates innovation. In other fields, companies can innovate by developing a new feature that no one else has. But on the Internet, if you have a feature that nobody else has, you can't use it. Everybody has to speak a common language (protocol), so any upgrades to the Internet have to be made with interoperability in mind.

The Internet is Scalable

Federation enables the tremendous scale of the Internet. Instead of a single operator managing billions of users and trillions of services, we only need to focus on interconnecting all the different operators. Federation also allows us to build the Internet out of a huge diversity of technologies (e.g. wireless, optical), with a huge range of capabilities (e.g. home links with tiny capacity, or undersea cables with huge capacity). These technologies are also constantly evolving, which means we can't aim for a fixed target (e.g. capacity and demand is constantly increasing by orders of magnitude).

The massive scale of the Internet also means that any system we design has to support the massive range of users and applications on the Internet (e.g. some need more capacity than others, some may be malicious).

The worldwide scale of the Internet means that our systems and protocols need to operate asynchronously. Data can't move faster than the speed of light (and often moves much slower than that). Suppose you send a message to a server on the other side of the world. By the time your message arrives, your CPU might have executed millions of additional instructions, and the message you sent might already be outdated.

The scale of the Internet means that even sending a single message can require interacting with many components (e.g. software, switches, links). Any of the components could fail, and we might not even know if they fail. If something does fail, it could take a long time to hear the bad news. The Internet was the first system that had to be designed for failure at scale. Many of these ideas have since been adopted in other fields.

Protocols

In this class, much of our focus will be on **protocols** that specify how entities exchange in communication. What is the format of the messages they exchange, and how do they respond to those messages?

For example, imagine you're writing an application that needs to send and receive data over the Internet. The code at the sender machine and the code at the recipient machine need to both agree on how the data is formatted, and what they should do in response to different messages.

Here's an example of a protocol. Alice and Bob both say hello, then Alice requests a file, and Bob replies with the file. To define this protocol, we need to define syntax (e.g. how to write "give me this file" in 1s and 0s), and semantics (e.g. Alice must receive a hello from Bob before requesting a file).

Different protocols are designed for different needs. For example, if Alice needs to get the file as quickly as possible, we might design a protocol without the initial hello messages. Designing a good protocol can be harder than it seems! We might also need to account for edge cases, bugs, and malicious behavior. For example, what if Alice requests a file, and Bob replies with hello? How should Alice respond?

Throughout this class, we'll see many protocols that have been standardized across the Internet. You'll

sometimes see the acronym RFC (Request For Comments) when we mention a protocol. Many standards are published as RFC documents that are eventually widely accepted, though not all RFCs end up adopted. RFC documents are numbered, and sometimes protocols are referred to by their RFC number. For example, “RFC 1918 addresses” refers to addresses defined by that particular document.

There are different standards bodies responsible for standardizing protocols. The IEEE focuses on the lower-layer electrical engineering side. The IETF focuses on the Internet and is responsible for RFCs.

Layers of the Internet

Layer 1: Physical Layer

In this section, we'll build the Internet from the bottom-up, starting with basic building blocks and combining them to form the Internet infrastructure. We'll use the postal system as a running analogy, since it shares many design choices with the Internet.

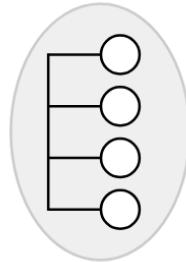
First, we need some way to send a signal across space. In the postal system, this could be a mailman, the Pony Express, a truck, a carrier pigeon, etc.

In the Internet, we're looking for a way to signal bits (1s and 0s) across space. The technology could be voltages on an electrical wire, wireless radio waves, light pulses along optical fiber cables, among others. There are entire fields of electrical engineering dedicated to sending signals across space, but we won't go into detail in this class.

Layer 2: Link Layer

In the analogy, now that we have a way to send data across space, we can use that building block to connect up two homes. We could even try to connect up all the homes in the local town.

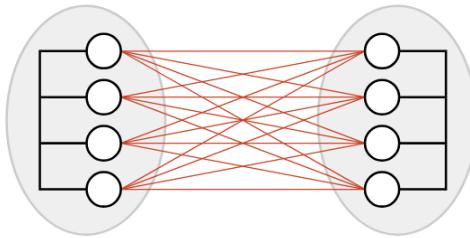
In the Internet, a **link** connects two machines. That link could be using any sort of technology (wired, wireless, optical fiber, etc.). If we use links to connect up a bunch of nearby computers (e.g. all the computers in UC Berkeley), we get a **local area network (LAN)**.



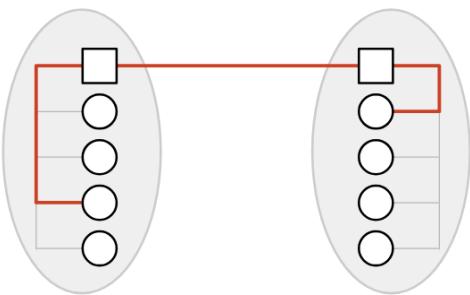
At Layer 2, we can also group bits into units of data called **packets** (sometimes called frames at this layer), and define where a packet starts and ends in the physical signal. We can also handle problems like multiple people simultaneously using the same wire to send data.

Layer 3: Internet Layer

We now have a way to connect everybody in a local area, but what if two people in different areas wanted to communicate? One possible approach is to add a bunch of links between different local networks, but this doesn't seem very efficient. (What if the two local networks were in different continents?)

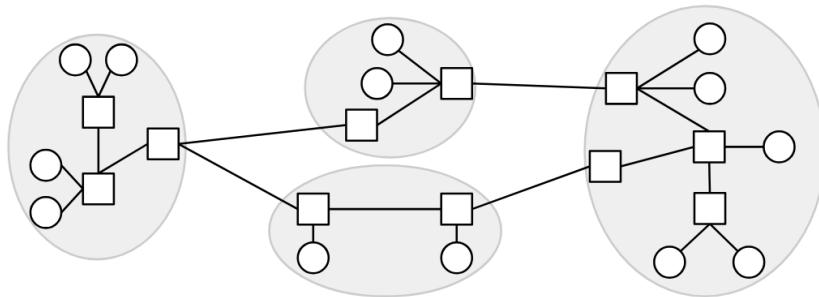


Instead, a smarter approach would be to introduce a post office in each network, and just connect the two post offices together. Now, if someone in network A wants to communicate with someone in network B, they can send the mail to the post office in network A. This post office forwards the mail to the post office in network B, which delivers the mail to the destination.



In the Internet, the post office receiving and redirecting mail is called a **switch** or **router**.

If we build additional links between switches, we can connect up local networks. With enough links and local networks, we can connect everybody in the world, resulting in the Internet.



One question we'll need to answer is how to find paths across a network. When a switch receives a packet, how does it know where to forward the packet, so that it gets closer to its final destination? This will be the focus of our routing unit.

We'll also need to make sure that there's enough capacity on these links to carry our data. This will be the focus of our congestion control unit.

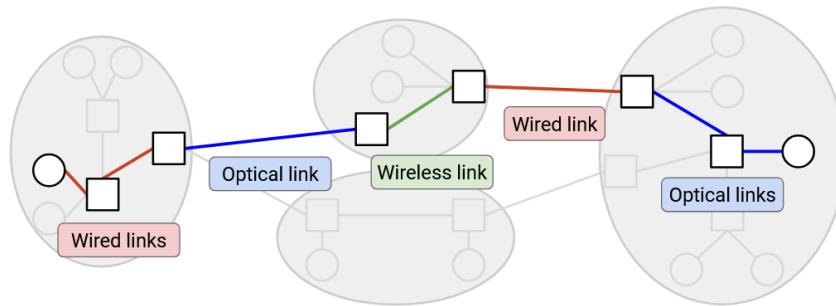
This picture now shows the infrastructure of the Internet, but in this class, we'll also study the operators managing the infrastructure. In the analogy, these are the people who build and manage the post office. On the Internet, the operators are **Internet service providers** like AT&T, Amazon Web Services, or even UC

Berkeley, who own and operate Internet structure. In addition to the hardware and software infrastructure, we'll need to think about these entities as real-life businesses and organizations, and consider their economic and political incentives. For example, if AT&T builds an undersea cable, they might charge a fee for other ISPs to send data through that cable.

Network of Networks

The Internet is often described as a **network of networks**. There are lots of small local networks, and what happens inside that local network can be managed locally (e.g. by UC Berkeley). Then, all the local networks can connect to each other to form the Internet.

In the network, different links might be using different Layer 2 technology. Some links might use wired Ethernet, and other links might use optical fiber or wireless cellular technology. At Layer 2, we figure out how to send a packet inside a local network, across the link(s) in that network, using the specific technology in that network. Then, at Layer 3, we use the ability to send packets along links as a building block to send packets anywhere in the Internet. As the packet hops across different networks, it may be transmitted across lots of different types of links.



In our analogy, we can see a distinction between homes and post offices. The homes are sending and receiving letters to each other. The post offices aren't sending or receiving their own mail, but they're helping to connect up the other homes.

In the Internet, **end hosts** are machines (e.g. servers, laptops, phones) communicating over the Internet. By contrast, a **switch** (also called a **router**) is a machine that isn't sending or receiving its own data, but exists to help the end hosts communicate with each other. Examples of switches are the router in your home, or larger routers deployed by Internet service providers (e.g. AT&T).

In these notes, we'll typically draw end hosts as circles, and routers as squares.

Layers of Abstraction

As we've built up the Internet, you might have noticed that we've been decomposing the problem into smaller tasks and abstractions.

"Modularity based on abstraction is the way things are done." (Barbara Liskov, Turing lecture). This is how we build and maintain large computer systems. Modularity is especially important for the Internet because

the Internet consists of many devices (hosts, routers) and many real-world entities (users, tech companies, ISPs), and having everybody agree on the breakdown of tasks is what enables the Internet to work at scale.

One major advantage of this layered, network-of-network approaches is, each network can make its own decisions about how to move data. For example, as your packet travels across Internet hops, some links might use wireless technology, and other links might use wired technology. The lower-layer protocols can change across different hops, and the Layer 3 protocol still works.

Layering also allows innovation to proceed in parallel. Different communities (e.g. hardware chip designers, software developers) can pursue innovation at different layers.

Layer 3: Best-Effort Service Model

It seems like we've built something that can send data anywhere in the world, so why not stop here? There are two issues with Layer 3 that we still need to solve.

The first issue involves the Layer 3 service model. If you use the Layer 3 infrastructure to send messages across the Internet, what service model does the network offer to you as a user? You can think of the service model as a contract between the network and users, describing what the network does and doesn't support.

Examples of practical service models might include: The network guarantees that data is delivered. Or, the network guarantees that data is delivered within some time limit. Or, the network doesn't guarantee delivery, but promises to report an error on failure.

The designers of the Internet didn't support any of those models. Instead, the Internet only supports **best effort** delivery of data. If you send data over Layer 3, the Internet will try its best to deliver it, but there is no guarantee that the data will be delivered. The Internet also won't tell you whether or not the delivery succeeded.

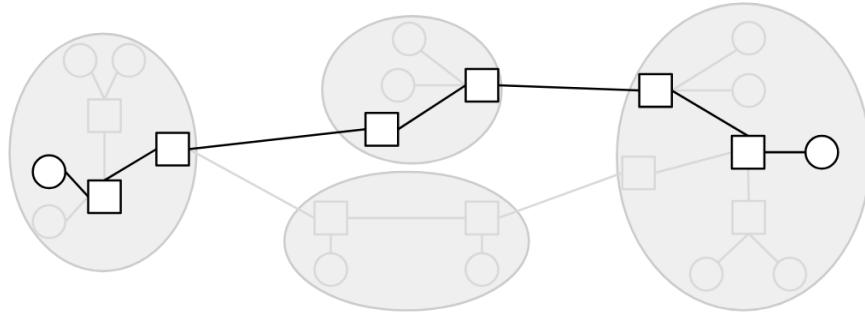
Why did the designers choose such a weak service model? One major reason is, it is much easier to build networks that satisfy these weaker demands.

Layer 3: Packets Abstraction

So far, up to Layer 3, we've been thinking about sending each message through the Internet independently. More formally, the primary unit of data transfer at Layer 3 is a **packet**, which is some small chunk of bytes that travel through the Internet, bouncing between routers, as a single unit.

The second issue at Layer 3 is: Packets are limited in size. If the application has some large data to send (e.g. a video), we need to somehow split up that data into packets, and send each packet through the network independently.

With this packet abstraction, we can now look at the life of a packet as it travels across the network. The sender breaks up data into individual packets. The packet travels along a link and arrives at a switch. The switch forwards the packet either to the destination, or to another switch that's closer to the destination. The packet hops between one or more switches, each one forwarding the packet closer, until it eventually reaches its destination. Note that because of the best-effort model, any of the switches might drop the packet, and there's no guarantee the packet actually reaches the destination.



Layer 4: Transport

We've identified two issues at Layer 3 so far. Large data has to be split into packets, and IP is only best-effort.

To solve both of these problems, we'll introduce a new layer, the **transport layer**. This layer uses Layer 3 as a building block, and implements an additional protocol for re-sending lost packets, splitting data into packets, and reordering packets that arrive out-of-order (among other features).

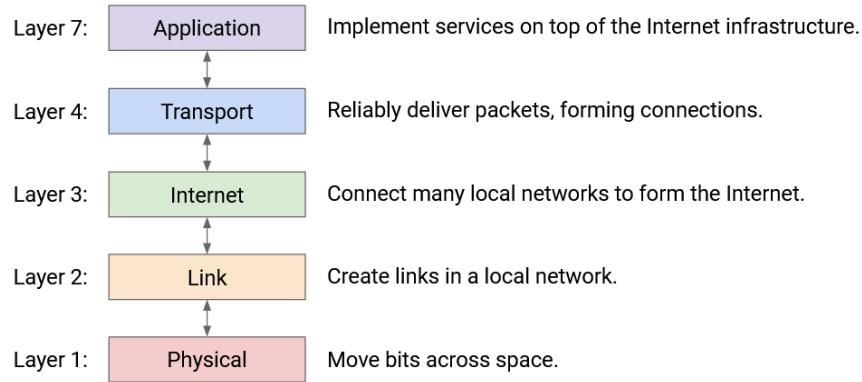
The transport layer protocol allows us to stop thinking in terms of packets, and start thinking in terms of **flows**, streams of packets that are exchanged between two endpoints.

Layer 7: Application

The application layer being built on top of the Internet is a powerful design choice. If, at lower layers, we built infrastructure for specifically transferring videos between end hosts, then email clients would have to build their own separate infrastructure for transferring emails. The Internet's design allows it to be a general-purpose communication network for any type of application data.

In this class, we'll focus more on the infrastructure supporting the application layer (e.g. the mailman, the post offices) and less on the applications themselves (e.g. the contents of the mail). We'll see some common application protocols near the end of the class, though.

Now that we've seen all the layers, notice that each layer relies on services from the layer directly below, and provides services to the layer directly above. For example, someone writing a Layer 7 (application) protocol can assume that they have reliable data delivery from Layer 4. They don't have to worry about individual packets being lost, since that's what Layer 4 already dealt with.



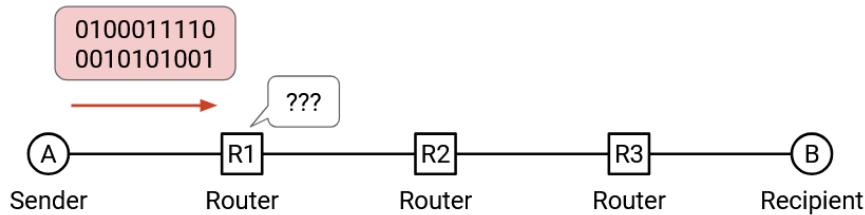
Two layers interact directly through the interface between them. There's no practical way to skip layers and build Layer 7 on top of Layer 3, for example.

Note: You might have noticed we skipped Layers 5 and 6. In the 1970s, when the layers were first standardized, the designers thought that these layers were needed, but they're obsolete in the modern Internet. If you're curious, the session layer (5) was supposed to assemble different flows into a session (e.g. loading various images and ads to form a webpage), and the presentation layer (6) was supposed to help the user visualize the data. Today, the functionality of these layers is mostly implemented in Layer 7.

Headers

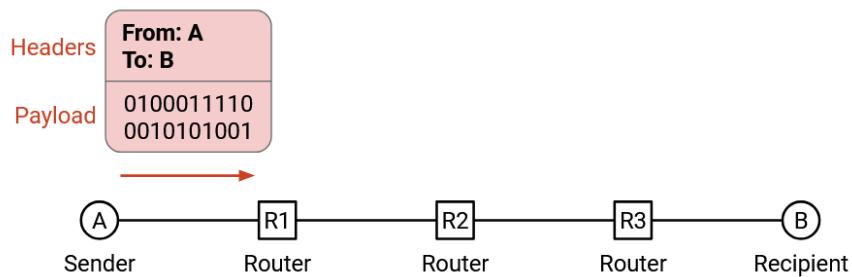
Why Do We Need Headers?

In the previous section, we saw that at Layer 3, data travels across the Internet in packets. Suppose an application wants to send a file over the Internet. We can take some bits of the image, put them in a packet, and send them over the Internet. When a switch receives this sequence of 1s and 0s, it has no idea what to do with these bits.



In the analogy, if I write a letter to my friend, and hand it to the post office, the post office has no idea what to do with it. Instead, we should put the letter inside an envelope, and write some information on the envelope (e.g. my friend's address) that tells the post office what to do with the letter.

Just like the envelope, when we send a packet, we need to attach additional metadata that tells the network infrastructure what to do with that packet. This additional metadata is called a **header**. The rest of the bits (e.g. the file being sent, the letter inside the envelope) is called the **payload**.



In the analogy, the post office shouldn't be reading the contents of my letter. It should only read what's on the envelope to decide how to send my letter. Similarly, the network infrastructure should only read the header to decide how to deliver the data.

The recipient cares about the inside of the letter, not the envelope. Similarly, the application at the end host cares about the payload, not the header. That said, the end hosts still need to know about headers, in order to add headers to packets before sending them.

Headers are Standardized

You can also think of headers as the API between the end hosts sending/receiving data, and the network infrastructure carrying the data. When we write software, we need to decide on the interface that users

will use to interact with our code (e.g. what functions users can call, the parameters to those functions). Similarly, the information in the header is how users access functions and pass parameters to the network.

Everybody on the Internet (every end host, every switch) needs to agree on the format of a header. If Microsoft Windows changes the code in its operating system to send packets with a different header structure, nobody else will understand the packets being sent.

This also means we need to be careful about designing headers. Once we design a header and deploy it on the Internet, it's very hard to change the design (we'd have to get everybody to agree to change it). This is why standards bodies can spend years designing and standardizing headers.

What Should a Header Contain?

What information should we put in the header?

The header should definitely contain the destination address, which tells us where to send the packet.

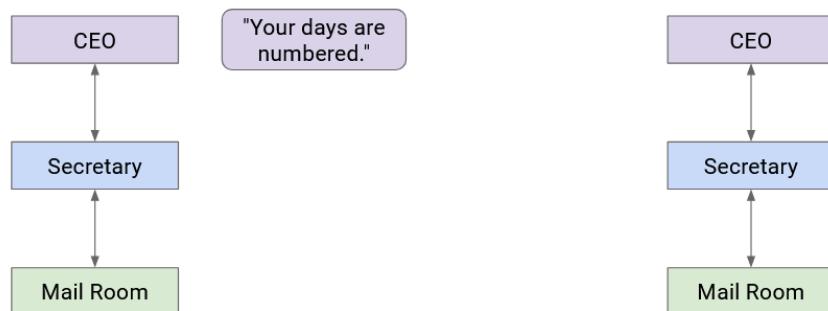
Headers could also contain other information that's not required, but is useful to have. Technically, the source address is not required to deliver the packet, but in practice, we almost always include the source address in the header. This allows the recipient to send replies back to the sender.

The header could also include a checksum, to ensure that packet is not corrupted while in transit.

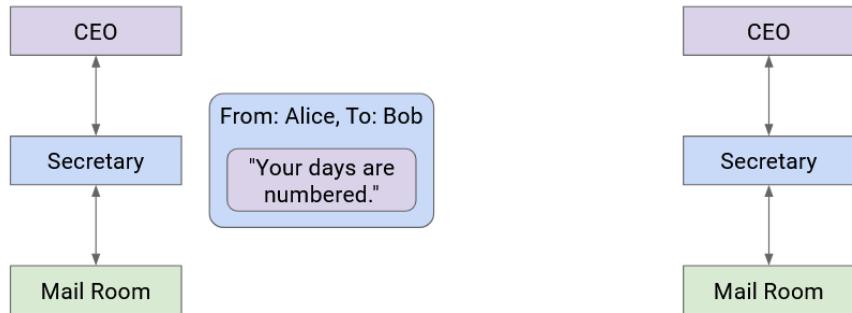
The header could also contain other metadata like the length of the packet. Note that packets can vary in size (e.g. the user might only need to send a few bytes).

Multiple Headers

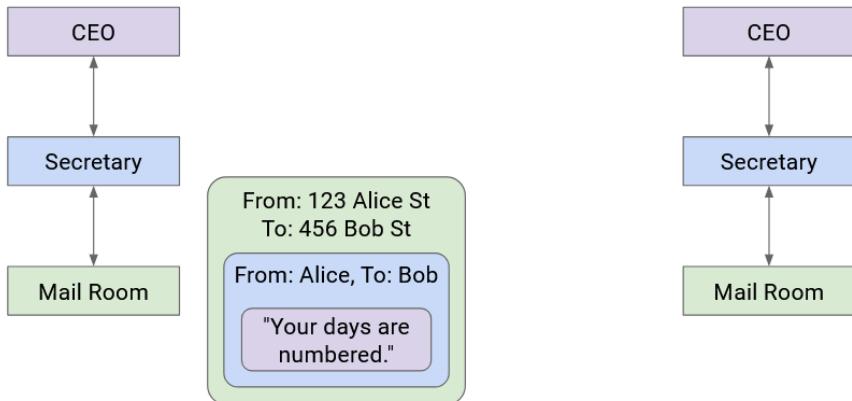
Let's go back to the postal analogy briefly. Suppose the boss of Company A wants to write a letter to the boss of Company B. How does the message get sent?



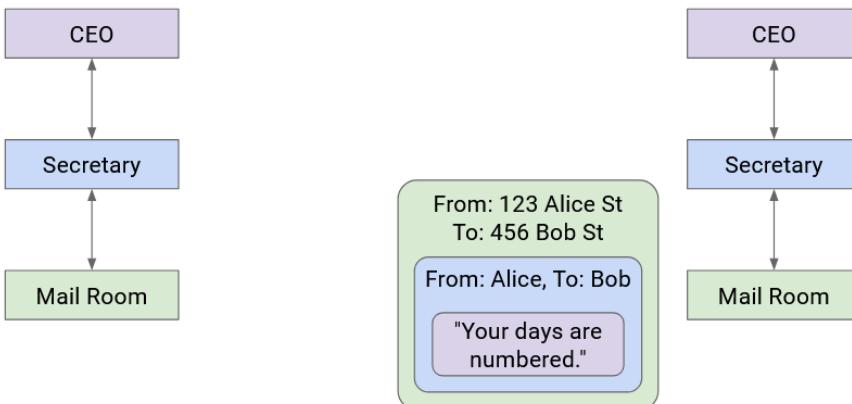
Company A's boss folds the letter and hands it to their secretary. Then, the secretary puts the letter in an envelope with Company B's boss's full name.



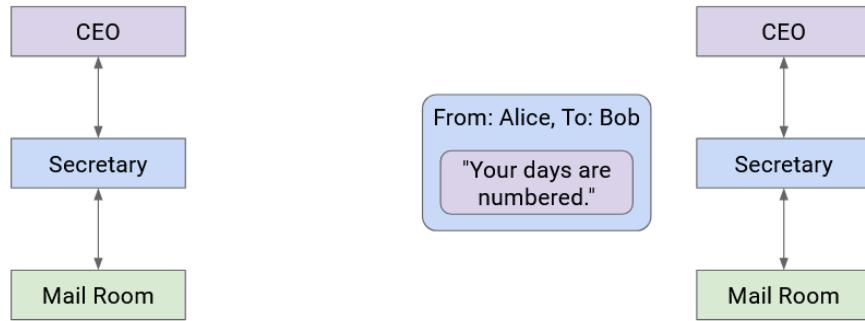
The secretary passes this letter to the mailroom. The postal worker puts the letter in a box with Company B's street address on it, and puts the package in a delivery truck.



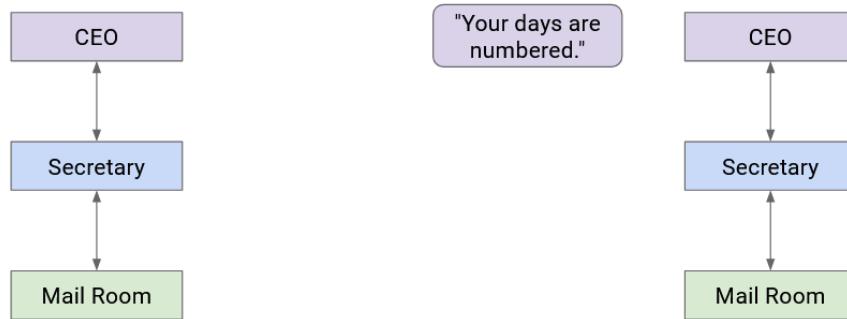
At this point, the letter itself is wrapped in multiple layers of identifying information (envelope, box). The delivery company sends the letter to Company B (possibly across several trucks, planes, mailmen, etc.).



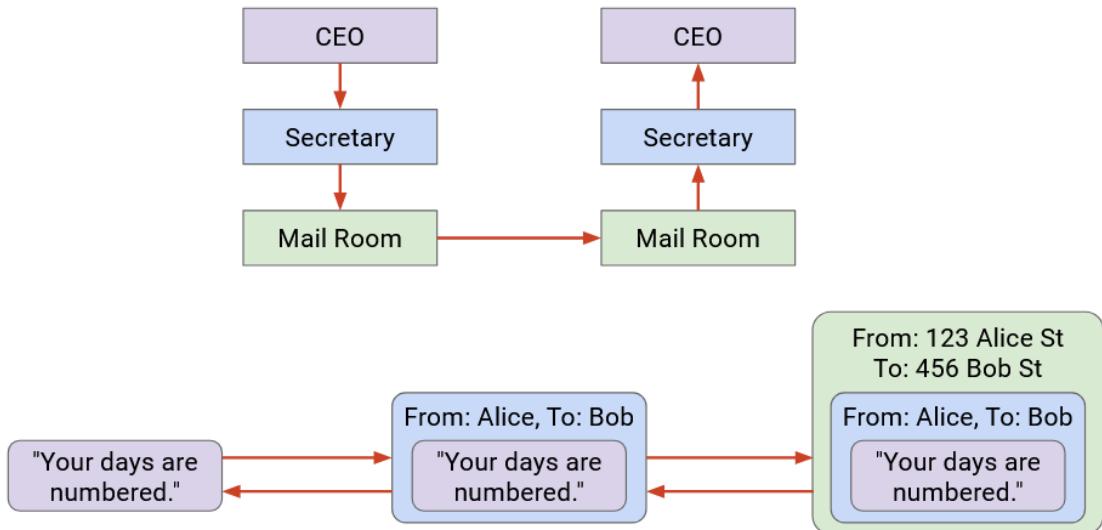
When the letter reaches Company B, the mailroom removes the box and passes the envelope to the secretary.



Then, the secretary sees the boss's name on the envelope, removes the envelope, and passes the letter up to the Company B boss.

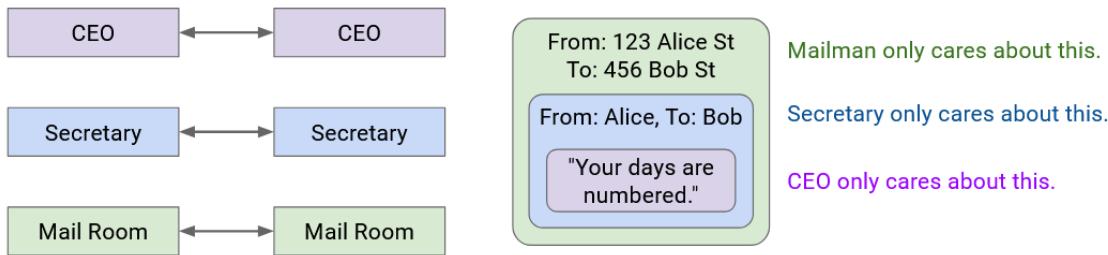


Notice that as we moved to lower abstraction layers, we wrapped more headers around the data. Then, as we moved to higher abstraction layers, we peeled layers off the data.



Each layer only has to understand its own header, and is “communicating” (in some sense) with its peers at the same layer. When Secretary A writes the name on the envelope, that's meant for Secretary B to read (not the mailmen, or the boss).

More formally, on the Internet, peers at the same layer communicate by establishing a protocol at that layer. The protocol only makes sense to entities at that specific layer.



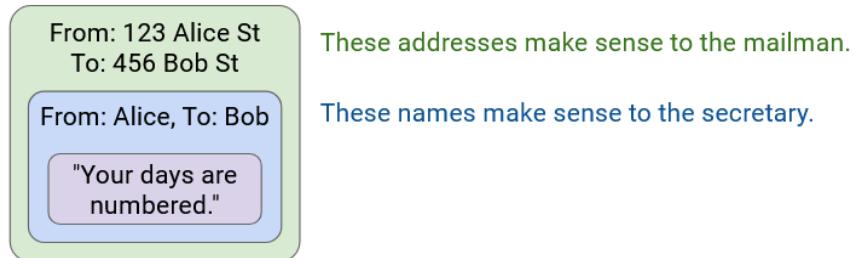
Note that some layers offer multiple choices of protocol (e.g. wireless or wired protocols at Layer 2). In these cases, the two people communicating need to use the same choice of protocol. A wired sender can't talk to a wireless recipient.

Addressing and Naming

Earlier, we said that our headers need to contain the address of the recipient. What actually is that address? Formally, a network address is some value that tells us where a host is located in the network.

As we look at the different layers in more detail, we'll see that different layers have different addressing schemes. If you want to send a letter inside Soda Hall, you could write the destination address as 413 Soda Hall, and the people in the building know where to deliver the letter. By contrast, if you want to send a letter to New York, you'd have to write a full street address like 123 Main Street, New York, NY.

Similarly, different layers in the Internet have different addressing schemes that work best for that particular layer. For example, sometimes a host is referred to by its human-readable name (e.g. www.google.com). Other times, that same host is referred to by a machine-readable IP address (e.g. 74.124.56.2), where this number somehow encodes something about the server's location (and could change if the server moves). Other times, that same host could be referred to by its hardware MAC address, which never changes.

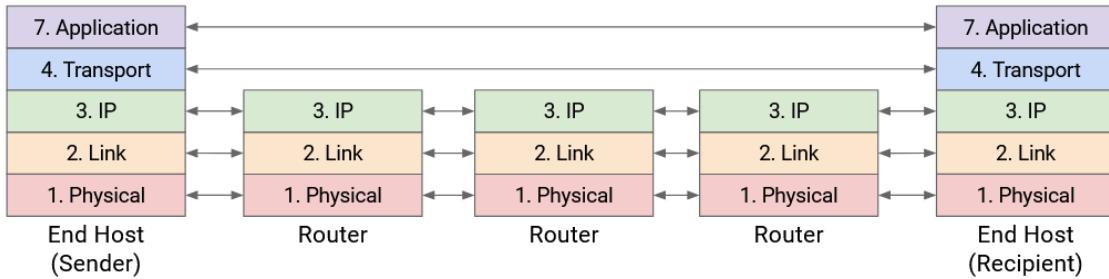


Layers at Hosts and Routers

The Internet is more than just a sender and a recipient. In addition to the two end hosts, there are routers forwarding the packet across multiple hops toward the destination. How do our ideas of layering and headers interact across all these machines?

The end hosts need to implement all the layers. Your computer needs to know about Layer 7 to run a web browser. Your computer also needs to know about Layer 1 to send the bits out along the wire. You'll also need all the layers in between in order for application-level data (the boss's letter) to be passed all the way down to the physical layer.

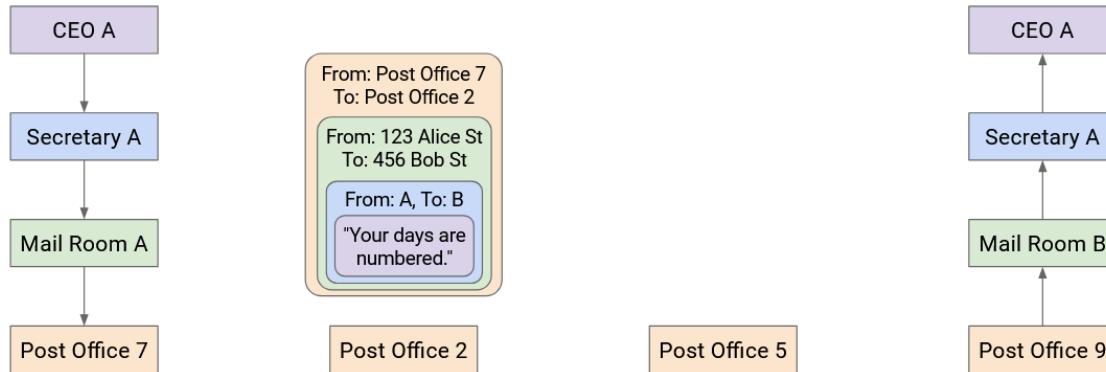
What about routers? The router does need Layer 1 for receiving bits on a wire, Layer 2 for sending packets along the wire, and Layer 3 for forwarding packets in the global network. However, the routers don't really need to think about Layer 4 and Layer 7. The router isn't running a web browser to display webpages, and the router doesn't need to think about reliability (recall, best-effort service model).



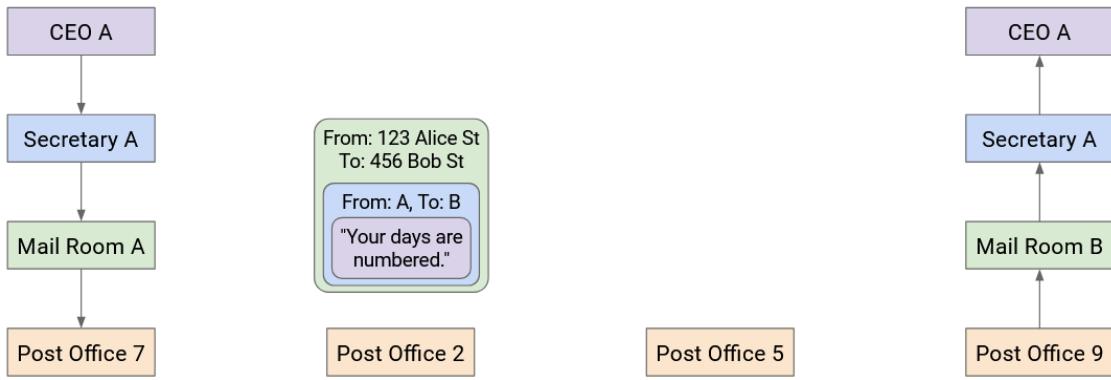
In summary: The lower 3 layers are implemented everywhere, but the top 2 layers are only implemented at the end hosts.

Multiple Headers at Hosts and Routers: Analogy

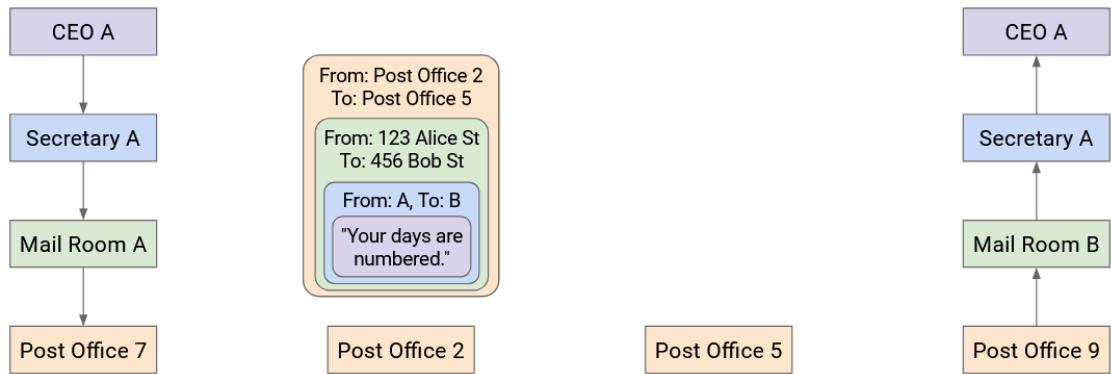
Let's think about sending mail again. Company A wrapped the letter in an envelope, which was then put in a box. The box doesn't magically travel to Company B. In fact, it might travel through several post offices.



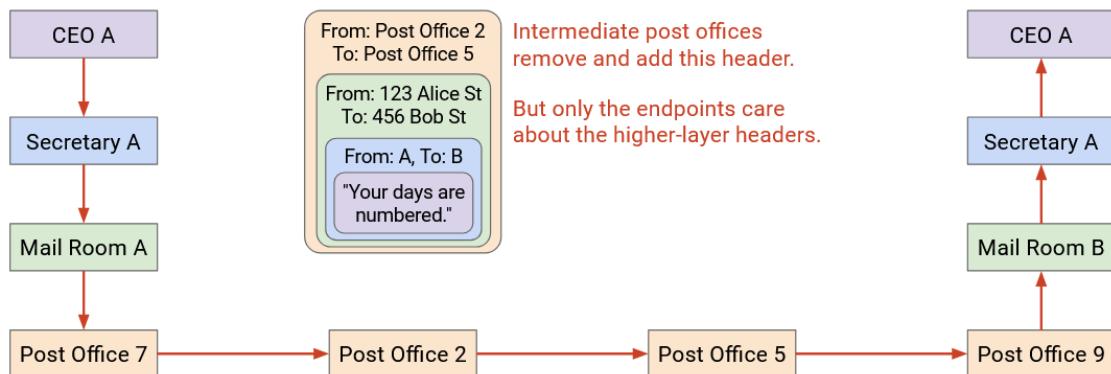
At each post office, the mailman opens the box and sorts through the mail. The mailman looks at the envelope (the next header revealed after opening the box), and sees that the envelope is meant for Company B.



The mailman then puts the envelope in another box, possibly different, so that the letter can reach the next post office on the way to Company B.



This process repeats at every post office. The box is opened, revealing the envelope inside. Then, the envelope goes in a new box, destined for the next post office. Notice that none of the post offices open the envelope to reveal the letter inside, because they don't need to read it.



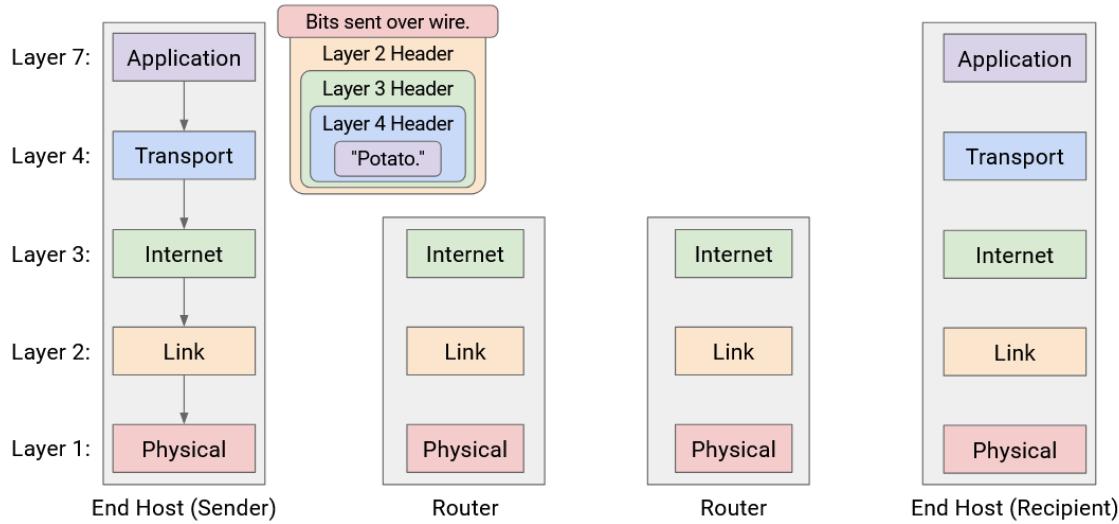
Eventually, the letter reaches Company B in a box, and this time, Company B opens the box, and the envelope, to reveal the letter inside.

Multiple Headers at Hosts and Routers

Now that we have the full picture with hosts and routers, let's revisit the demo of wrapping and unwrapping headers, as the packet takes multiple hops across the network.

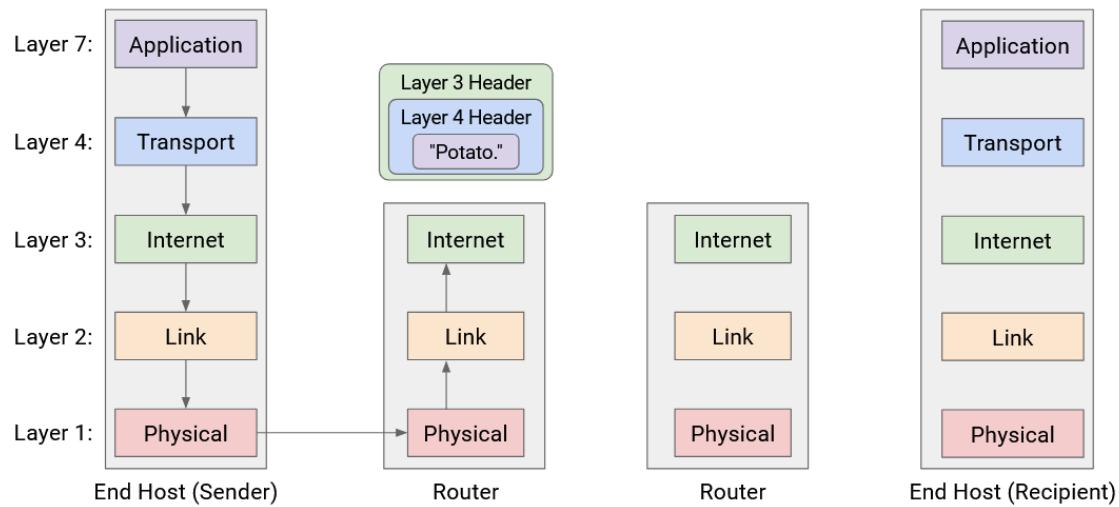
First, Host A takes the message and works its way down the stack, adding headers for Layer 7, 4, 3, 2, and 1. We now have a packet wrapped with headers for every layer.

The Layer 1 protocol sends the bits of this packet along the wire, to the first router on the way to the destination.

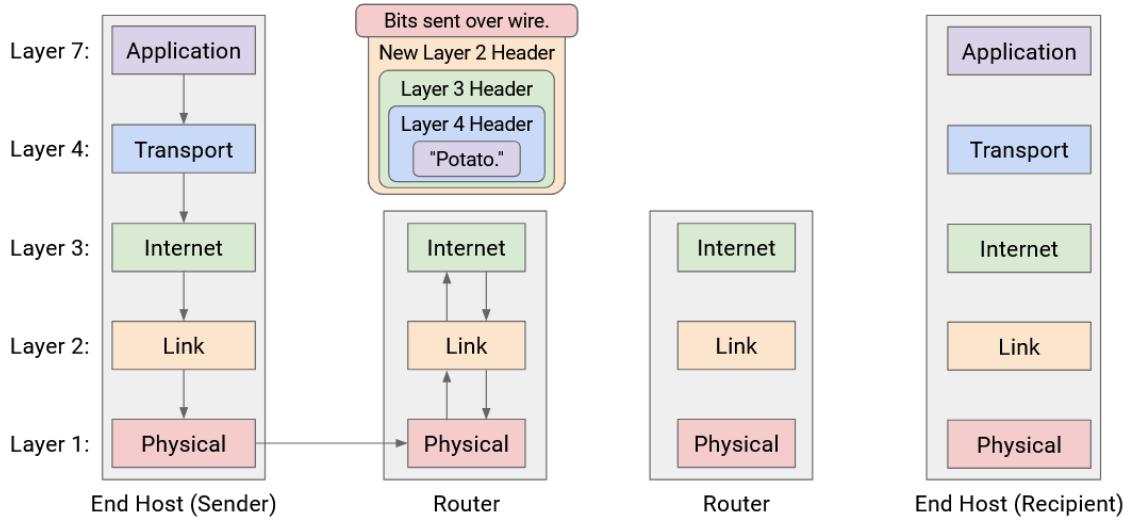


This router must forward the packet to the next hop, so that the packet eventually reaches Host B. We know that forwarding packets in the global network is a Layer 3 job. Therefore, the router must parse this packet up to Layer 3.

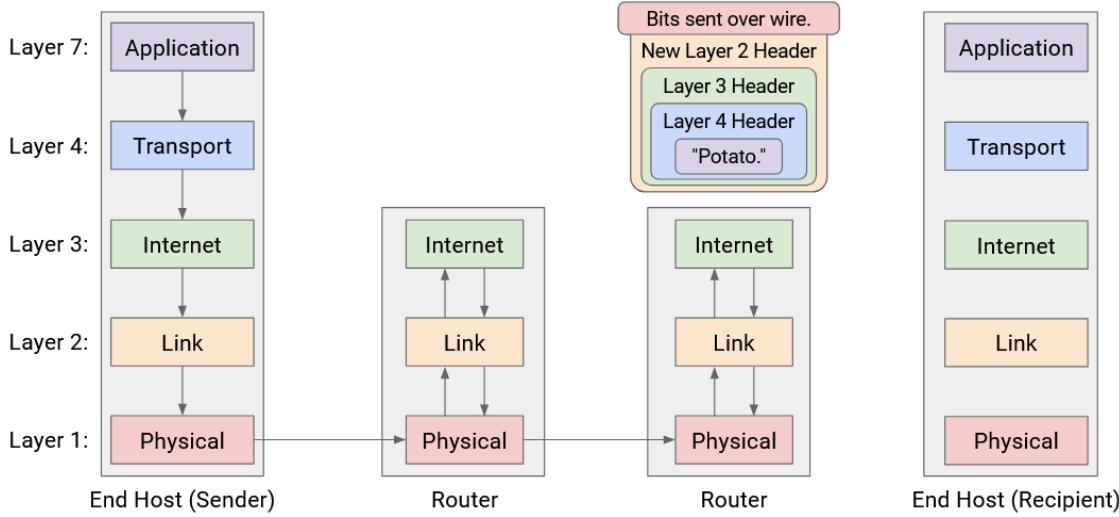
The router reads and unwraps the Layer 1 and Layer 2 headers, revealing the Layer 3 header underneath. The router reads this header to decide where to forward the packet next.



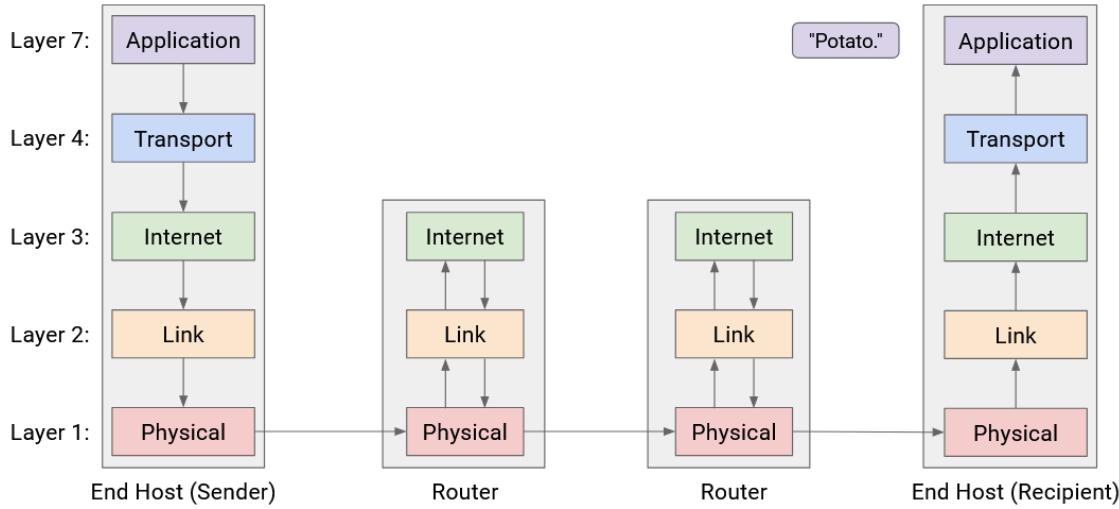
Now, to pass the packet along to the next hop, the router must go down the stack again, wrapping new Layer 2 and Layer 1 headers, and then sending the bits along the wire to the next hop.



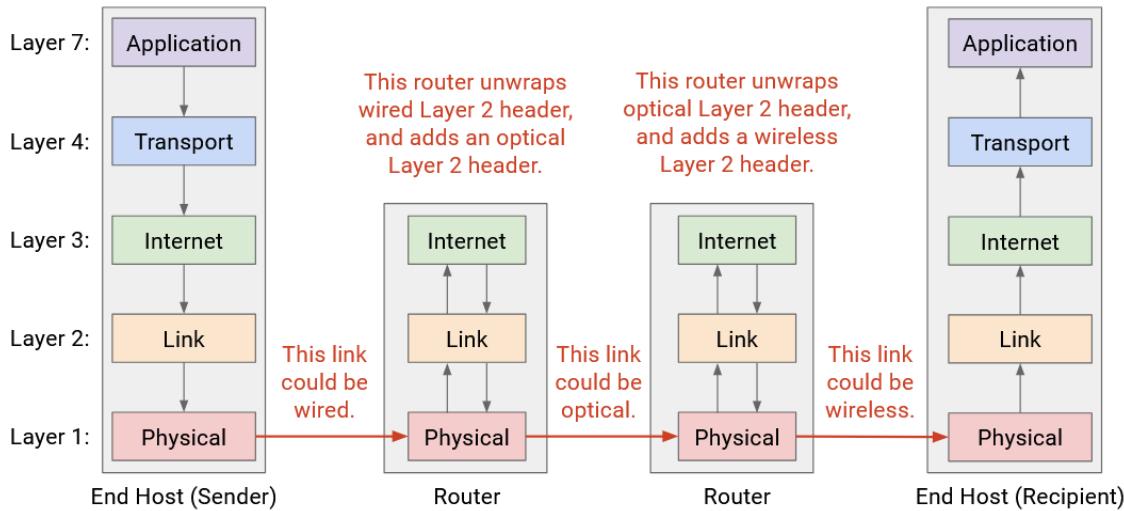
This pattern repeats at every router: Layers 1 and 2 are unwrapped to reveal the Layer 3 header, and then new Layer 2 and Layer 1 headers are wrapped before sending the packet to the next hop. Notice that none of the routers look beyond the Layer 3 protocol, because the upper layers are only parsed by the end hosts.



Eventually, the packet reaches Host B, who unwraps every layer, one by one: Layer 1, 2, 3, 4, 7. Host B has successfully received the message!

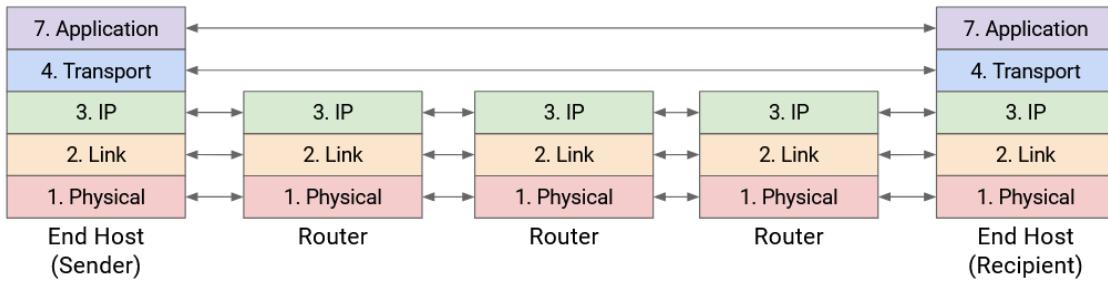


One consequence of this layering scheme is that each hop can use different protocols at Layer 2 and 1. For example, the first hop could get sent along a wire, and the initial Layer 2 and 1 headers used by Host A and the first router can be for a wired protocol. By contrast, a later hop could get sent along a wireless link, and the Layer 2 and 1 headers used by the routers on either end of that hop can be for a wireless protocol.

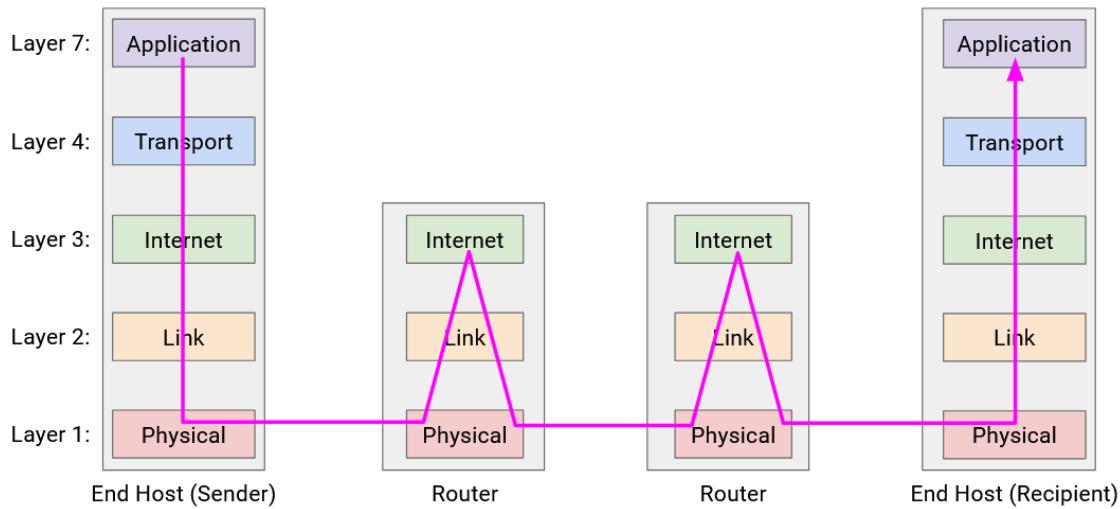


More generally, we said that each layer only needs to communicate with its peers at the same layer. We can now see this at play across all the layers. At Layers 4 and 7, the two hosts must speak the same protocols to send and receive packets. The host's peer is the other host.

By contrast, at Layers 1 and 2, the router must speak the same protocol as the previous-hop and the next-hop router, so that the router can receive packets from the previous hop and send packets to the next hop. The router's peers are its neighboring routers along the path.



In summary: Each router parses Layers 1 and 2, while the end hosts parse Layers 1 through 7.



Network Architecture

Design Paradigms

So far, we've seen a bottom-up view of the Internet, starting with fundamental pieces to build up the overall picture. In this section, we'll take a top-down view of the Internet, and analyze the overarching architectural choices in the design.

These Internet design paradigms influence why the Internet works the way it does, and also influences the applications we build on top of the Internet. These paradigms were a radical departure from how systems were historically built.

These designs are just one of many possible designs, and many design choices were made years ago, before the Internet grew to its current scale. Other designs exist, and debates still exist about what the best design is.

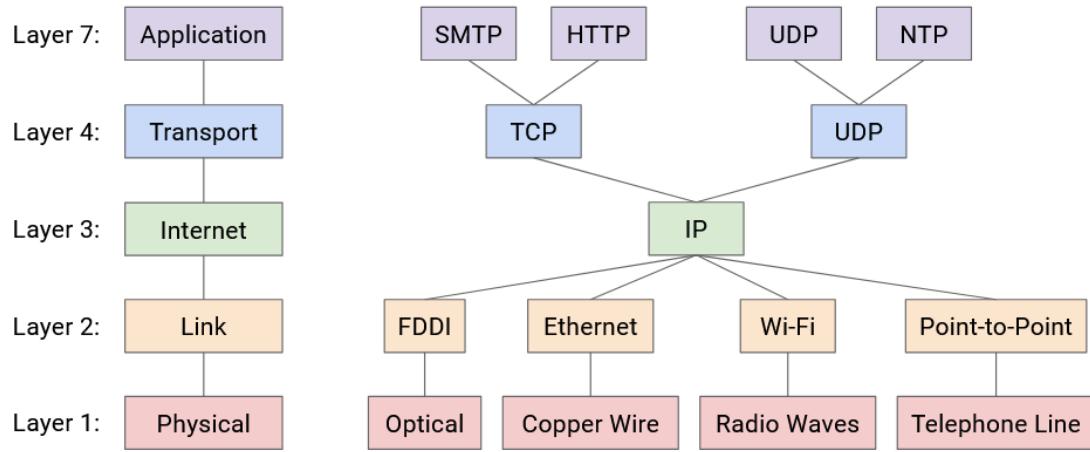
For example, the Internet was built to be federated (independent operators cooperating), but in recent years, software-defined networking (SDN) emerged as a more centralized approach to managing a network.

In the original Internet, switches were intentionally designed to be dumb and forward data without parsing it. However, in the modern Internet, attackers might try to overwhelm a switch by flooding it with useless data, and switches might need a way to detect this. Early Internet designers who came up with the dumb infrastructure paradigm did not consider this security implication.

Narrow Waist

It's possible to have multiple protocols at a given layer. For example, at Layer 7, we could use HTTP to serve websites, or NTP to sync system clocks, both built on the same Internet infrastructure. Or, at Layer 2, we could use Ethernet for wired networks, or Wi-Fi for wireless networks.

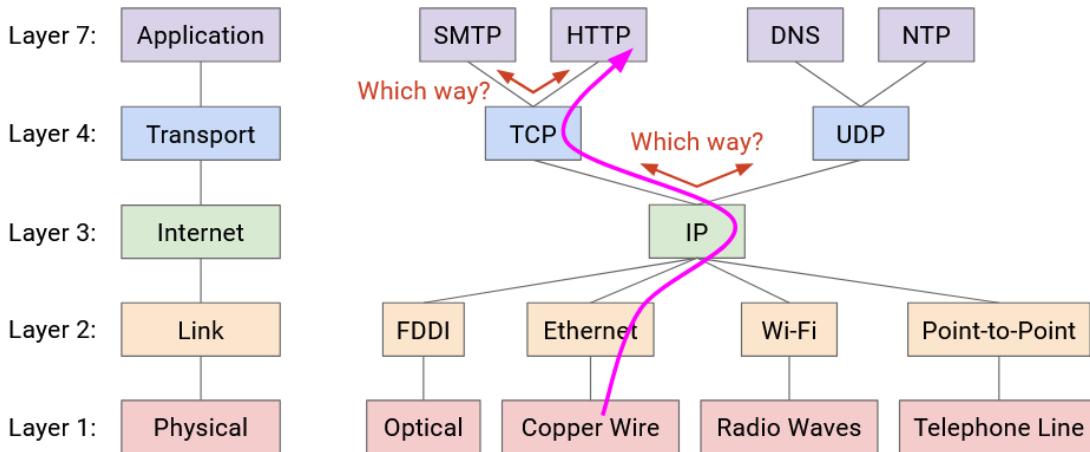
Note that even though there are multiple protocols at a given layer, you can commit to using a specific stack of protocols for your application. For example, you can commit to using HTTP over TCP over IP, and you don't need to use the other Layer 7 or Layer 4 protocols. Then, everybody using your application uses the same stack.

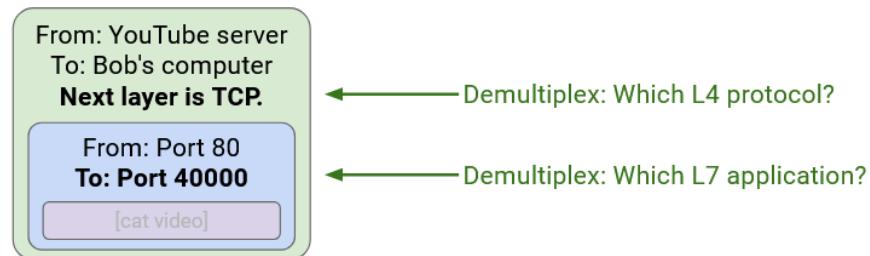
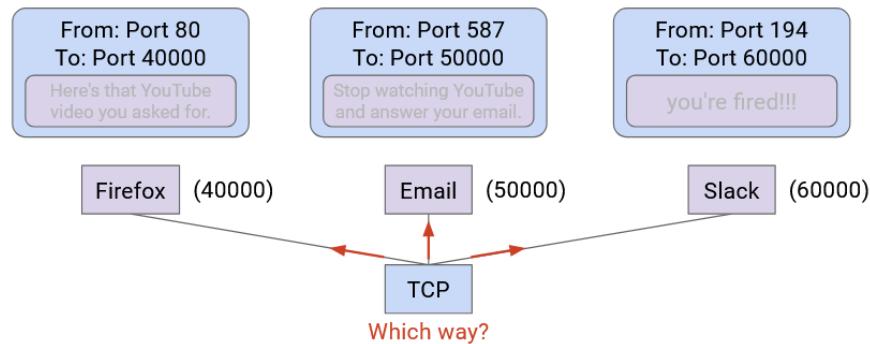
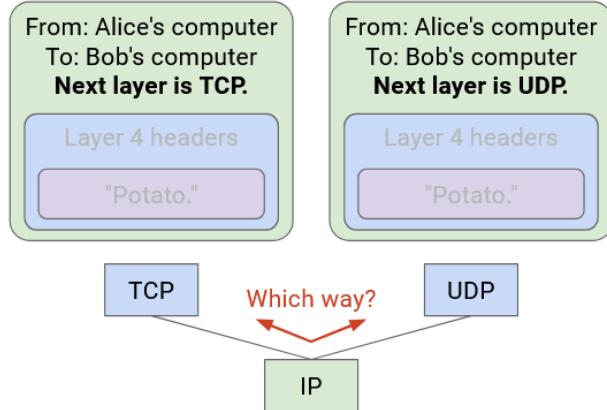


If you look at this diagram, you'll notice there's only one protocol at Layer 3. This is the "narrow waist" that enables Internet connectivity. Ultimately, everybody on the Internet must agree to speak IP so that packets can be sent across the Internet.

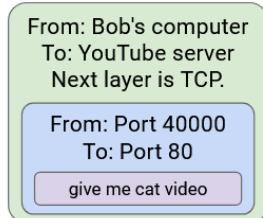
Demultiplexing

TODO write about demultiplexing.

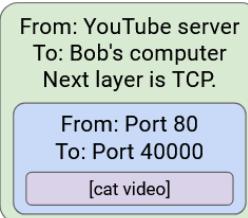




Outgoing packet:
Bob picks a random port number, but sends to YouTube's fixed port, 80.



Incoming reply: YouTube replies to Bob's chosen port. Bob's computer passes the packet to the correct application (Firefox, not Slack).



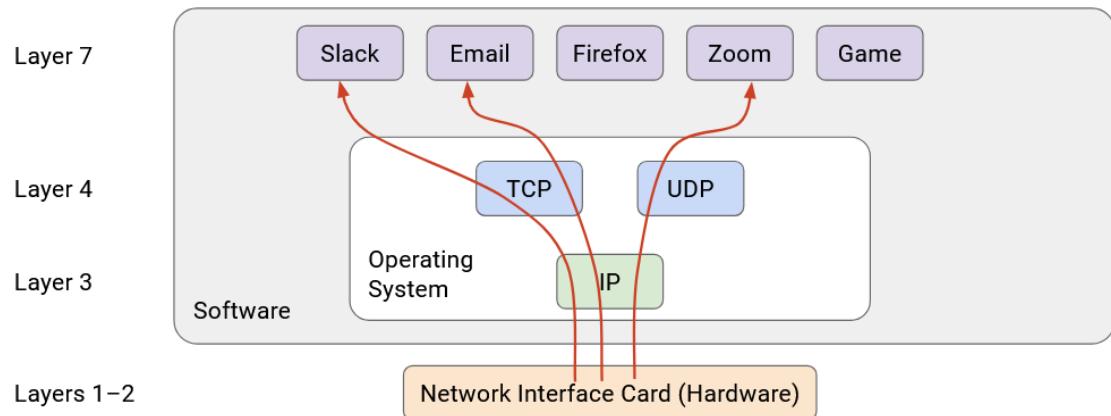
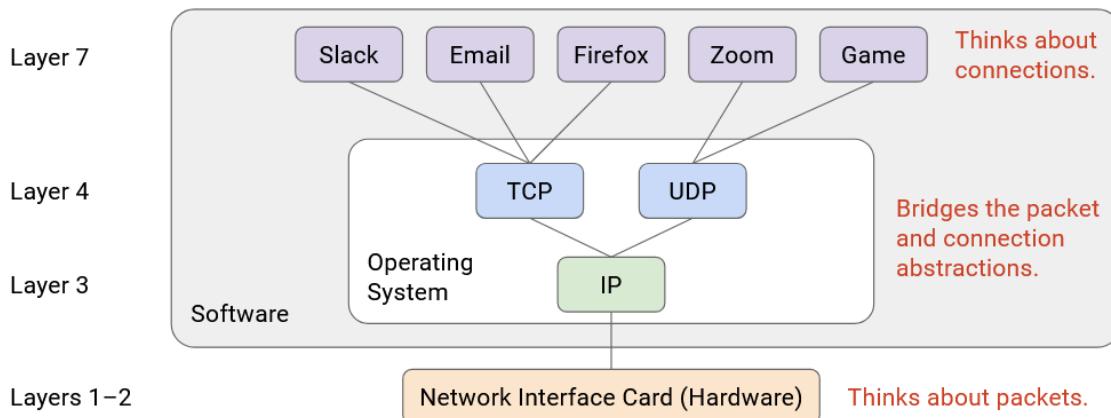
Be careful about naming. In networking, two different things are called ports. A physical port is the actual physical place where you plug a link into a switch. A logical port is a number in the Layer 4 header to disambiguate which application a packet belongs to.

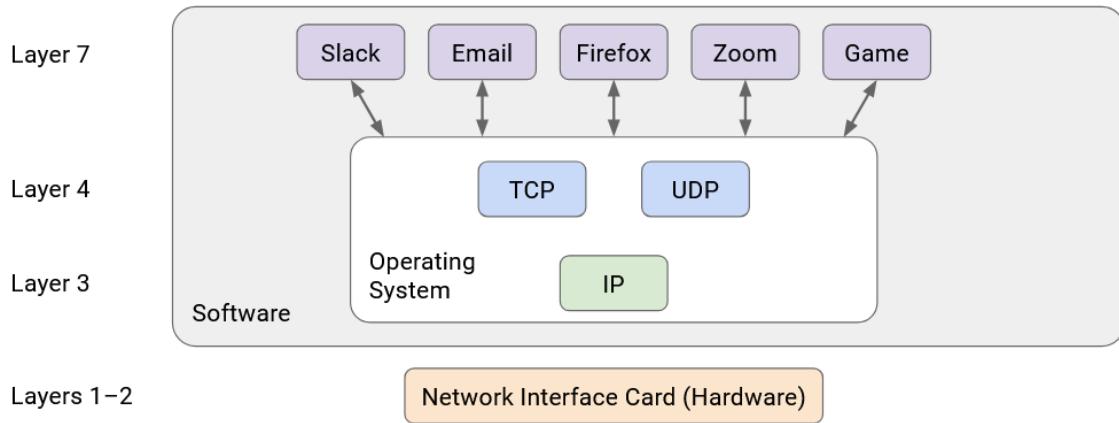


Logical port: A number identifying an application. Exists in software.

Physical port: The hole you plug a cable into. Exists in hardware.

Note: The term **socket** refers to an OS mechanism for connecting an application to the networking stack in the OS. When an application opens a socket, that socket is associated with a logical port number. When the OS receives a packet, it uses the port number to direct that packet to the associated socket.





End-to-End Principle

Why did we design the Internet with the layering structure that we did? Why do only the hosts understand Layers 4 and 7, and not the routers as well?

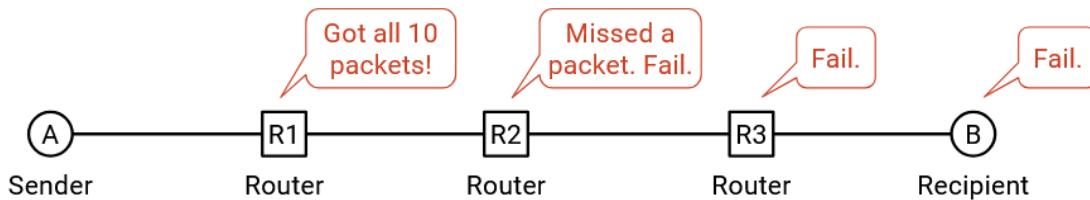
The **end-to-end principle** offers wisdom and guidance for designing the Internet. David D. Clark, a scientist at MIT and a member of the Internet Architecture Board, was a major contributor to this principle. Two of his papers, “End-to-End Arguments in System Design” (1981) and “The Design Philosophy of the DARPA Internet Protocols” (1988), were hugely influential on the philosophy of the Internet design.

The end-to-end principle guides the debate about what functionality the network does and doesn’t implement. The principle is quite broad and has many applications, but we’ll focus on the question of: Should we implement reliability (Layer 4) in the network, or only at the end hosts?

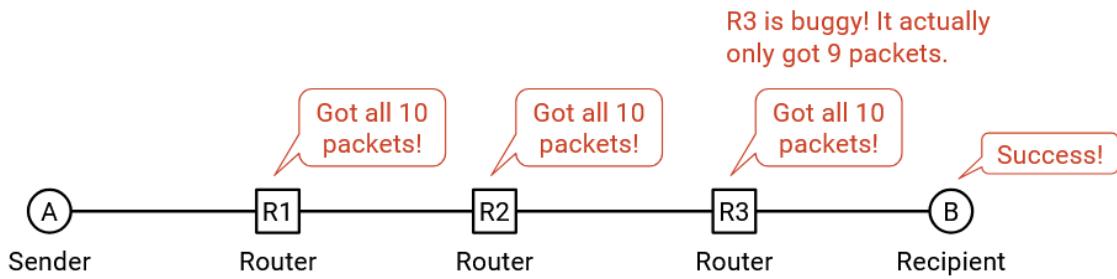
For now, let’s think of a simple protocol for reliability. Host A wants to send 10 packets to Host B, so it sends the packets, numbered 1 through 10, across the network. The goal is for B to either receive all the packets, or realize that some packets got lost and error (we’ll ignore recovering from the error).

What would the Internet look like if we implemented reliability in the network? Unlike our picture from earlier, every router must now understand Layer 4 in addition to Layers 1, 2, and 3.

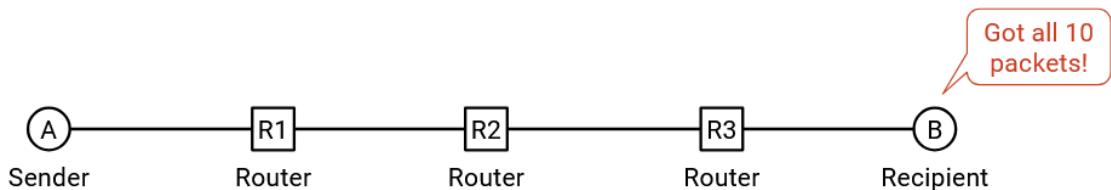
With this new picture, an intermediate router must reliably send a packet to its next hop. It must guarantee that the next hop received all the packets, and if not, the router must re-send any lost packets. The hosts don’t check that all packets were received, and instead rely on the network to ensure that all packets were received.



In this approach, the hosts have to trust the network. If one of the routers is buggy, and drops a packet, there’s nothing the hosts can really do about it.



The other approach is the end-to-end approach, where we do not implement reliability in the network, and we instead force the two end hosts to enforce reliability. Routers can drop packets, and it's up to the end hosts to verify that all packets were received.



In the end-to-end approach, where the end hosts implemented reliability, the control is with the hosts. The hosts could still be buggy and drop packets, but this time, the hosts have the power to fix the bug themselves. More generally, if you're writing code, it's better if you have the control over making the feature correct, instead of relying on other people who might mess up (and you can't fix their mistakes).

With this comparison in mind, if we used the first approach, where we relied on the network to be correct, we can't actually guarantee perfect reliability if the network is buggy. The end hosts would probably end up doing an end-to-end check (as in the second solution) anyway.

In the old Internet, every link did implement reliability. However, as we saw, the modern Internet only implements best-effort in the network, and forces the end hosts to implement reliability, in line with the end-to-end principle.

In summary: Some application requirements must be implemented end-to-end in order to ensure correctness. Also, the end-to-end implementation is sufficient, and no additional support from the network is needed. Because the end-to-end implementation alone is sufficient, adding network functionality would introduce additional unnecessary complexity (and cost), without helping us actually achieve the requirements.

Note that the end-to-end principle is not a proof or a theorem that's always true. It's a guiding principle and a philosophical argument, and different designers might make different arguments for or against the principle.

Here's an example of the end-to-end principle not being a strict rule. Even though the end-to-end principle says to implement reliability in the end hosts only, we could still add some extra reliability in the network in addition to the end-to-end check. This might be useful if we have highly unreliable links. Suppose there are 10 links between A and B, and each one fails 10% of the time. Then, each time we send the packet, it has a 65% chance of getting dropped. However, if each router was modified to send two copies of the packet for reliability purposes, each link only fails 0.1% of the time, and packets now only have a 1% chance of

getting dropped. Wireless links will sometimes implement reliability to reduce error rates and improve performance for the end hosts.

The end-to-end principle extends to other fields as well. For example, in security, the end-to-end principle might say that two end hosts communicating should encrypt their messages at the end hosts, instead of at intermediate points in the network.

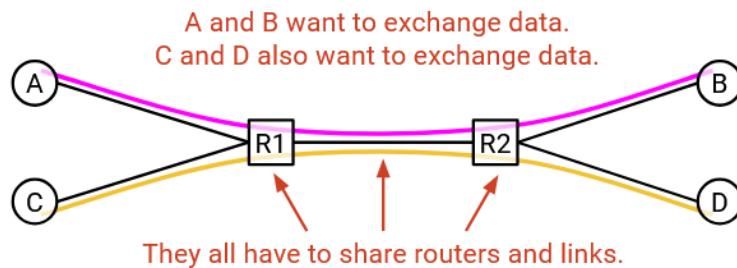
The end-to-end argument in Clark's words: "The function in question can completely and correctly be implemented only with the knowledge and help of the application at the end points. Therefore, providing that function as a feature of the communication system itself is not possible. Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement."

Designing Resource Sharing

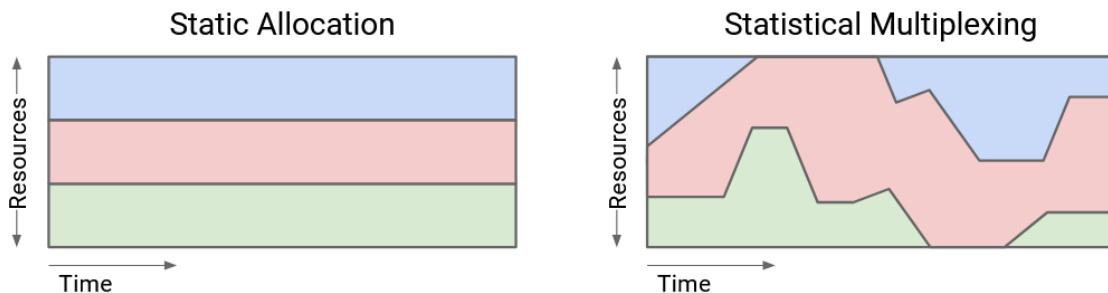
Sharing Resources: Statistical Multiplexing

Links and switches on the Internet have finite capacity. One key design problem we need to solve is: How do we share these resources between different Internet users?

Let's formalize the problem a bit more. Recall that a flow is a stream of packets exchanged between two end hosts (e.g. a video call between you and a friend). The Internet needs to support many simultaneous flows at the same time, despite limited capacity.



We often say that the network resources are **statistically multiplexed**, which means that we'll dynamically allocate resources to users based on their demand, instead of partitioning a fixed share of resources to users.



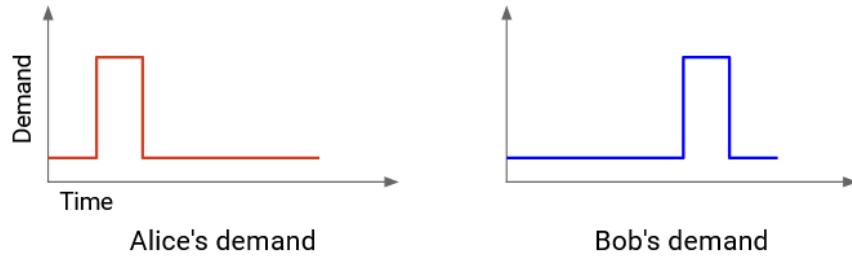
As an analogy, consider your personal computer. It's not the case that your computer preemptively allocates half its CPU to Firefox, and half its CPU to Zoom, and only allows each application to use its half of the CPU. Instead, your computer dynamically allocates resources to different applications depending on their needs.

Statistical multiplexing is now everywhere in computer science. For example, in cloud computing, different companies might dynamically share resources in a datacenter.

Statistical multiplexing is a great way to efficiently share network resources, because user demand changes over time. You probably aren't using a constant 10 Mbps of bandwidth every second, 24 hours a day. You probably have more demand while you're awake, and less while you're sleeping.

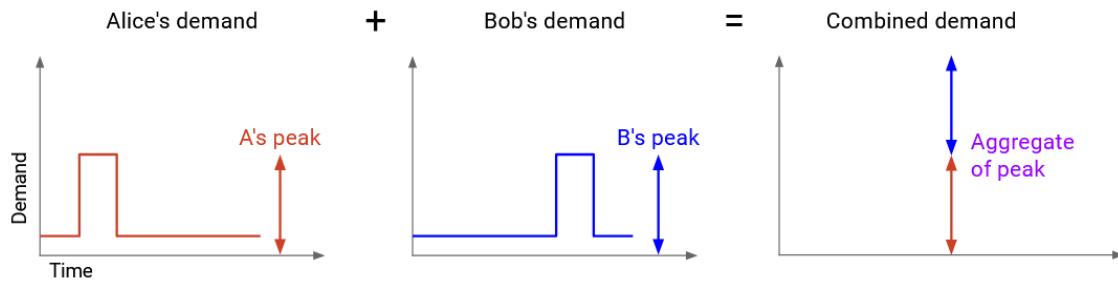
The premise that makes statistical multiplexing work is: In practice, the peak of aggregate demand is much less than the aggregate of peak demands.

Let's unpack what this means. Suppose we have two users, A and B. We can plot each user's demand over time.

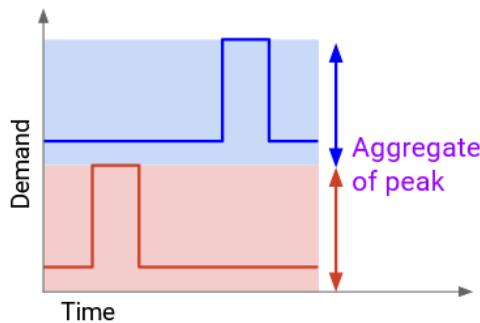


How much capacity do we need to allocate in order to fully meet both users' demands?

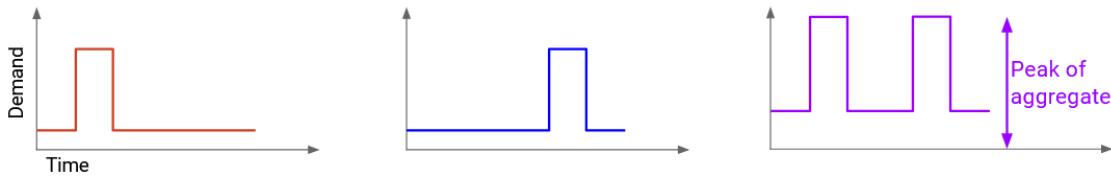
The bad strategy (no statistical multiplexing) is to compute the aggregate of peak demands. We find A's peak demand and B's peak demand, and add them together.



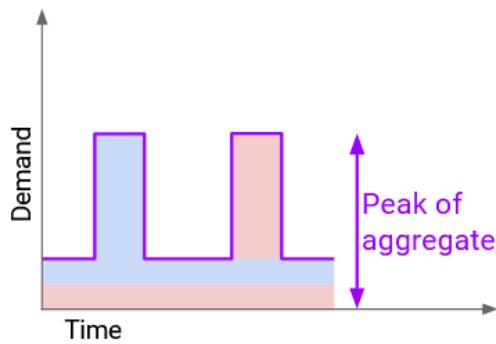
If we allocate this much capacity, we can definitely meet their demands. A's peak demand is X, so we allocate X to A, and likewise, we allocate Y to B. However, this approach is wasteful, because A's peak and B's peak didn't happen at the same time.



The better strategy (statistical multiplexing) is to first compute the aggregate demand by graphing their combined demand over time. For example, the 10am demand in the new graph is the A's 10am demand, plus B's 10am demand. Then, we compute the peak of the aggregate demand.



If we allocate this much capacity, we can no longer statically allocate a portion to each user. However, by dynamically changing the amount we allocate to each user over time, we can still successfully meet their demands, even while having less capacity.



The statistical multiplexing approach allows us to support the same users with less capacity (cheaper for us, more efficient use of resources). For many distributions, we can show that the peak of the aggregate is actually closer to the sum of the average demands, which is much less than the sum of the peak demands.

In practice, in the network, we don't provision for the absolute worst case, when everything peaks at the same time. Instead, we share resources dynamically and hope that peaks don't occur at the same time. Peaks could still happen at the same time, which would cause packets to be delayed or dropped (recall the link queue). Nevertheless, we made the design choice to statistically multiplex and use resources more efficiently, while dealing with the consequences (occasional simultaneous peaks).

At the end of the day, statistical multiplexing is a design choice with trade-offs, and different users might make different choices. For example, financial exchanges sometimes decide to build their own dedicated networks to support peak demand, because they care more about ensuring network connectivity during peak periods, and they can afford the extra cost.

Sharing Resources: Circuit Switching vs. Packet Switching

We now know that we can use statistical multiplexing to decide how much capacity to build. Our next question is: How do we actually dynamically allocate resources between users?

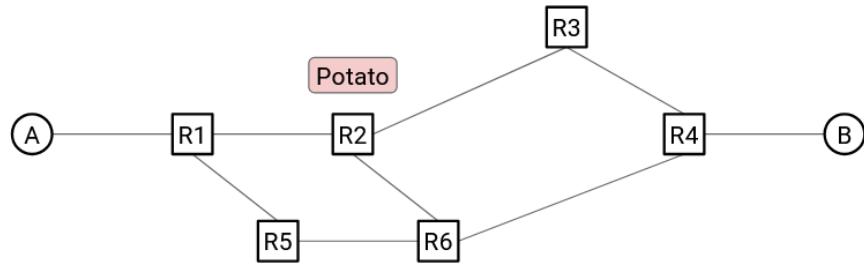
As an analogy, consider a popular restaurant with many customers and a limited supply of tables. There are two ways we could imagine allocating tables to customers. We could have customers make reservations, or we could seat customers first-come first-serve.

The two approaches to sharing resources in the network are similar. One approach is **best-effort**. Everybody

sends their data into the network, without making any reservations, and hopes for the best. There's no guarantee that there will be enough bandwidth to meet your demand.

The canonical design for best-effort is called **packet switching**. The switch looks at each packet independently and forwards the packet closer to its destination. The switches don't think about flows or reservations.

In addition to packets being independent from each other, the switches are also independent from each other. As a packet hops across switches, every switch considers the packet independently (the switches don't coordinate).

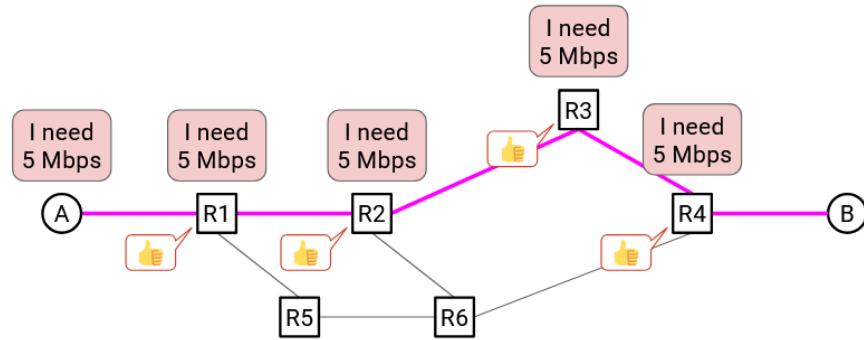


The other approach is based on **reservations**. At the start of a flow, users explicitly request and reserve the bandwidth they need. After the data is sent, the resources can be released for others to reserve.

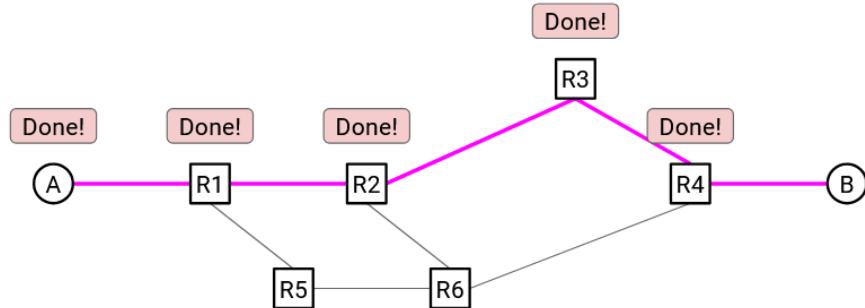
The canonical design for reservations, explored in both research and industry, is called **circuit switching**.

At the start of a flow, the end hosts identify a path (sequence of switches and links) through the network, using some routing algorithm. (We haven't discussed routing algorithms to find this path yet, so you can assume it happens by magic for now.)

Then, the source sends a special reservation request message to the destination. Along the way, every switch hears about this request as well. If every switch accepts the request, then the reservation is made, and a circuit of switches has been established between the source and destination.

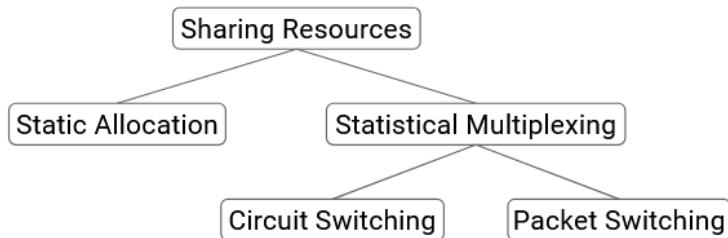


Once the reservation is confirmed by every switch, data can be sent. Eventually, when the flow ends, the source sends a teardown message to the recipient. Along the way, every switch sees this message and releases its capacity.



Note: We use the term circuit here because this idea came from the phone network, which uses this same idea to allow two people to call each other.

Remember, both circuit switching and packet switching are embodying statistical multiplexing. The main difference is the granularity at which we're allocating resources: per-flow with reservations, or per-packet with best-effort. Even in circuit switching, we're dynamically allocating resources based on reservations. We are not preemptively reserving for all flows that might ever exist.



Circuit Switching vs. Packet Switching Trade-offs

We now have two approaches to sharing resources on the Internet. Which is better? It depends on the criteria we're using to evaluate each approach.

There are four dimensions we can use to compare the two approaches.

1. Is this a good abstraction (or API) for the network to offer to an application developer?

Circuit switching offers a more useful abstraction to developers, because there's a guarantee of reserved bandwidth. This gives the developer more predictable and understandable behavior (assuming all goes well). As an analogy, consider reserving a machine in the cloud to run some task. It's easier for the developer to reason about performance if they know the specs of the machine they're getting. If the developer had no idea what machine they were using, the task could still run, but the performance is less predictable.

Circuit switching is also a useful abstraction if you're a network operator who has to distribute resources to users. You know exactly how much bandwidth each user is requesting, and you can charge them the appropriate amount of money. It's a little harder to implement an intuitive business model if there are no guarantees about what you're offering to a client.

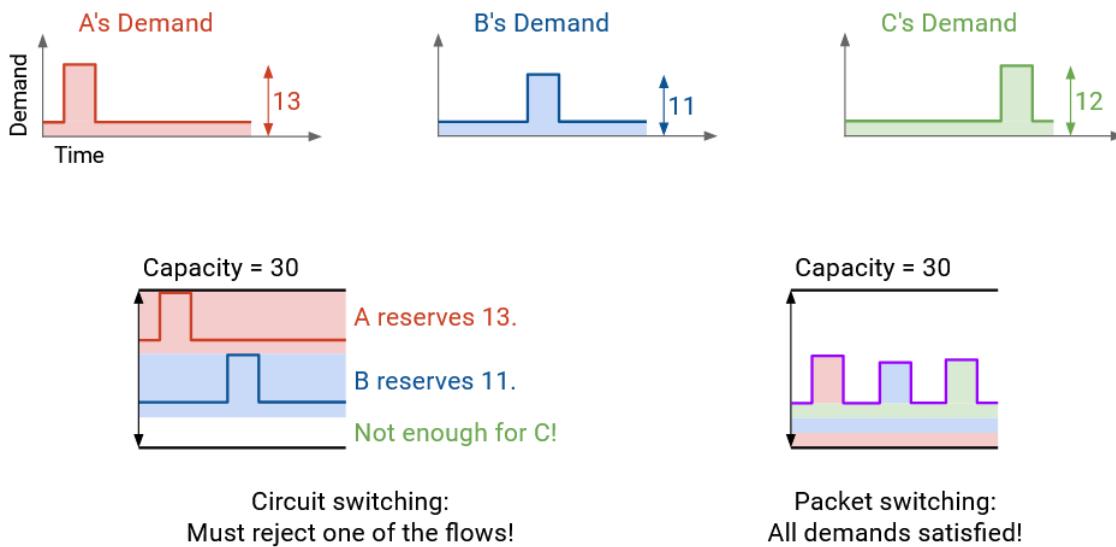
2. Is the approach efficient at scale? Does the approach use all the available bandwidth on the network, or is some bandwidth wasted?

Packet switching is typically more efficient. Exactly how much better depends on the burstiness of the traffic sources.

If each sender sends data at a constant rate throughout time, then both circuit switching and packet switching makes full use of the capacity.



By contrast, if each sender's rate varies over time, then packet switching gives us a better use of bandwidth.



Here's an example of demand varying over time. With reservations, the three flows must reserve 12, 11, and 13 Mbps. One of the reservations will be rejected, since we can only distribute 30 Mbps.

This approach is wasting bandwidth in two different ways. The flow reserving 12 Mbps is not actually using its bandwidth for most of its time. Also, if the 12 Mbps and 11 Mbps flows get reservations, we have 7 Mbps left over that isn't being reserved by anybody.

By contrast, in the packet switching approach, where we just send packets as they arrive, the total amount of bandwidth being used at any time never exceeds 30 Mbps. We can support every flow with the bandwidth we have.

Formally, the burstiness of a flow is defined by the ratio between its peak rate and its average rate. There's no clear threshold for when something is smooth or bursty (they're more descriptive terms).

Voice calls usually have smoother ratios like 3:1, while web browsing usually has burstier ratios like 100:1. (Voice calls having a smooth ratio is also why the phone network used reservations!)

Another reason why packet switching is more efficient is: Circuit switching spends additional time setting up and tearing down a circuit. This is especially inefficient for very short flows (e.g. downloading a tiny file).

3. How well does each approach handle failure at scale?

Packet switching is better at handling failure at scale. If a router fails, we can just send packets along a different path in the network. (We haven't discussed how yet, but it turns out routing algorithms are good at adjusting to failure.) The end host doesn't have to do anything different.

By contrast, in circuit switching, if a router along the path fails, the network still has to find a new path, but there's more for the end host to do. The host has to somehow detect failure, and it has to resend a reservation request. It also has to free up the reservation along the old path somehow. What if the new reservation request is rejected?

This failure mode scales poorly. If a single router goes down, but millions of flows were using that router, then millions of reservation requests have to be simultaneously re-established.

We won't solve these problems in detail, but hopefully you're getting a sense that handling failures in circuit switching is a pretty hard problem.

4. How complex is it to implement each approach at scale?

If you actually tried to design circuit switching, a lot of additional design questions start to make the protocol really complicated, really quickly.

How do the routers know that the reservation was successful? When 2 sees the request and agrees, how does it know that 3 and 4 also agreed? (Possible approach: We send a confirmation back in the other direction, indicating that the reservation is confirmed.)

What if the reservation request is lost along the way? 1 and 2 agree, but the request packet is dropped before it reaches 3 and 4. (Possible approach: Set a timer, and if the reservation isn't confirmed in time, delete the reservation. Now the end host has to try again.)

What if the request is sent and everybody agrees, but the confirmation on the way back is dropped? 4 and 3 see the confirmation, but the confirmation packet is dropped before it reaches 2 and 1.

What if the reservation is declined? Should the end host try again and request less? Should the end host wait a bit and try again with the same request? Should the router say in the rejection, "I can't do 10 Mbps, but I can give you 8 Mbps?"

We won't solve every design problem, but hopefully you're noticing that circuit switching is harder to implement than it first seemed.

The fundamental problem that makes circuit switching complicated is state consensus. All the routers have to keep track of extra state, and they all have to agree on what that state is.

You might have heard of the Paxos protocols, which are extremely complicated protocols for getting multiple processors to agree on state. In practice, people run these algorithms on a group of 4-5 servers. With circuit switching, we're basically asking the Internet to run that on Internet scale, with millions of routers and flows.

In summary: Circuit switching gives the application better performance with reserved bandwidth. It also gives the developer more predictable behavior.

However, packet switching gives us more efficient sharing of bandwidth, and avoids startup time. It also gives us easier recovery from failure, and is generally simpler to implement (less for routers to think about).

Circuit Switching vs. Packet Switching In Practice

In the modern Internet, packet switching is the default approach.

There are limited cases where circuit switching is used. For example, RSVP (Resource Reservation Protocol) can be used within a small local network, to allow routers (not end hosts) to reserve bandwidth between themselves.

Another use of circuit switching in the modern Internet is dedicated circuits (e.g. MPLS circuits, leased lines). As a company, you can specifically buy some Internet bandwidth (possibly including physical infrastructure) dedicated to your business. This is very expensive compared to a standard Internet connection.

Dedicated circuits are deployed at less ambitious scales than hypothetical full-Internet circuit switching. Someone usually manually sets up the reservation. The reservation is long-lived (e.g. years). The reservation is at the granularity of companies, not individual flows.

Brief history: When the Internet was first designed in the 1970s-1980s as a smaller-scale, government-funded research project, it was packet switched.

In the 1990s, when the government stopped funding the Internet and control passed over to commercial enterprises, research and industry thought we would need to change to circuit switching. The designers predicted that voice and live TV would be the main heavy-duty uses of the Internet. Both of these applications have smooth bandwidth demand, well-suited for circuit switching. Also, because ISPs had to make money off the new commercialized Internet, they thought that circuit switching would offer a more intuitive business model.

There was a lot of work in research and standards bodies to implement circuit switching, but ultimately, this was a failed vision, for many of the reasons we discussed. Also, the main applications driving Internet growth were email and the web, not voice calls and TV, which is another reason why circuit switching vision didn't work out.

An interesting consequence of these design choices is, users and developers adapted to the realities of packet switching. If you watch a video and the connection is poor, you're used to the application adapting and the video quality decreasing. (Contrast with broadcast TV, which wouldn't do this.) This is a lesson in how technology can transform user behavior!

Links

Properties of Links

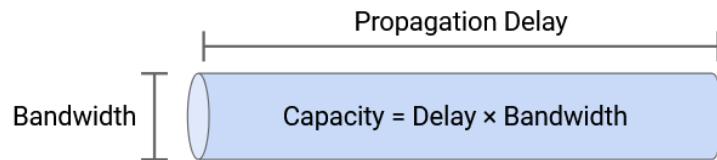
Now that we have a picture of how the layers of the Internet are built, let's focus on how a packet is sent across a link.

There are three properties we can use to measure the performance of a link.

The **bandwidth** of a link tells us how many bits we can send on the link per unit time. Intuitively, this is the speed of the link. If you think of a link as a pipe carrying water, the bandwidth is the width of the pipe. A wider pipe lets us feed more water into the pipe per second. We usually measure bandwidth in bits per second (e.g. 5 Gbps = 5 billion bits per second).

The **propagation delay** of a link tells us how long it takes for a bit to travel along the link. In the pipe analogy, this is the length of the link. A shorter pipe means that water spends less time in the pipe before arriving at the other end. Propagation delay is measured in time (e.g. nanoseconds, milliseconds).

If we multiply the bandwidth and the propagation delay, we get the **bandwidth-delay product (BDP)**. Intuitively, this is the capacity of the link, or the number of bits that exist on the link at any given instant. In the pipe analogy, if we fill up the pipe and freeze time, the capacity of the pipe is how much water is in the pipe in that instant.



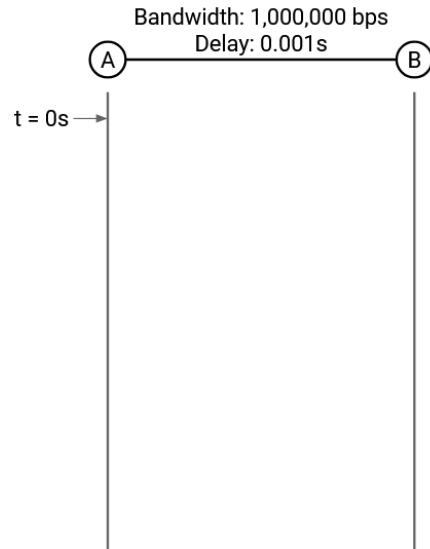
Note: You might sometimes see the term **latency**. In the context of a link, the latency is its propagation delay, though this word can also be used in other contexts (e.g. the latency from end host to end host, across multiple links). Latency by itself is not formally defined, and is context-dependent.

Timing Diagram

Suppose we have a link with bandwidth 1 Mbps = 1 million bits per second, and propagation delay of 1 ms = 0.001 seconds.

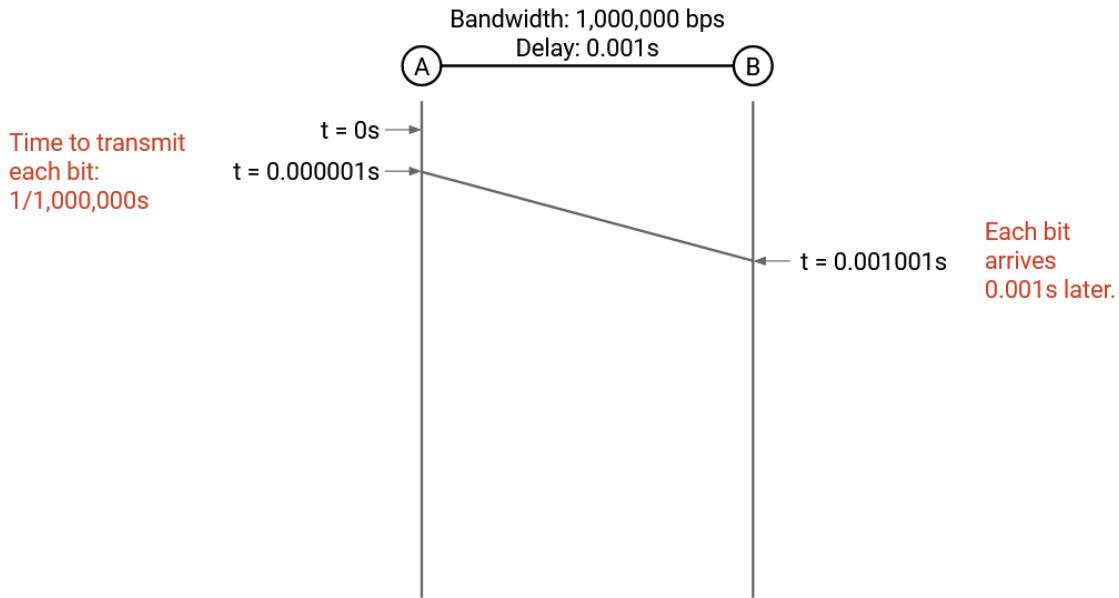
We want to send a 100 byte = 800 bit packet along this link. How long does it take to send this packet, from the time the first bit is sent, to the time the last bit is received?

To answer this question, we can draw a timing diagram. The left bar is the sender, and the right bar is the recipient. Time starts at 0 and increases as we move down the diagram.



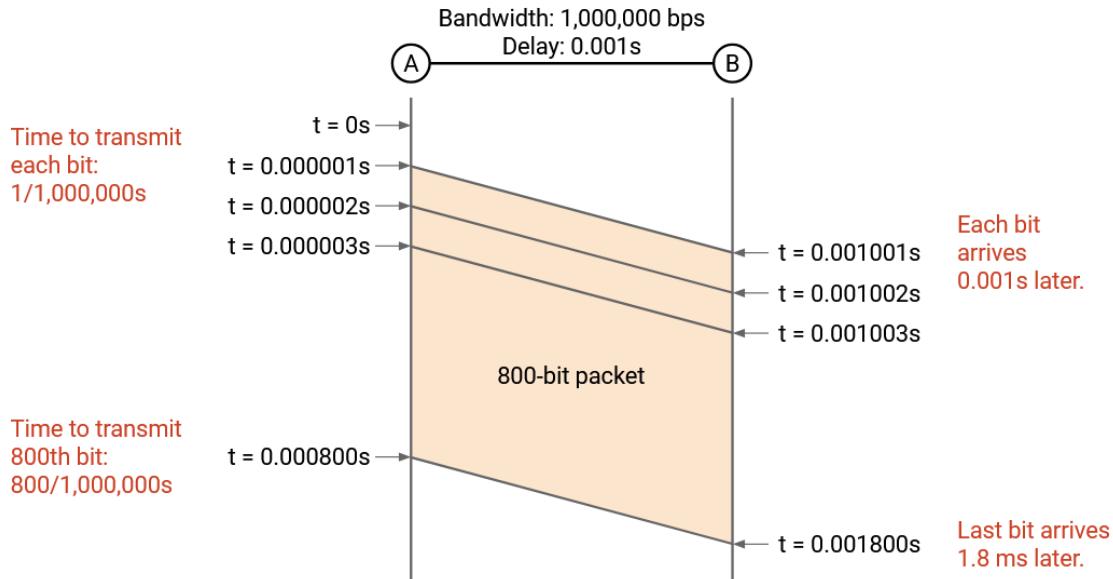
Let's focus on the first bit. We can put 1,000,000 bits on the link per second (bandwidth), so it takes $1/1,000,000 = 0.000001$ seconds to put a single bit on the link. At time 0.000001 seconds, the link has a single bit on it, at the sender end.

It then takes 0.001 seconds for this bit to travel across the link (propagation delay), so at time $0.000001 + 0.001$ seconds, the very first bit arrives at the recipient.



Now let's think about the last bit. From before, it takes 0.000001 to put a bit on the link. We have 800 bits to send, so the last bit is placed on the link at time $800 \cdot 0.000001 = 0.0008$ seconds.

It then takes 0.001 seconds for the last bit to travel across the link, so at time $0.0008 + 0.001$ seconds, the very last bit arrives at the recipient. This is the time when we can say the packet has arrived at the recipient.



Packet Delay

More generally, the **packet delay** is the time it takes for an entire packet to be sent, starting from the time the first bit is put on the wire, to the time the last bit is received at the other end. This delay is the sum of the transmission delay and the propagation delay.

The transmission delay tells us how long it takes to put the bits on the wire. In the example, this was $800 \cdot (1/1,000,000)$. In general, this is the packet size divided by the link bandwidth.

Since the transmission delay is a function of bandwidth, we can calculate packet delay in terms of the two link properties of bandwidth and propagation delay.

Bandwidth and Propagation Delay Tradeoffs

Consider two links:

Link 1 has bandwidth 10 Mbps and propagation delay 10 ms.

Link 2 has bandwidth 1 Mbps and propagation delay 1 ms.

Which link is better? It depends on the packets you're sending.

Suppose we wanted to send a single 10-byte packet. For both links, the time it takes to put one packet on the wire is negligible, and the propagation delay is the dominant source of delay. Link 2 has the shorter propagation delay, so it's the better choice.

Suppose we instead wanted to send a single 10,000-byte packet. Now, the transmission delay is the dominant source of delay, and we prefer Link 1, which allows us to put the bytes on the wire faster (higher bandwidth). You could validate this intuition with formal packet delay calculations: Link 1 takes roughly 18 ms to send this packet, while Link 2 takes roughly 81 ms.

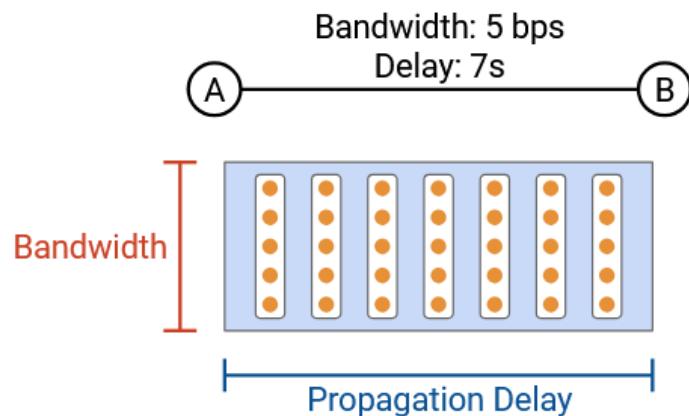
For a real-world example, consider a video call. If the video quality is poor, you probably have insufficient bandwidth (and shortening propagation delay won't help). By contrast, if there's a delay between the time you speak and the time the other person answers, the propagation delay is probably too long (and more bandwidth won't help).

Pipe Diagram

So far, we've been drawing timing diagrams to denote when network events happen (e.g. when the recipient gets the packet).

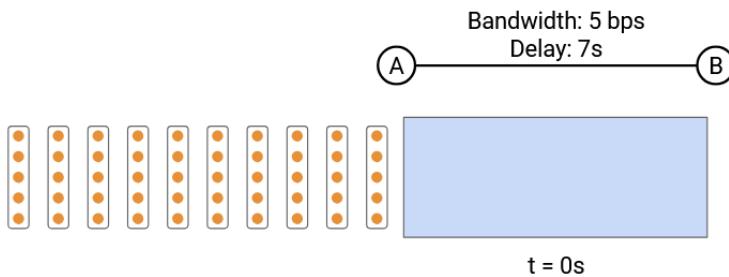
Another way to view packets being sent across the network is to draw the bits on the link at a frozen moment in time. Both views convey the same information, but depending on the context, one view might be more useful than the other.

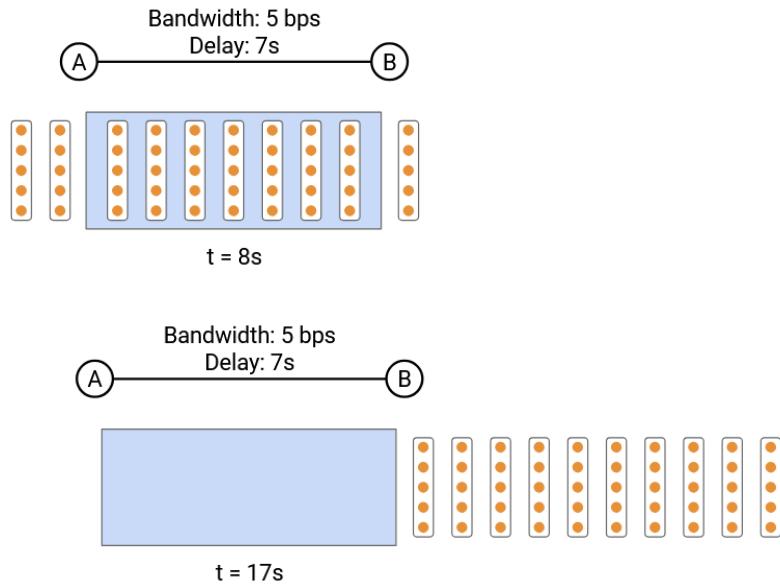
To draw the link, we can imagine the link is a pipe (similar to the water analogy) and draw the pipe as a rectangle, where the width is the propagation delay, and the height is the bandwidth. The area of the pipe is the capacity of the link.



Suppose we want to send a 50-byte packet across the link. In the pipe view, we can show a frozen moment in time with the packet being sent along the link.

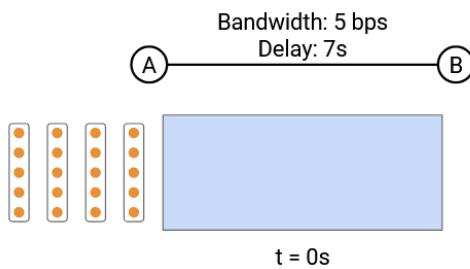
The packet is arranged in a rectangle, where the height of the rectangle tells us how many bytes were placed on the wire in a single time step. At every time step, the packet slides right in the pipe. Eventually, the packet starts to exit the pipe, and at each time step, one column of the rectangle exits the pipe.



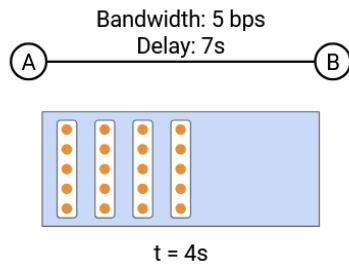


Non-obvious fact: The packet transmission delay in the timing diagram corresponds to the width of the rectangle.

To see why, suppose we have a link that can send 5 bits per second, and we have a 20-bit packet. In the timing diagram, there are 11 seconds between the time of the first and last bit being sent.

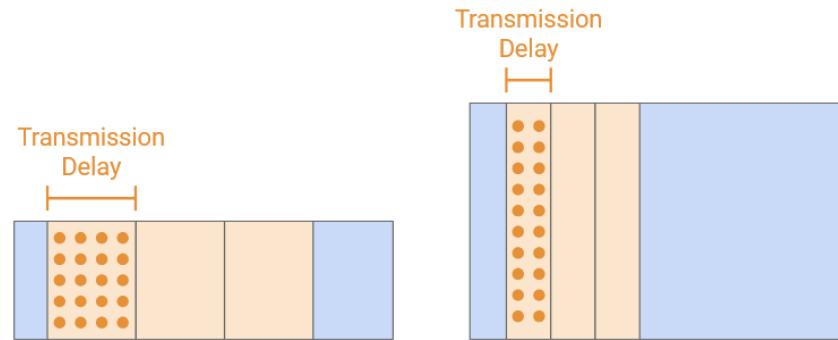


In the pipe diagram, every second, a column of 5 bits marches into the pipe. We need 4 columns to enter the pipe, which takes 4 seconds. This means the width of the packet in the pipe is 4 columns of packets = 4 seconds.



The pipe diagram lets us view the packet transmission time on the same axis as the propagation delay, and compare the two terms.

Pipe diagrams can be useful for comparing different links. Let's look at the exact same packets traveling through three different links.



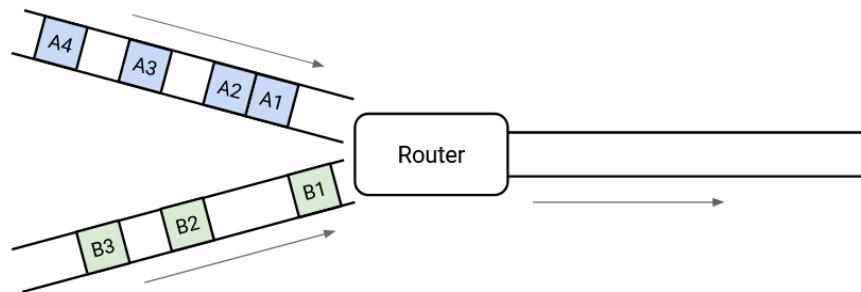
If we shorten the propagation delay, the pipe width gets shorter. The pipe height stays the same, and the shape of each rectangular packet is the same. (Remember, you can think of the packet height as the number of bits marching into the pipe at each time step, and the packet width as the time it takes to march all bits into the pipe.)

Other observations here: The packet width staying the same means the transmission delay didn't change. Also, the area of the link decreased, which tells us that the link has less capacity.

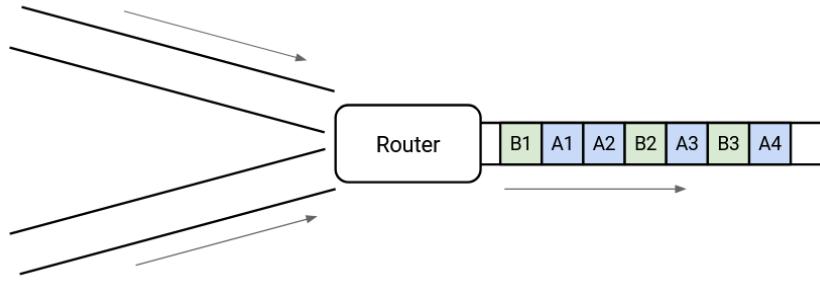
When we increase the bandwidth, the pipe height gets taller, indicating that we can march more bits into the pipe per unit time.

Notice that the shape of the packets also changed. The packets are now taller, because we can march more bits into the pipe per unit time. As a result, we finish feeding the packet into the pipe much faster, so the width of the packets (transmission delay) decreases.

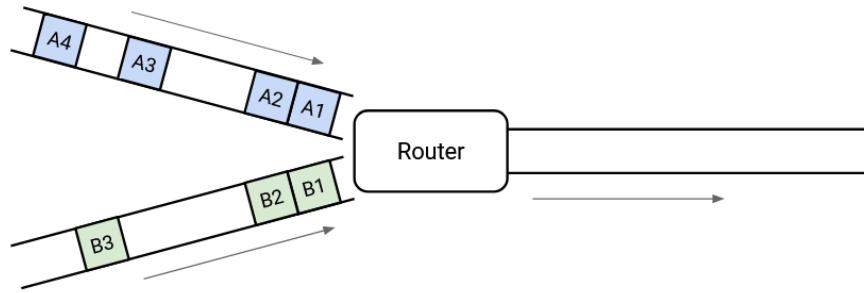
Overloaded Links



Consider this picture of packets arriving at a switch. The switch needs to forward all the packets along the outgoing link. In this case, there's no problem, because the switch has enough capacity to process every packet as it arrives.

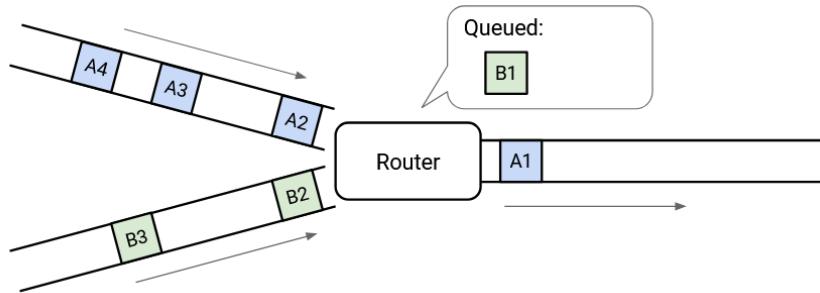


What about in this picture?

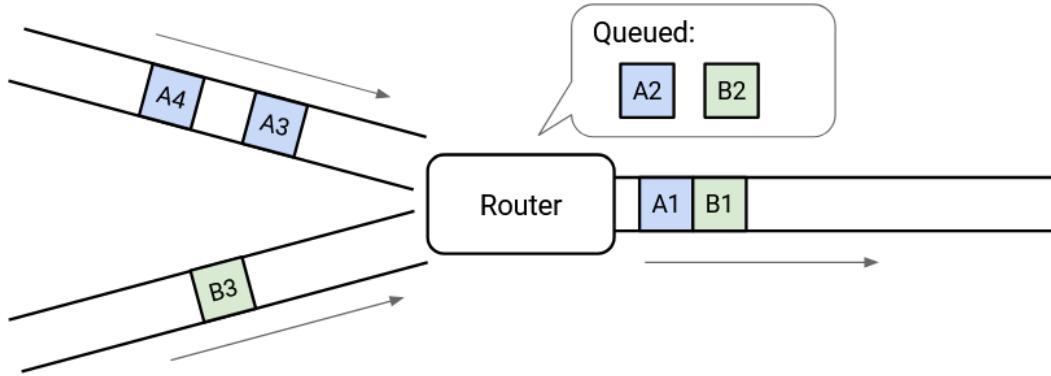


In the long term, we have enough capacity to send all the outgoing packets, but at this very instant in time, we have two packets arriving simultaneously, and we can only send out one. This is called **transient overload**, and it's extremely common at switches in the Internet.

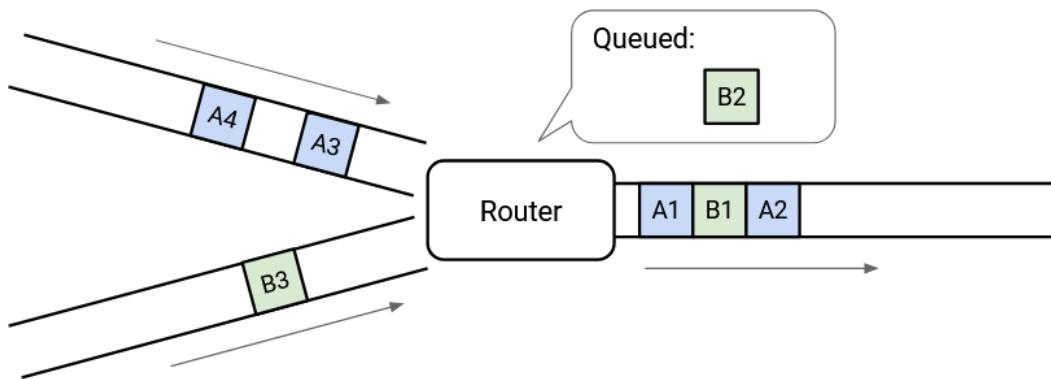
To cope with transient overload, the switch maintains a queue of packets. If two packets arrive simultaneously, the switch queues one of them and sends out the other one.



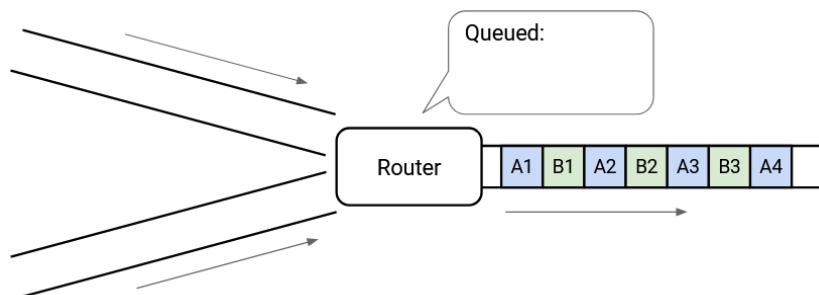
At any given time, the switch could choose to send a packet from one of the incoming links, or send a packet from the queue. This choice is determined by a **packet scheduling** algorithm, and there are lots of different designs that we'll look at.



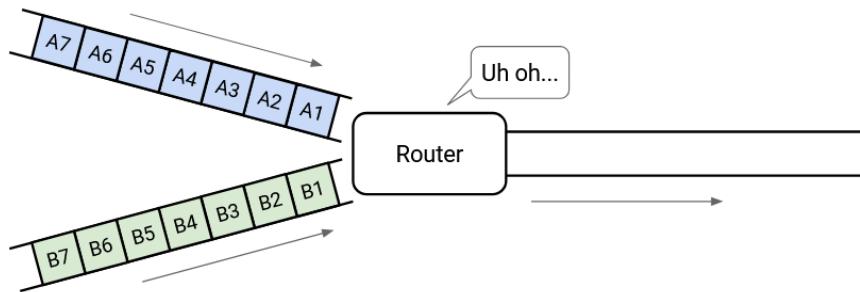
When there are no incoming packets, the switch can drain the queue and send out any queued packets.



This allows queues to help us absorb transient bursts.



What if the incoming links looked like this?



Now we have **persistent overload**. There just isn't enough capacity on the outgoing link to support the level of incoming traffic.

We could fill the queue up, but that still isn't enough to support the incoming load. One way or another, the switch will drop packets.

How do we account for persistent overload? Operators need to properly provision their links and switches. If they notice that a switch is frequently overloaded, they might decide to upgrade the link (which may require manual work).

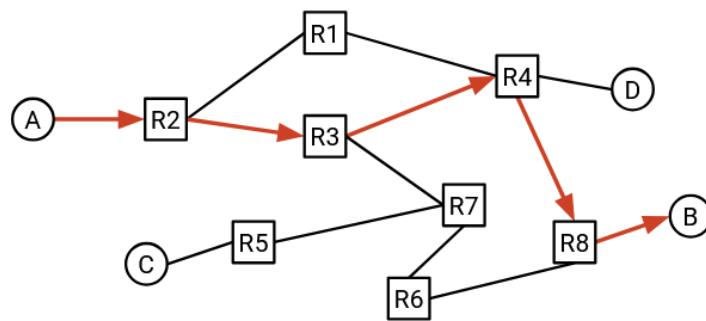
One possible solution to overload is to have the router tell the senders to slow down. (We'll study this later when we look at congestion control.) Ultimately, there's not much we can do to solve overload, though, which is why the Internet is designed to only offer best-effort service.

Now that we have a notion of queuing, we need to go back and update our packet delay formula. Now, packet delay is the sum of transmission delay, propagation delay, and queuing delay.

Introduction to Routing

What is Routing?

Suppose that machine A and machine B are both connected to the Internet. Machine A wants to send a message to machine B, but the two machines are not directly connected to each other. How does machine A know where to send the message, so that the message will eventually reach machine B? What path will the message take through the network to reach its destination of machine B? In this unit, we'll be studying **routing** to answer these questions.



First, we'll devise a model of the Internet so that we can pose routing as a well-defined problem. We'll also see what answers to the routing problem look like, and what makes an answer valid and good.

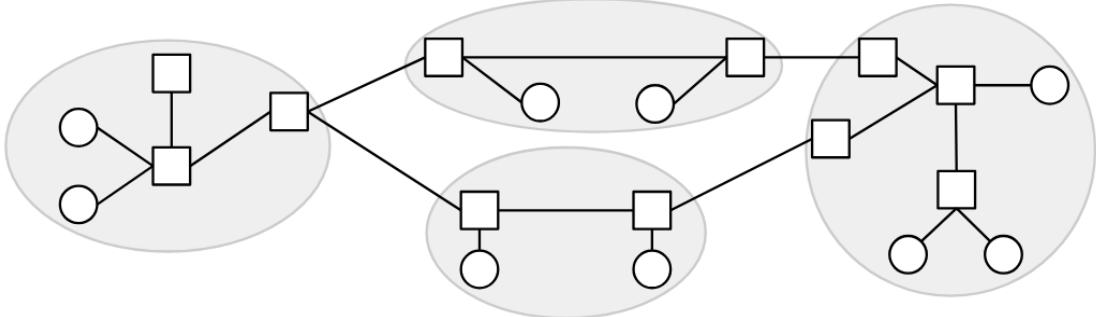
Next, we'll look at several different types of routing protocols that can be implemented to help generate answers to the routing problem. We'll also see how addressing protocols can be used to make our routing protocols scalable to the entire Internet.

Finally, we'll take a brief look at the real-life hardware we use to implement these routing protocols.

Inter-Domain and Intra-Domain Routing

One possible strategy for routing is to build a model of the Internet that includes every single machine in the world, and design a single giant routing protocol that will allow us to send packets anywhere in the world. However, this is infeasible in practice because of the scale of the Internet.

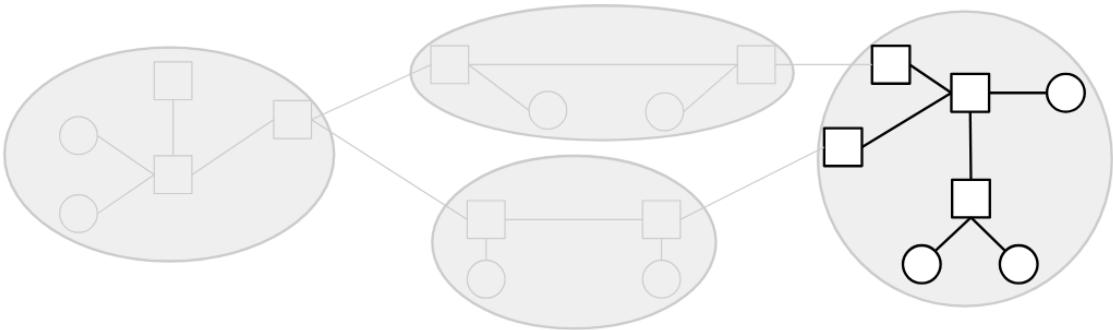
Instead, we'll take advantage of the fact that the Internet is a network of networks. In other words, the Internet consists of many local networks. Each local network implements its own routing protocol that specifies how to send packets within just that local network. Then, we can connect up all those local networks and implement a routing protocol across all the local networks, specifying how to send packets between different local networks.



Local networks are not identical. For example, they might differ in size: Some networks might have more machines than others. Or, the machines might be spread out over a wider physical area (e.g. the entire UC Berkeley campus), or a smaller area (e.g. your home). Networks can also differ in the bandwidth they need to support, the allowable failure rate, the number of support staff available, the age of the infrastructure, the amount of money available to build and support it, and so on.

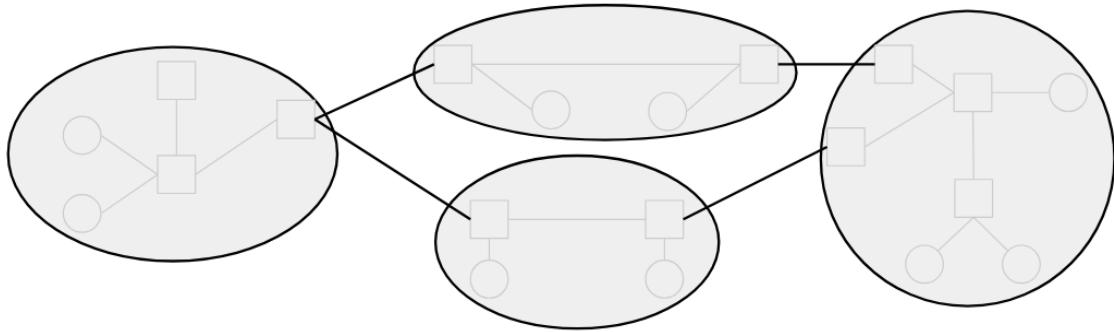
Because each network has its own structure and requirements, different local networks might choose to use different routing protocols. A strategy for routing packets might be effective on one network, but not another one.

With the network of networks model, we can let individual local networks choose a routing strategy for packets within their network. Each operator can choose the protocol that works best for them. The protocols for routing packets within a local network are called **intra-domain** routing protocols, or **interior gateway protocols (IGPs)**. Real-world examples include OSPF (Open Shortest Path First) and IS-IS (Intermediate System to Intermediate System).



By contrast, protocols for routing packets across different networks are called **inter-domain** routing protocols, or **exterior gateway protocols (EGPs)**. In order to support sending packets across different local networks, every network needs to agree to use the same protocol for routing packets between each other. If different networks used different inter-domain protocols, there's no guarantee that that the entire Internet could be connected in a consistent way. What if one operator only implemented Protocol X, and another operator only implemented Protocol Y? It's not clear how these two local networks would be able to exchange messages.

Because every network must agree to use the same inter-domain protocol, there is only one protocol implemented at scale on the Internet, namely BGP (Border Gateway Protocol).



This model of interior and exterior gateway protocols is convenient for intuition, but in practice, there is not always a clear distinction between them. For example, BGP is sometimes also used inside a local network, in addition to between different networks.

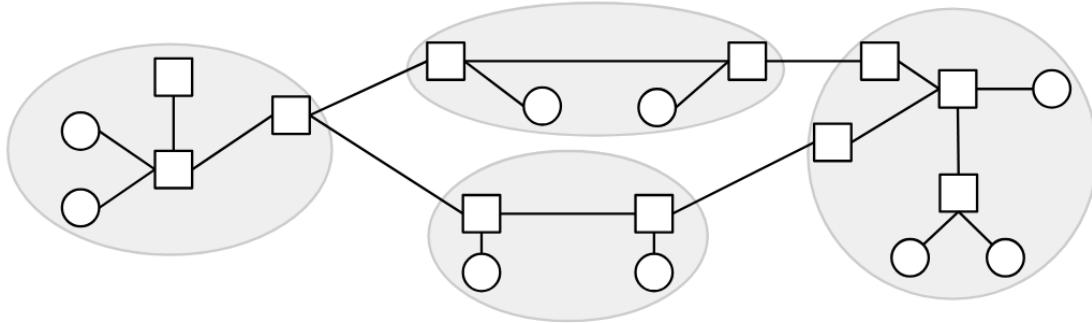
Regardless of whether a protocol is deployed internally within a network, or externally between all networks, we can additionally classify the routing protocol by looking at what the underlying algorithm is doing. In particular, we'll study distance-vector protocols, link-state protocols, and path-vector protocols (more about each type later).

Model for Intra-Domain Routing

Modeling the Network as a Graph

Let's create a simplified model of the Internet to help us formally define the routing problem.

Recall from the previous unit that we can think of the Internet as a set of machines, connected together with a set of links, where each link connects two of the machines on the network.



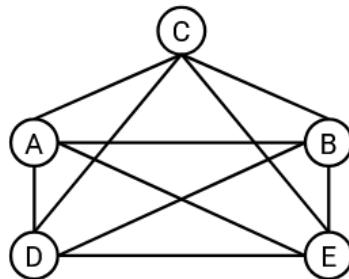
We can represent the network topology as a graph, where each node represents a machine, and each edge between two nodes represents a link between two machines.

Historically, sometimes links could connect more than two machines, but in modern networks, links essentially always connect exactly two machines.

Full Mesh Network Topology

Suppose we have two machines, A and B. If the two machines want to exchange messages, we could add a link between them.

But what if we had five machines instead of two? One possible approach is to create a link between every pair of machines, such that every machine is connected to every other machine. This is sometimes called a full mesh topology.



What are some drawbacks of this approach?

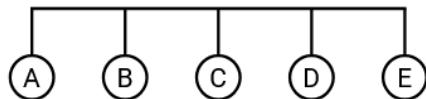
This approach doesn't scale well. If we tried to scale this to the size of the modern Internet, we'd need a wire connecting every pair of computers in the world. When a new computer joins the network, we'd have to create new links between that computer and every other computer in the world.

Although it can't scale to the entire Internet, there are still some benefits to a full mesh topology in smaller settings. In particular, having links between every pair of machines gives us a lot of bandwidth on the network. Every machine has a dedicated link to all other machines, and each pair of machines can use the full bandwidth on their dedicated link.

In general, there is no guarantee that each machine has a direct link to all other links. In other words, there is no guarantee that the underlying graph is fully-connected.

Single-Link Network Topology

In addition to the full mesh topology, there are other ways in which we can deploy links to connect up multiple machines. For example, we could use a single link to connect up all five machines:



(Here, we're temporarily breaking the assumption that a link connects only two machines, by considering a link that connects more than two machines.)

This approach would scale better than the full mesh topology. For example, if a new computer joined the network, instead of creating five new links between the new computer and the five existing computers, we can just extend the existing wire to the new computer.

However, this approach is more limited in the amount of bandwidth available to the machines. In particular, there is only a single link, and all five machines need to share the bandwidth on this link.

In order to create more sophisticated network topologies, we will need to introduce the idea of a router.

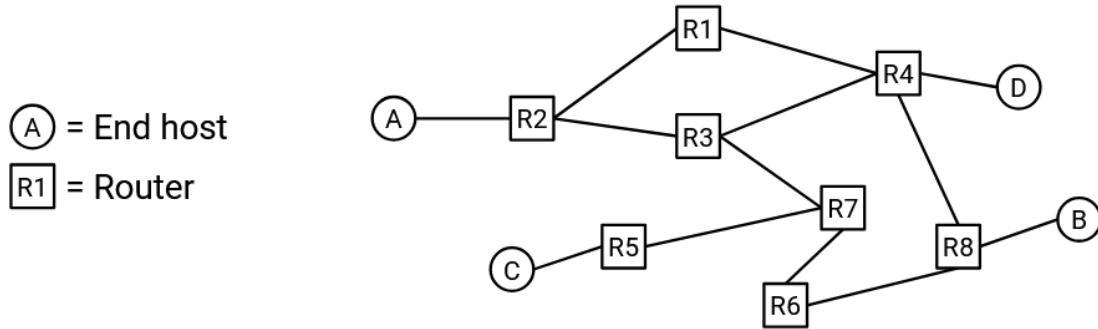
Routers and Hosts

In our simplified model, we'll classify every machine as being one of two types.

End hosts are machines connecting to the Internet to send and receive data. Examples of end hosts include applications on your own personal computer, such as your web browser. Web servers, such as a Google web server receiving Google search queries and sending back search results, are also end hosts. These machines send outgoing packets to other destinations, and could be the final destination for incoming packets. However, these machines usually do not receive and forward intermediate packets (i.e. packets with some different final destination).

Routers, by contrast, are machines connected to the Internet responsible for receiving and forwarding intermediate packets closer to their final destination. For example, consider the router installed in your home network, or routers living in a data center building somewhere. These machines usually do not create

and send new packets of their own, and they usually are not the final destination for packets. For example, in your daily Internet use, you might want to send packets to a Google web server to perform a search, but you probably don't need to send a message directly to your home router or a data center. Those routers will help you forward your packet toward Google, but they are not the final destination of your packet.



Depending on the network design, routers could be legal destinations, but in this unit, we'll ignore routers as destinations. However, do note that routers potentially can be sources and send new packets of their own.

Routers are sometimes also called switches. There are historical differences between routers and switches, but nowadays, the terms are used interchangeably. In these notes, we'll use "router" when possible.

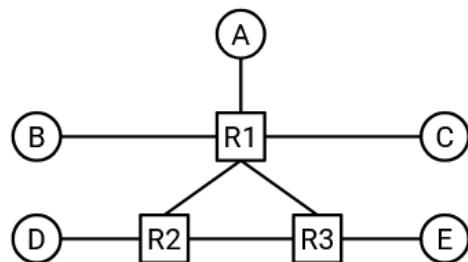
In our graph model of the Internet, routers appear as intermediate nodes that are usually connected to multiple neighbors. End hosts appear as nodes that are usually connected to one or more routers. In practice, these assumptions aren't always true.

In these notes, when possible, we'll always draw routers as squares and end hosts as circles. In practice, sometimes routers are represented by other symbols. For example, this is a common router symbol used in network diagrams:



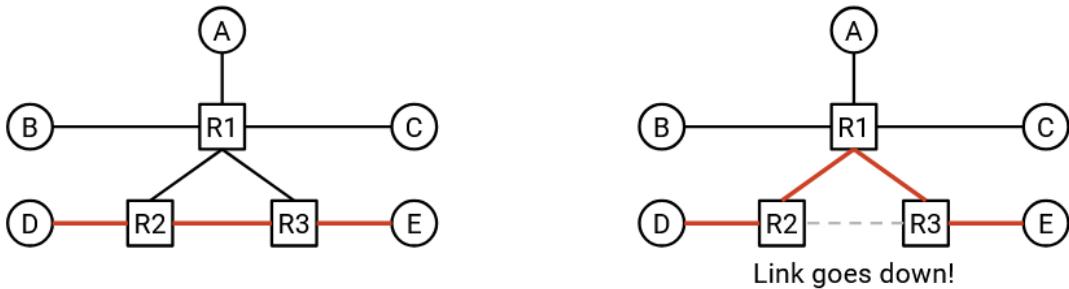
Network Topologies with Routers

Now that we have routers in addition to end hosts, we can create more complicated network topologies like this:



This topology lets us combine the benefits of the full-mesh and single-link topologies. In particular, this topology uses fewer links than the full mesh topology from earlier. Also, this topology has more bandwidth than the single link topology from earlier.

This topology is also more robust to failure. If a link goes down, the packet can take a different path through the network and still reach its destination.



End Hosts in Routing

Note that end hosts generally do not participate in routing protocols, since they don't forward intermediate packets. Instead, end hosts are often connected to a single router with a single link. By default, the end host sends all outgoing messages to the router, which will figure out how to send the packet to its final destination. This strategy of sending everything to the router is sometimes called the **default route** of the end host.

When designing routing protocols, we often ignore end hosts, except as destinations (since the routers need to figure out how to reach different destinations).

Packets

Recall from the previous unit that when an application wants to send data over the Internet, the application creates a packet containing the data. As the packet is passed to lower-layer protocols, additional headers are wrapped around the packet with metadata to help the packet reach its destination.

In the routing unit, we'll consider a simplified model where each packet has a header with metadata, and a payload with the application-level data. We'll ignore nested headers and multiple layers for now.

Routing protocols are not concerned with the application-level data. It doesn't matter whether the user is trying to send an image, or an HTML webpage, or an audio file; from the perspective of routing, we have a sequence of 1s and 0s, and we need a protocol to send those bits to their destination.

In the header, the main metadata field we're concerned with is the destination address. This tells us the final destination of the packet. When a router receives a packet, the router reads the metadata field in the header to determine how to send the packet towards its final destination. The problem of figuring out where to send the packet is the key problem we'll need to solve in routing.



Addressing

How do we write down the destination of the packet in the packet header? We'll need some way of addressing each machine on the network. In other words, we need a protocol that assigns an address to each machine on the network.

Later in this unit, we'll discuss scalable approaches to addressing. For now, let's assign each machine a unique label (e.g. we could label three routers X, Y, and Z), and treat those labels as the addresses for each router. This will allow us to think about the routing problem and the addressing problem separately.

At this point, we can define the routing problem: When a router receives a packet, how does the router know where to forward the packet such that it will eventually arrive at the final destination?

Network Topologies Change

At this point, we have defined the routing problem, but there are still a few more practical considerations that make the routing problem difficult.

If the Internet could be drawn as a fixed, constant graph that never changes, then perhaps we could solve the routing problem by simply looking at the graph and computing paths through the graph.

However, the network topology is constantly changing. For example, links might fail at unpredictable times. Now, packets must be sent along a different route in order to reach the destination.

New links might also be added, creating additional paths that can be considered during routing.

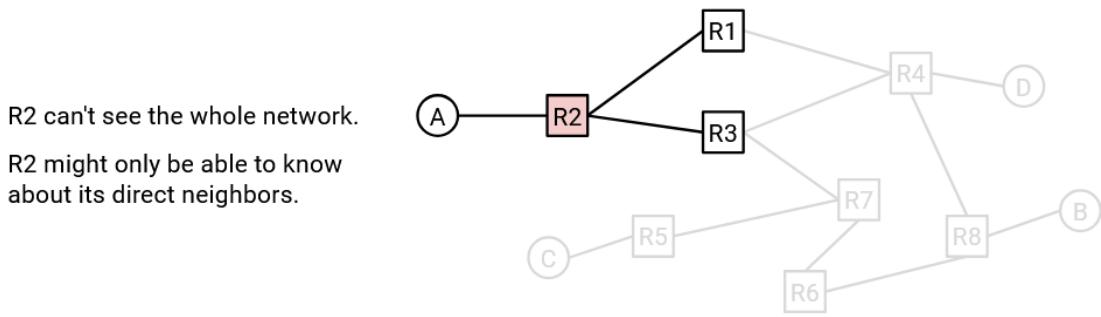
The routing protocols we design need to be robust to these changing network topologies.

Routing Protocols are Distributed

If the network changes, perhaps we could solve the routing problem by updating our graph and then computing paths through the new graph.

Another problem that makes routing difficult is that routers don't inherently have a global, birds-eye view of the entire network. For example, if a link somewhere else in the network fails, there's no way for all

routers to automatically know this. We will have to somehow propagate that information about the new network topology to the routers as part of our routing protocol.



This leads to routing protocols often being distributed protocols. Instead of a single central mastermind computing all the answers, each router must compute its own part of the answer (possibly without full knowledge of the network topology). Collectively, the answers computed by each router must form a global answer to the routing problem that allows packets to reach their end destination.

The distributed nature of routing protocols also means that we have to account for individual routers failing. If there was a single computer that was solving the problem, and that computer crashed and forgot the answer, we could simply make the computer re-compute the entire answer from scratch. However, in a distributed protocol, if one router crashes and forgets its part of the answer, our protocol will need to find a way to help this one router recover from failure and re-learn its part of the answer.

Links are Best-Effort

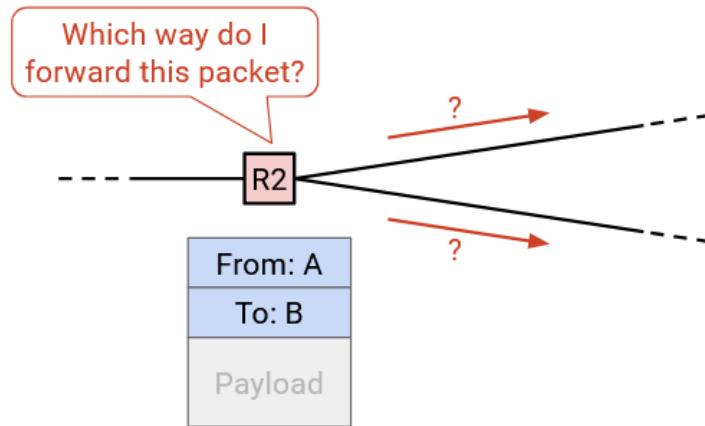
Recall from the previous unit that protocols at Layer 3 and below are best-effort. In other words, when a packet is sent over a link, there is no guarantee that the packet reaches the destination. The link might drop the packet.

When designing routing protocols, we'll need to account for this problem as well.

Routing States

Bad Routing Strategies

So far, we've defined the routing problem as this: When a router receives a packet, how does the router know where to forward the packet such that it will eventually arrive at the final destination?



Once we find an algorithm (a routing protocol) to solve this problem, we can apply that algorithm to generate an answer, which we'll call a **routing state**. You can think of a routing state as a set of rules that each router uses to forward packets it receives. What does a routing state look like, and how can we check if a given routing state is valid or good?

To start, we could consider some bad strategies for generating routing states. One possible routing strategy is: The router forwards the packet to a randomly-selected neighbor. Intuitively, we can already see that routing states generated this way probably won't be valid. If we use this strategy, we can't be sure that packets will reach their final destination.

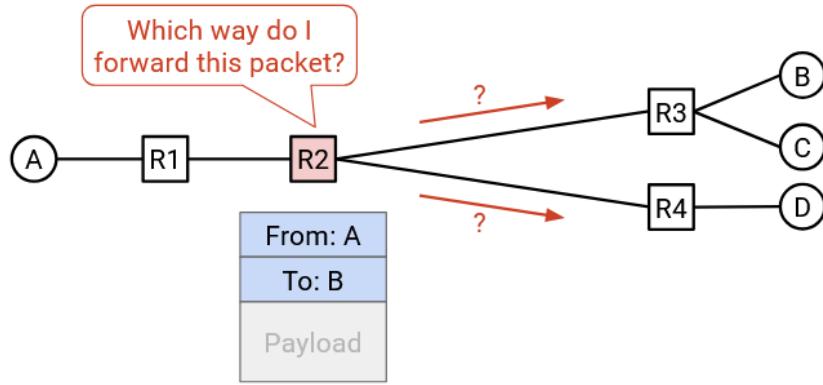
Another possible bad strategy is: The router forwards a copy of the packet to every single one of its neighbors. Intuitively, this might be valid, in the sense that copies of the packet will eventually spread across the entire network and probably reach the destination. However, this strategy is inefficient, because it wastes a lot of bandwidth forwarding the packet to routers that were not needed to send the packet to its final destination.

We can intuitively see that these two strategies are bad, but to analyze smarter routing protocols, we'll need to formally define what a routing state looks like. Then, we'll need to formalize what makes a routing state valid, and what makes a routing state good.

Forwarding Tables

In our model of the network, each router has some number of outgoing links connecting it to adjacent routers and hosts. In other words, in the underlying graph, each router node has some number of neighbors, connected to the router by an edge.

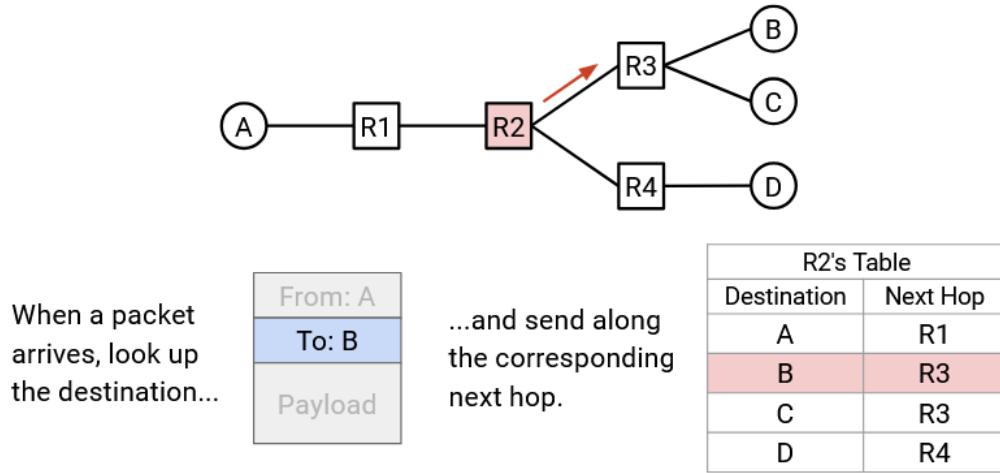
When the router receives a packet, with its final destination in the metadata, the router needs to decide which of the adjacent routers or hosts the packet should be forwarded to. The next intermediate router that the packet will be forwarded to is called the **next hop**.



For example, consider this network. If R2 receives a packet whose final destination is B, the natural corresponding next hop would be R3. The possible choices of next hop are R1, R3, and R4 (the three routers adjacent to R2), and R3 is the next hop that sends the packet closer to B.

If R2 instead receives a packet whose final destination is A, then the natural corresponding next hop would be R1 instead.

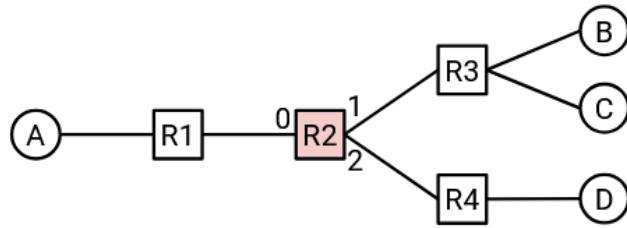
For each possible final destination, we can write down the corresponding next hop to forward the packet closer to that destination. The result is called a **forwarding table**.



Note that in the mapping of destination to next hop, a next hop can be used more than once. For example, in R2's forwarding table, packets destined for B and packets destined for C will both be forwarded to R3.

By writing down the forwarding table for each intermediate router, we now have a full routing state for the network. In other words, given a packet with some final destination, we know exactly how each router will forward that packet.

In the physical world, instead of mapping destinations to next hops, routers will often map destinations to **physical ports**, where each physical port corresponds to a link. In the graph model, we would now be mapping each destination to an edge, instead of mapping each destination to a neighboring node. In the physical world, you can think of this as a router having several outgoing wires, where each wire is connected to another router. Instead of writing down neighboring routers in the forwarding table, the router instead writes down which wire a packet should be sent along.



R2's Table (Conceptual)	
Destination	Next Hop
A	R1
B	R3
C	R3
D	R4

R2's Table (Reality)	
Destination	Port
A	0
B	1
C	1
D	2

This is a subtle distinction, and it reflects the fact that the router doesn't really care about the identity of the neighboring router. The only decision the router needs to make is to send the packet along one of the wires, regardless of who the wire is connected to. In these notes, we'll draw forwarding tables as mapping destinations to next hops (instead of physical ports), for simplicity.

Destination-Based Forwarding

A consequence of using a forwarding table is that given a packet, the decision of where to forward the packet depends only on the destination field of the packet. In other words, if a router receives many different packets, all with the same destination, they will all be routed to the same next hop (assuming the forwarding table stays unchanged). Since each destination is only mapped to a single next hop, there's no way for two packets with the same destination to be forwarded to different routers. This approach is called **destination-based forwarding** or **destination-based routing**.

Destination-based routing is the most common approach to routing, and it's what's used in the modern Internet. In theory, other approaches could exist where additional metadata is used to make forwarding decision, but these are usually only used in limited applications (e.g. inside a particular local network).

In later units, when we consider data center topologies, we might consider destination-based forwarding approaches where there might be more than one next hop for a specific destination. In this unit, we'll assume that each destination is mapped to only one next hop.

Routing vs. Forwarding

Now that we've introduced the idea of a forwarding table, we need to make a distinction between the process of creating the forwarding table, and the process of using the forwarding table.

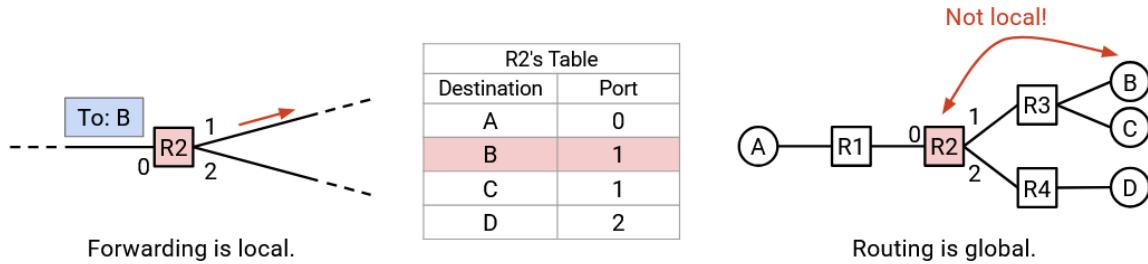
Routing is the process of routers communicating with each other to determine how to populate their forwarding tables.

Forwarding is the process of receiving a packet, looking up its appropriate next hop in the table, and sending the packet to the appropriate neighbor.

Forwarding is not the same as routing. When forwarding packets, routers use the existing forwarding table, with no knowledge of how that table was generated.

Forwarding is a local process. When a router is forwarding packets, the router doesn't need to know the full network topology. The router also doesn't care about where the packet goes after it's been forwarded to the next hop. The router only needs to know about the arriving packet, and its own forwarding table.

By contrast, routing is a global process. In order to fill out the forwarding tables, we will need to learn something about the global topology of the network.



For example, in when filling in R2's forwarding table, we had to somehow learn that destination B is associated with R3, even though host B is not directly connected to R2. During routing, each router will need to know about non-local destinations as well.

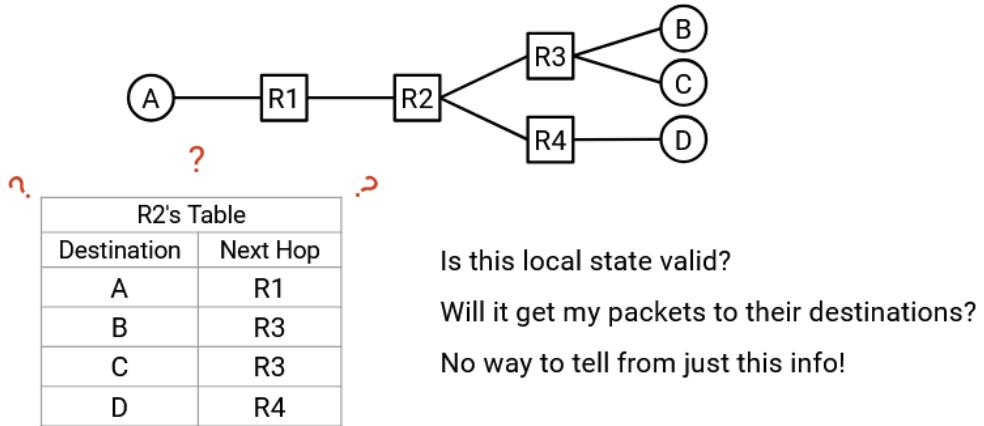
Routing State Validity is Global

Recall that a routing state consists of a forwarding table for each router, which collectively tells us how packets will travel through the network. Given a routing state, how can we tell if the routing state is correct or incorrect?

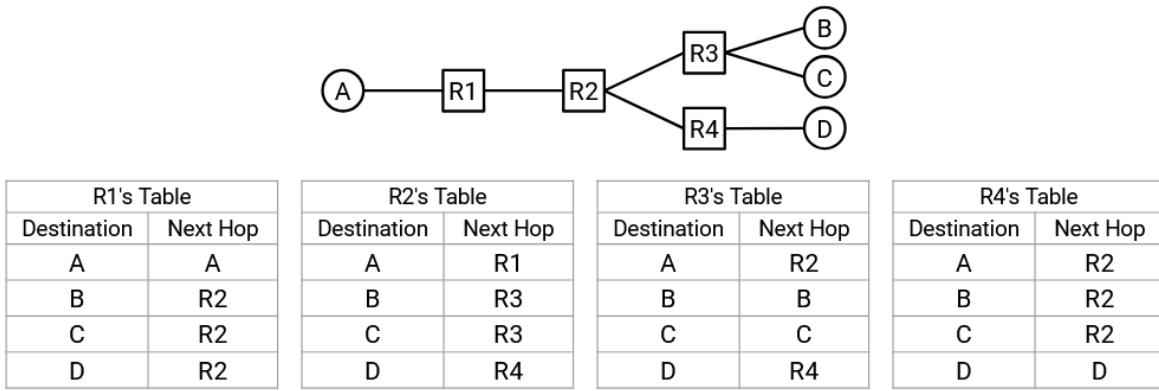
First, we need to formally define **routing state validity** to determine whether a routing state is valid (though this term may not be widely used outside CS 168 at UC Berkeley). The main requirement for validity is: the routing state needs to produce forwarding decisions that ensure that packets actually reach their destination.

Note that validity must be evaluated in the global context, not a local context. Looking at local routing state, such as a single router's forwarding table, cannot tell us whether a routing state is valid. For example, in a router R2's local forwarding table, we might see that the next hop for destination A is router R3, but

we have no way to decide if this is valid. Will forwarding packets to R3 help packets reach destination A? There's no way to tell from just the forwarding table.



Instead, we need to consider the global routing state, which consists of the collection of all the forwarding tables in all of the routers.



Routing State Validity Definition

Now, we can define a formal condition that we can use to check whether or not packets will reach their destination for a given routing state.

A global routing state is valid if and only if, for any destination, packets do not get stuck in dead ends or loops.

A **dead end** occurs if a packet arrives at a router, but the router doesn't know how to forward the packet to its destination, so the packet is not forwarded. This might occur if the router's forwarding table doesn't contain an entry for the packet's destination.

Note that the dead end condition only applies to the intermediate routers, and not the end hosts. When a packet reaches the destination end host, there's no need for the end host to forward the packet any further, so we won't consider end hosts in the dead end condition.



A **loop** occurs if a packet is sent in a cycle around the same set of nodes. Note that because we're using destination-based forwarding, where the next hop only depends on the destination, once a packet enters a loop, it will be trapped in the loop forever. When the packet arrives at the router the first time, or the 10th time, or the 500th time, it will be forwarded the exact same way (since the final destination is the same). Since this applies to every router on the loop, the packet will be stuck in the loop forever.



This condition (no dead ends, no loops) is both necessary and sufficient for a route to be valid. Let's check both directions of this logical implication.

No dead ends and no loops is a necessary condition for validity. In other words, a state is valid only if there are no dead ends and no loops.

Proof: If there's a dead end, the packet won't reach the destination. The packet will reach the dead end and not be forwarded.

If there are loops, the packet won't reach the destination. The packet will be trapped in the loop forever (because of destination-based forwarding, described earlier). Also, note that the final destination can't be part of the loop, since the destination won't forward the packet. Therefore, a packet trapped in the loop won't reach the destination.

Now, let's check the other direction. If there are no loops and no dead ends, then the state is valid.

Proof: Assume that the routing state has no loops or dead ends. A packet won't reach the same node twice (because there are no loops). Also, the packet won't stop before reaching the destination (because there are no dead ends). Therefore, the packet must keep wandering through the network, reaching different nodes. There are only a finite number of unique nodes to visit, so the packet must eventually reach the destination. Therefore, the routing state must be valid.

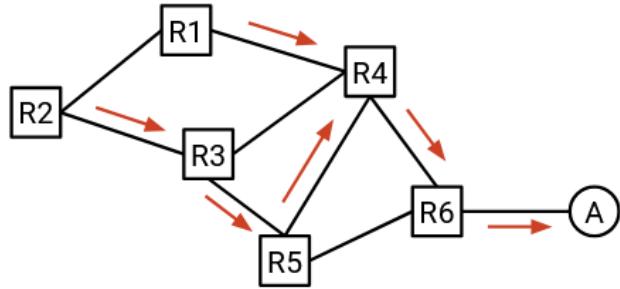
Directed Delivery Trees

Now that we have a formal definition for routing state validity, we can ask: given a global routing state, how can we check if it's valid?

To simplify the problem, let's start by considering only a single destination end host, ignoring all other end hosts. In each router, we can look up this destination to get the corresponding next hop, which tells us how each router will forward packets meant for this destination.

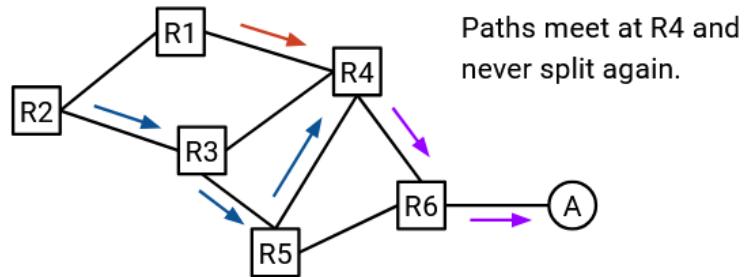
We can represent the next hop at each router (for this single destination) as an arrow, which shows us all the possible paths that this packet might take to reach the single destination.

R2's Table	
Destination	Next Hop
A	R3
...	...



In the resulting graph, each node will only have one outgoing arrow. This reflects our assumption that in each router's forwarding table, there is only one next hop corresponding to a destination.

Notice that in the resulting graph, once two paths meet, they never split. In other words, even if there are multiple incoming arrows (paths) to a node, since there is only one outgoing arrow, those paths will now converge into a single path. This reflects our destination-based forwarding approach, because each router only uses the final destination to decide how to forward a packet. The router does not care how the packet arrived at the router in the first place.



The arrows we've drawn form a set of paths that a packet can take to reach the single destination. This set of paths is called a **directed delivery tree**.

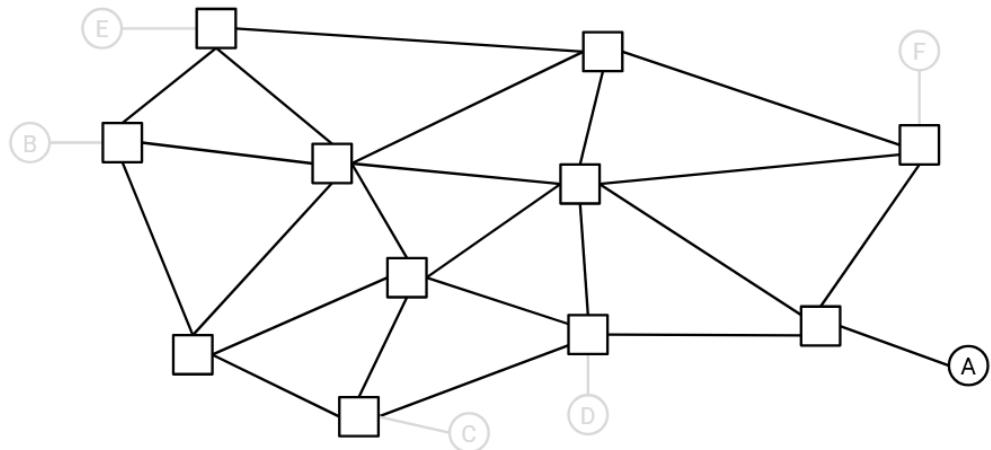
In graph terms, the arrows in a valid delivery tree must form an **oriented spanning tree**, rooted at the destination. Recall that a spanning tree is a set of edges in the graph that touch every node and form a tree. We want the delivery tree to be a tree, since there should be no cycles (packets can't travel in loops). We want the delivery tree to be spanning (touch every node), because we want to be able to reach the destination from everywhere. The delivery tree is oriented because the edges have arrows, which tells us which direction to forward the packet.

All edges in a valid delivery tree should point toward the destination. In other words, starting from any node, following the arrows should always result in reaching the destination.

Verifying Routing State Validity

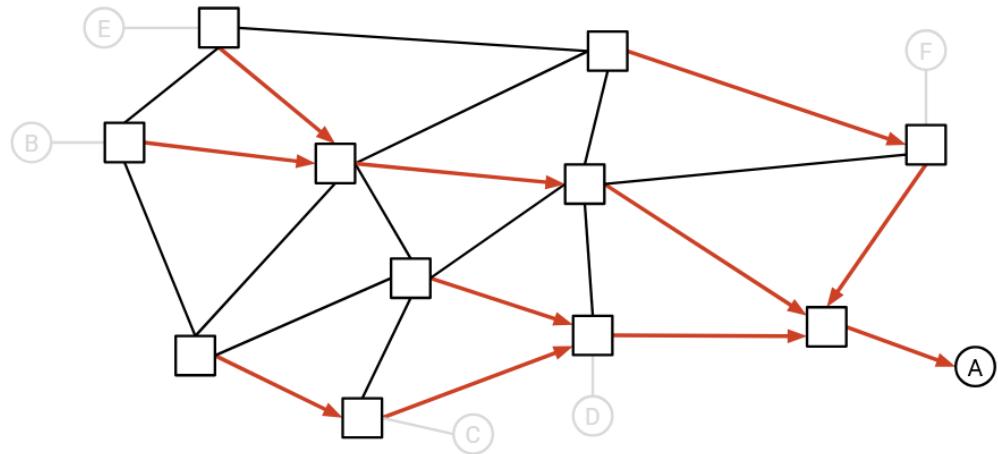
As before, let's consider only a single destination end host, ignoring all other end hosts.

Example: Even though there are multiple end hosts here, let's only consider end host A.



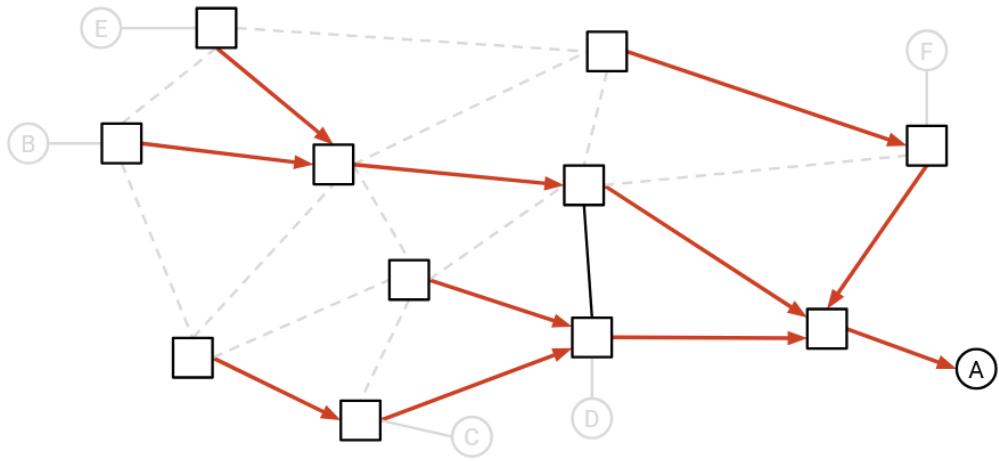
Using the forwarding tables at each router, we will draw the arrows into the network to form the directed delivery tree for this single destination. Formally, for each router (node in the graph), we will draw a single outgoing arrow from that node.

Example: Using the forwarding tables (not shown), we can draw one outgoing arrow per router.



For simplicity, at this point we can delete all the links without arrows on them. These links without arrows will never be used to send packets to the single destination, since they are not on the delivery tree.

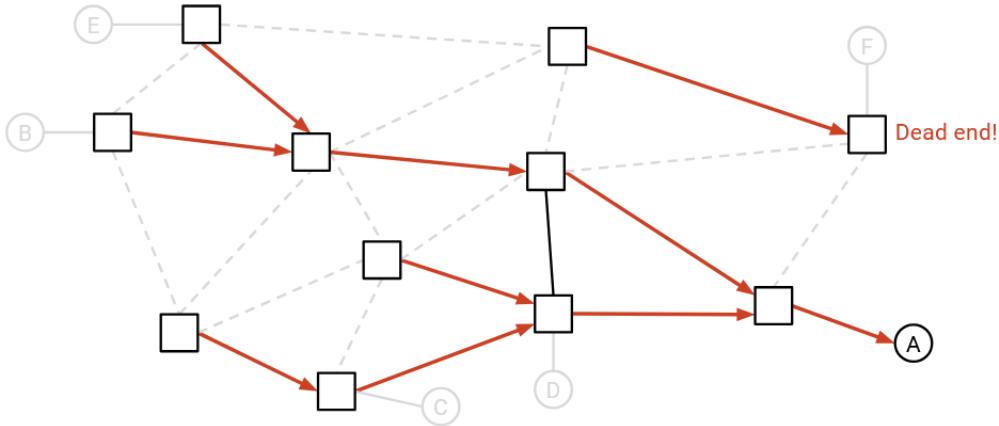
Example: We can delete all the links without arrows.



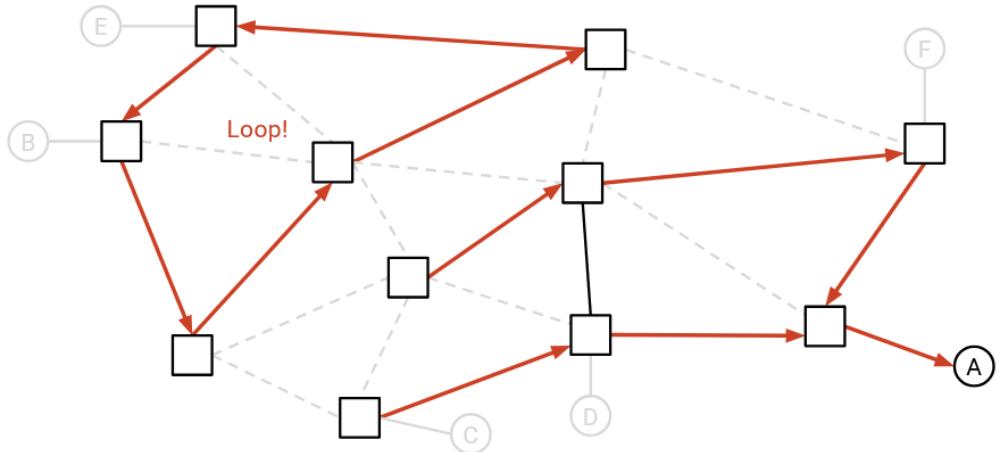
If the remaining graph is a valid directed delivery tree (spanning tree, all arrows pointing toward destination), then we can say that the routing state is valid for this single destination.

In the above example, the residual graph is indeed a valid spanning tree converging at A, so we can say this routing state is valid for A.

Here are some examples of invalid routing states:



This state is invalid. Intuitively, there is a dead end router. A packet bound for A could get sent to this router, and this router would discard the packet without forwarding it. Formally, the remaining graph is not a spanning tree, because the edges are not all connected (there are two disconnected components, which is not allowed in a tree).



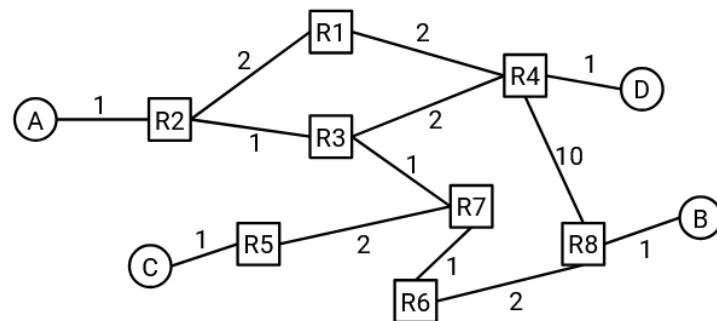
This state is also invalid. Intuitively, there is a loop that the packet could get stuck in. Formally, the remaining graph is not a spanning tree, because the edges are disconnected, and there is a cycle.

We can repeat this process, once for every different end host (isolating a different end host each time). If the routing state is valid for all destinations, then we can say that the routing state is valid, and will always deliver packets to their correct destinations.

Least-Cost Routing

Now that we have a definition of what makes a routing state valid (routes have no loops and dead ends), we can additionally define what makes a routing state good. It's possible that a network has multiple valid routing states, and we want some metric that can help us determine whether one route is better than another.

Least-cost routing is a common approach for measure whether a route is good. In least-cost routing, we assign a numeric cost to every link, and look for routes that minimize the cost. In other words, we want routes that result in packets traveling along the lowest-cost paths to their destinations.



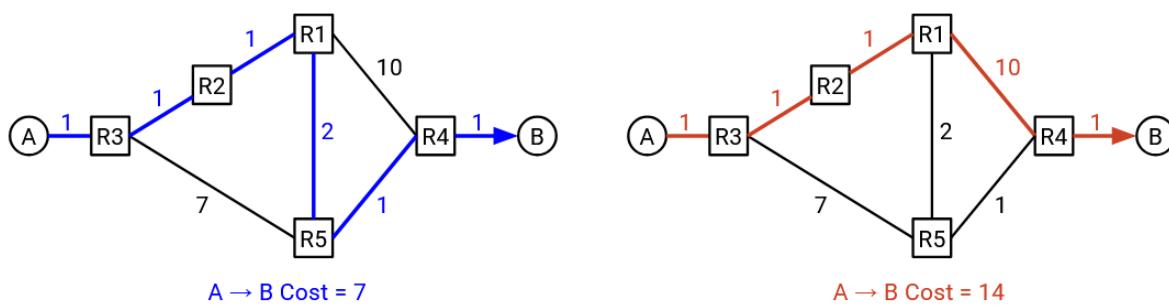
There are many different costs we could consider assigning to links. The cost could depend on the price of building the link, the propagation delay, the physical distance of the link, the unreliability, the bandwidth, among other factors. For example, we could assign costs based on the quality of the link (bandwidth and propagation delay), such that the lowest-cost path prefers higher-quality links.

By allowing operators to set link costs arbitrarily, we give the operator the ability to optimize the network for their specific needs. The costs we assign depend on the operator's goals for the network. If we had a 400 Gbps link with 20 ms propagation delay, and a 10 Gbps link with 5 ms propagation delay, which one is lower-cost? It depends on if we're optimizing for bandwidth, propagation delay, some combination, or something else entirely.

If we assign a cost of 1 to every link, then the least-cost path is the path that travels along the fewest links. We sometimes call this minimizing the **hop count**. In these notes, if the edges of a graph are not labeled with a cost, you can assume all the edges have cost 1,

The operator of a network can decide how to assign costs to each link. The operator might manually assign costs. Or, the operator could have the network automatically configure the costs, although this may not work with some metrics that can't be automatically measured (e.g. the network has no idea about the financial cost to build the link).

When designing a routing protocol, we can abstract away how the costs were assigned. From the routing protocol's perspective, somebody else (e.g. the network operator) has already assigned the costs, based on something that they consider important. The algorithm takes in the costs as an input, and computes the least-cost paths, regardless of what the costs actually represent.



Note that costs are local to each router. A router knows about the cost of its own outgoing links, but there is no way for the router to automatically know the costs of all links. This is consistent with the constraint we mentioned earlier, where routers don't have a global view of the entire network's topology.

For simplicity, routing protocols make some assumptions about how the costs are defined.

We'll assume that costs are always positive integers. This is consistent with many common real-life metrics, such as length of a link or monetary cost of a link. If we're trying to minimize the total physical distance traveled by a packet, a negative link cost doesn't make sense. You can't travel along a link and decrease the total distance traveled. This assumption will help simplify our protocols later, since we won't have to worry about edge cases like negative-weight loops (where the least-cost solution would be to travel around the loop forever).

We'll assume that costs are symmetrical. The cost from A to B is the same as the cost from B to A. This reflects the diagrams we'll draw, where an edge is labeled with a single symmetric cost. In theory, it's possible to have asymmetric link costs, but this is not done in practice, and would lead to more complicated routing protocols.

With these assumptions, our definition of good routes (least-cost) is consistent with our definition of valid routes. In particular, a least-cost route won't have any loops, because costs are positive (traversing the loop

would only increase the cost).

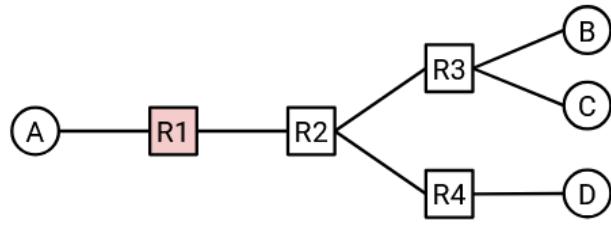
Static Routing

One possible way to generate routes is to have the network operator manually populate the forwarding table. This is known as **static routing**.

Static routing by itself isn't practical (e.g. not scalable, prone to human error), but even with a routing protocol implemented, some routes still need to be manually created by operators. You can think of these manual routes as the "trivial" or "base case" routes, from which the routing protocol generates more complex routes.

If we're directly connected to another machine that we want to route packets to, we can manually configure a route to forward packets to that other machine. These routes are called **direct routes** or **connected routes**. For example, your home router is connected to your personal computer with a link, so your home router can add an entry in the forwarding table corresponding to your computer. This entry is added by telling the router about the connection, and is not added from running any routing protocol.

R1's Table	
Destination	Next Hop
A	Direct
B	R2
C	R2
D	R2



It is also possible to use static routing to hard-code entries for destinations in the forwarding table, even if we aren't directly connected to that destination. This can be useful if there's a route that never changes, and we want that route to always stay in our forwarding table, regardless of what the routing protocol is doing.

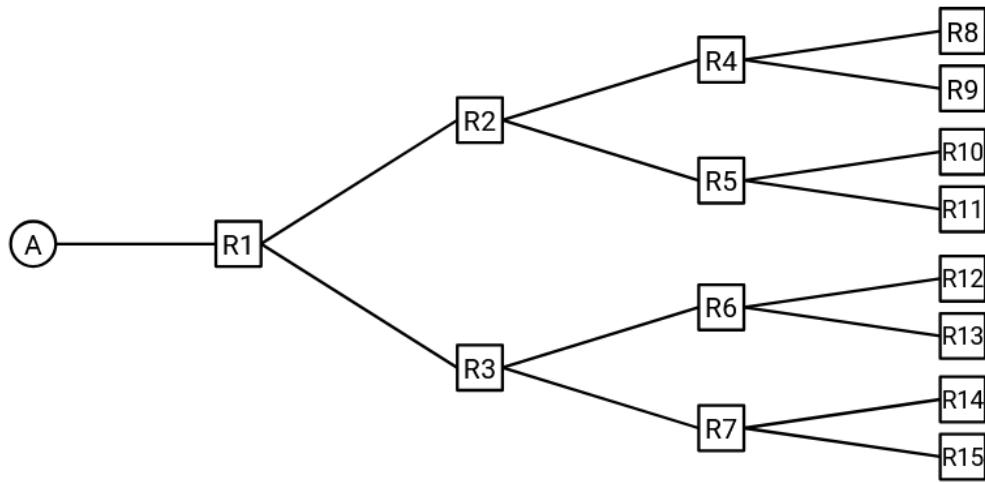
Distance-Vector Protocols

Algorithm Sketch

In this section, we'll design a **distance-vector protocol**, which is one of three classes of routing algorithms (along with link-state and path-vector).

Distance-vector protocols have a long history on the Internet and ARPANET (the predecessor to the Internet). The prototypical distance-vector protocol is the **Routing Information Protocol (RIP)**, and the D-V protocol we'll design shares many similarities with RIP.

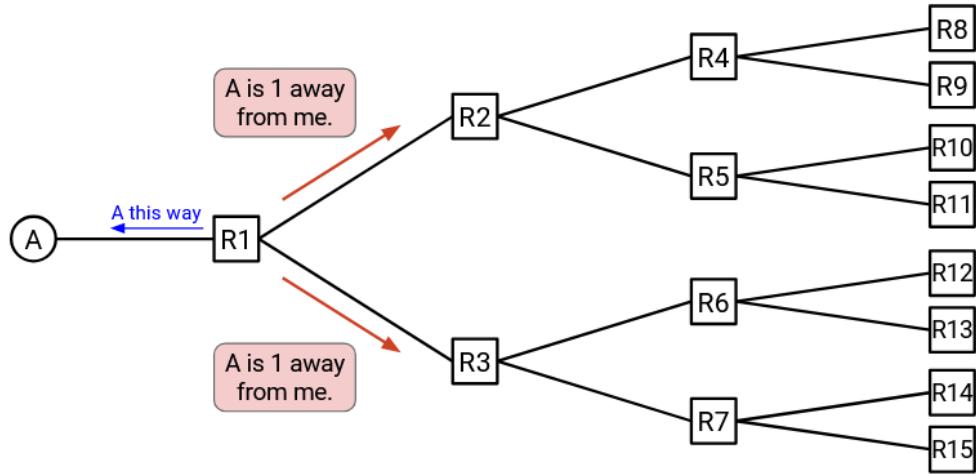
To gain some intuition for the routing protocol we'll study in this section, consider the following network.



To start out, every router's forwarding table is empty. Our goal is to fill in the forwarding tables of every router, such that packets can be routed from anywhere to the destination, A.

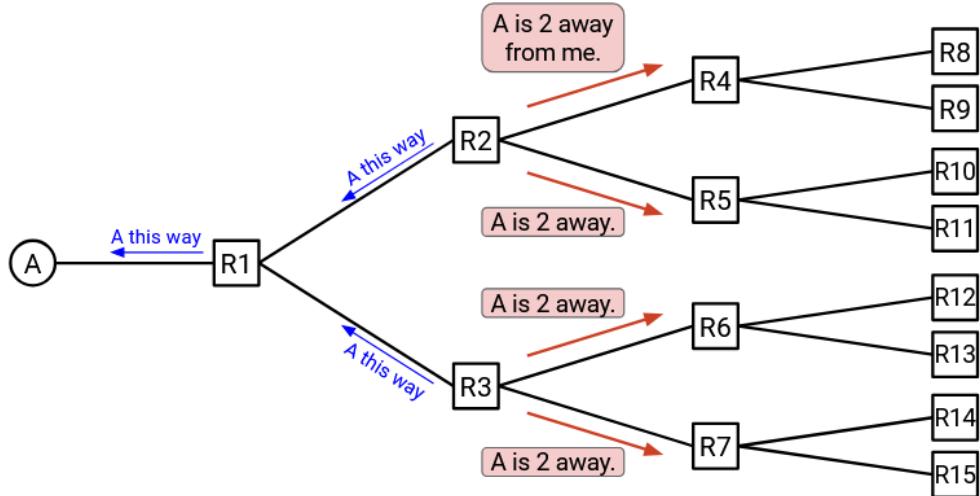
To start, A can tell R1: "I am A." Now, R1 knows how to forward packets to A.

Now that R1 has a path to A, it can tell its neighbors, R2 and R3: "I am R1, and I can reach A."



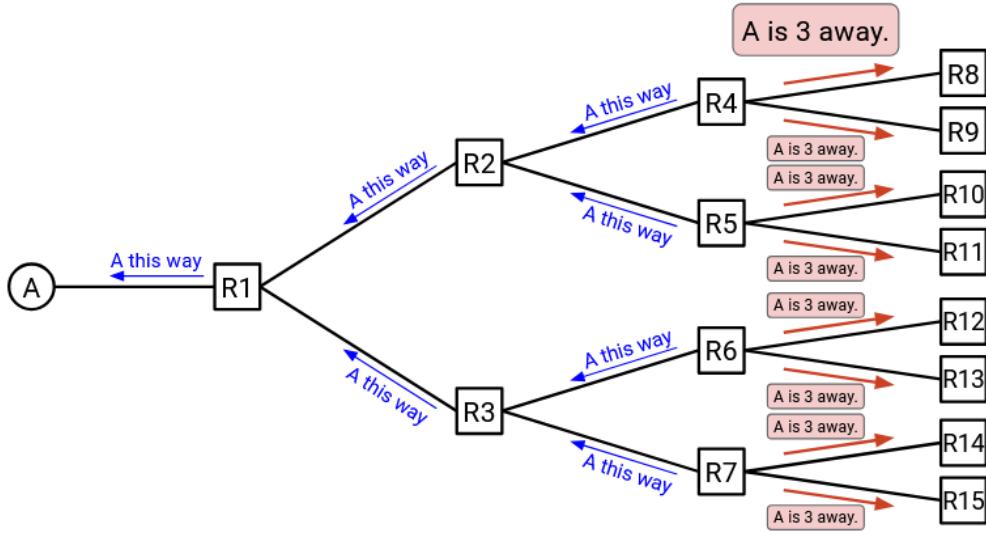
Now, R2 and R3 know that they can reach A by forwarding packets to R1.

R2 can now tell its neighbors, R4 and R5: "I am R2, and I can reach A." Similarly, R3 can tell its neighbors, R6 and R7: "I am R3, and I can reach A."



Now, R4 and R5 know that packets for A can be forwarded to R2, and R6 and R7 know that packets for A can be forwarded to R3.

The process continues: R4, R5, R6, and R7 each tell their neighbors who they are, and that they can reach A. By the end, everybody's forwarding table is filled in, and we can route packets from anywhere in the network towards A.



In summary: When you receive an announcement from someone saying they can reach A, you should write down who sent the announcement. Now, you can send messages bound for A through that person.

Also, now that you have a way to send messages to A, you should make an announcement to all of your neighbors, so that they can send messages bound for A through you.

What if there were multiple destinations? We could run this same algorithm repeatedly, once per destination. The forwarding table would then contain multiple entries, one per destination.

In these notes, we'll focus on a single section for simplicity, but the protocol we'll design can extend to multiple destinations.

Let's review our protocol so far.

For each destination:

- If you hear about a path to that destination, update the table.
- Then, tell all your neighbors.

Direction of Announcements and Messages

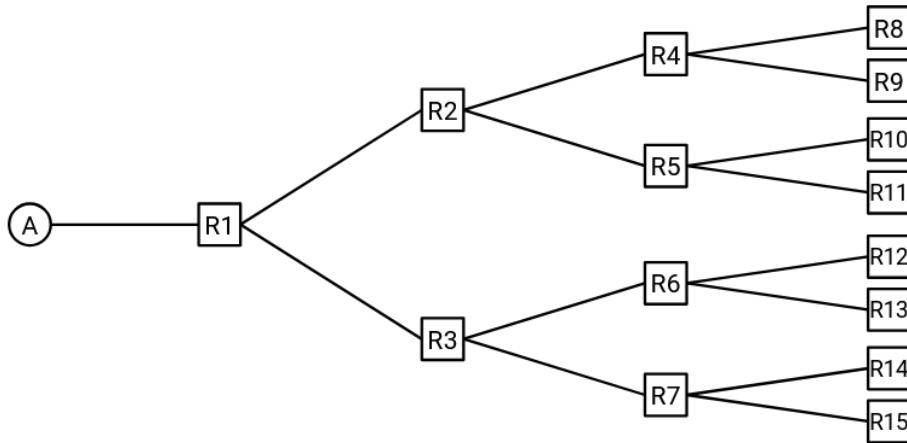
In this protocol, it's easy to confuse the direction in which announcements and messages are sent.

The announcements for how to reach A start at A, and propagate outward. For example, B sent an announcement to D, saying "I am B, and messages for A can be sent through me."

By contrast, the actual messages being sent to A are sent inward, toward A. For example, a message might start at D and be sent to B on its way to A.

Routing announcements ("I can reach A") propagated outward, away from A.

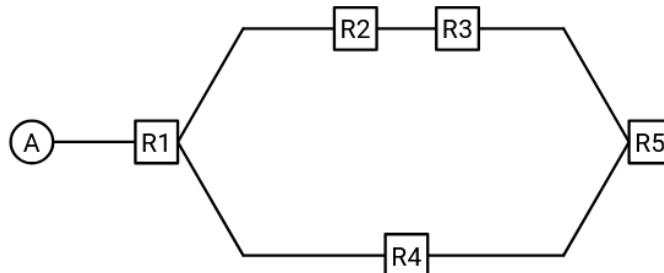
When **forwarding** packets toward A, packets travel *inward*, toward A.



The direction of announcements is exactly the opposite of the direction of the messages themselves. Be careful not to confuse announcements with the actual messages!

Rule 1: Bellman-Ford Updates

What if there are multiple paths to reach A?



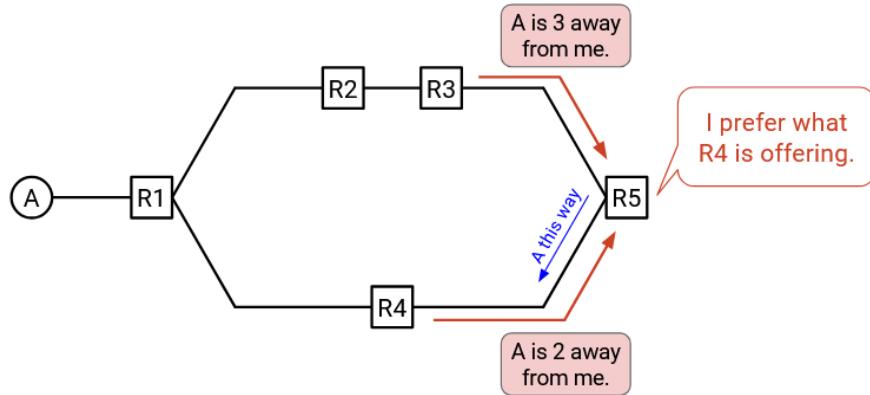
In this scenario, both R3 and R4 will announce that they can reach A. Should R5 choose to forward packets to R3 or R4?

Recall that our goal is to find least-cost routes through the network. To allow routers to pick the least-cost path out of multiple being advertised, we'll need to also include costs in the announcements.

R3's announcement now says: "I am R3, and I can reach A with cost 3."

R4's announcement now says: "I am R4, and I can reach A with cost 2."

Now, R5 notices that R4 is offering the shorter path, and decides to forward packets via R4.

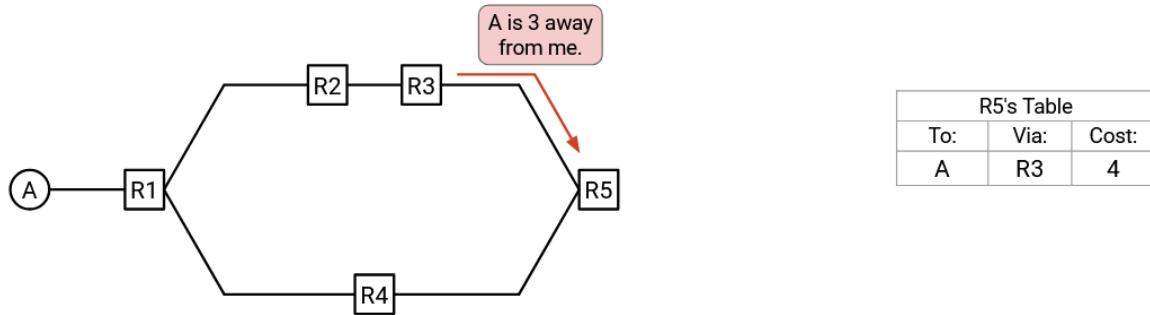


We'll use the forwarding table to remember the best-known cost to the destination (and the corresponding next-hop). Each entry of the forwarding table now tells us: the destination, the next-hop for that destination, and the cost to reach the destination via that next hop.

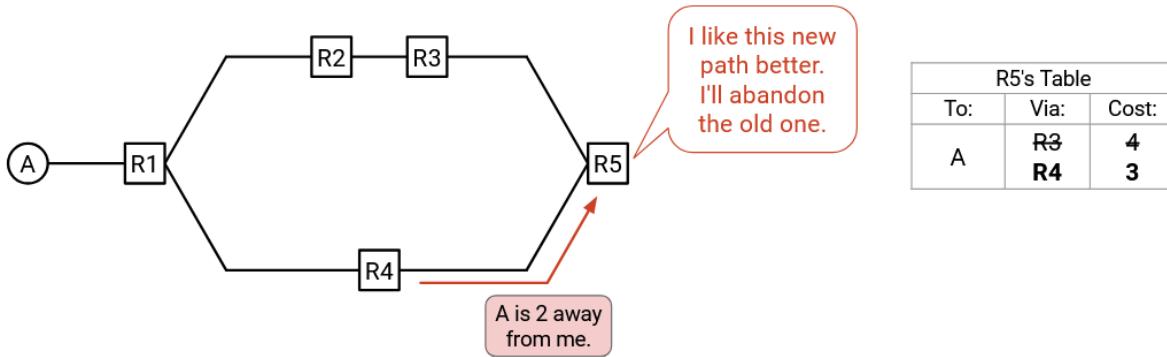
Note: Formally, the forwarding table stores key-value pairs, mapping each destination to a 2-tuple containing the next hop and the distance. We'll draw tables with 3 columns for simplicity.

R5 might not hear about both paths simultaneously, so we'll need to be more precise about what happens when we hear about a new path. There are three possibilities when we hear about a path:

1. If the table doesn't have a path to the destination, accept the path. If I don't have a way to reach A, I should accept any path offered.



2. If the new path (that we hear about) is better than the best-known path (from the forwarding table), we should accept the new path, and replace the old path from the table.



3. If the new path (that we hear about) is worse than the best-known path (from the forwarding table), we should ignore the new path, and keep using the path in the table.

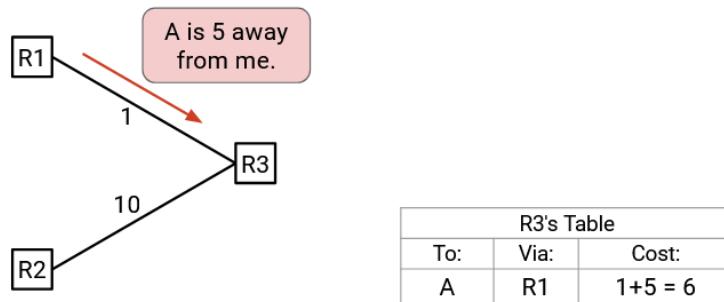
Let's review our protocol so far.

For each destination:

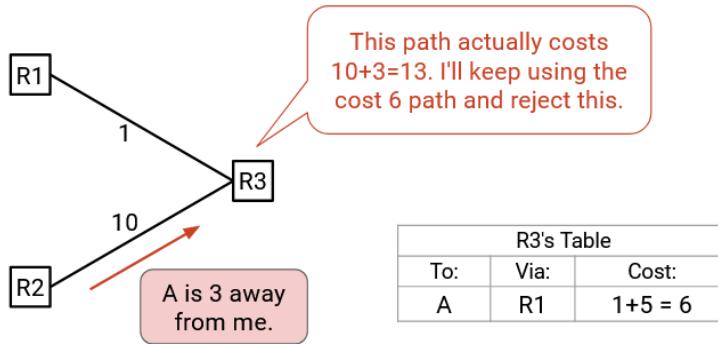
- If you hear about a path to that destination, update the table if:
 - The destination isn't in the table.
 - The advertised cost is better than the best-known cost.
- Then, tell all your neighbors.

How do we know if a new path is better or worse? We have to be careful, because not all link costs are the same. When someone advertises a path, the cost via that path is actually the sum of two numbers: The link cost from you to the neighbor, plus the cost from the neighbor to the destination (as advertised by the neighbor).

As a concrete example, suppose we hear: "I am R1, and A is 5 away from me." The cost of this new path is actually 1 (the link cost from us to R1), plus 5 (the cost from R1 to A, from the advertisement), which is 6.



Later, we might hear: "I am R2, and A is 3 away from me." It is incorrect to just look at the cost in the advertisement. In this case, the cost of the new path is actually 10 (the link cost from us to R2), plus 3 (the cost from R2 to A, from the advertisement), which is 13. This cost is not better than our best-known cost of 6, so we don't update the table. Packets still get forwarded to R1.



Let's review our protocol so far.

For each destination:

- If you hear about a path to that destination, update the table if:
 - The destination isn't in the table.
 - The advertised cost, **plus the link cost to the neighbor**, is better than the best-known cost.
- Then, tell all your neighbors.

For every announcement we hear, we have to compare two numbers. One number is the best-known cost in the table. The other number is the sum of the link cost to the neighbor, plus the advertised cost from neighbor to destination. If the latter number is lower, we use the new path and abandon the old path.

Rule 1: Distributed Bellman-Ford Algorithm

Does this operation look familiar? It turns out, this is exactly the relaxation operation from Dijkstra's shortest paths algorithm!

Bellman-Ford is another shortest paths algorithm that relies on relaxation as the key operation. Bellman-Ford is even simpler than Dijkstra's: Cycle through all the edges repeatedly, relaxing every edge, until we get all the shortest paths.

You might have implemented Dijkstra's or Bellman-Ford before in a data structures class, like CS 61B at UC Berkeley. Unfortunately, the code you wrote wouldn't be very useful for our routing protocol. Remember, the routing protocol must be distributed, because routers don't have a global view of the network (no central mastermind). Also, the routers are operating asynchronously. There's nobody enforcing the order in which routers perform relaxation operations, or the order in which routers send out announcements.

Instead, the routing protocol that we've been designing is a distributed, asynchronous version of the Bellman-Ford algorithm. The protocol is distributed, because we aren't asking a single computer to run the entire algorithm. Instead, every router is computing its own part of the answer (populating its own forwarding table) without seeing the entire graph. The protocol is asynchronous, because the routers can all run the algorithm at the same time, without needing to control the order of operations.

```

def bellman_ford(dst, routers, links):
    distance = {};  nexthop = {}
    for r in routers:
        distance[r] = INFINITY
        nexthop[r] = None
    distance[dst] = 0

    for _ in range(len(routers)-1):
        for (r1, r2, linkcost) in links:
            if distance[r1] + linkcost < distance[r2]:
                distance[r2] = distance[r1] + linkcost
                nexthop[r2] = r1

    return distance, nexthop

```

Everyone starts infinity away from the destination, except for the destination itself (0 away).

Bellman-Ford loops through nodes and relaxes repeatedly.

In distance-vector, each router relaxes in parallel, with no order between routers.

The relaxation operation.

Note: Although we're showing a single destination for simplicity, don't forget that our routing protocol will be able to find shortest paths to all destinations, just like the centralized (single-computer) Dijkstra's or Bellman-Ford algorithms.

Bellman-Ford Demo

Terminology note: When we send a message like "I am R1, and I can reach A with cost 5," to our neighbors, this is often called **announcing** or **advertising** a route. Notice that the advertisement contains three values: the destination, your identity (so your neighbors can forward to you), and the total cost from you to the destination.

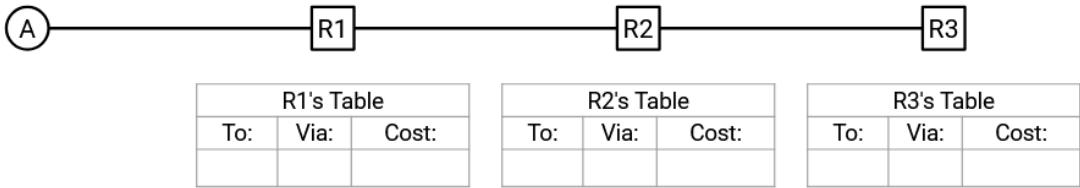
To restate the algorithm so far one more time:

When you receive an announcement from another router, you add the cost from the destination to the other router (this cost is in the announcement), plus the cost of the link from the other router to you. If this sum is less than the best-known distance to destination in your table, you replace your forwarding table entry for this destination with the new next hop (identity of the other router from the announcement) and the new distance (the sum you just computed).

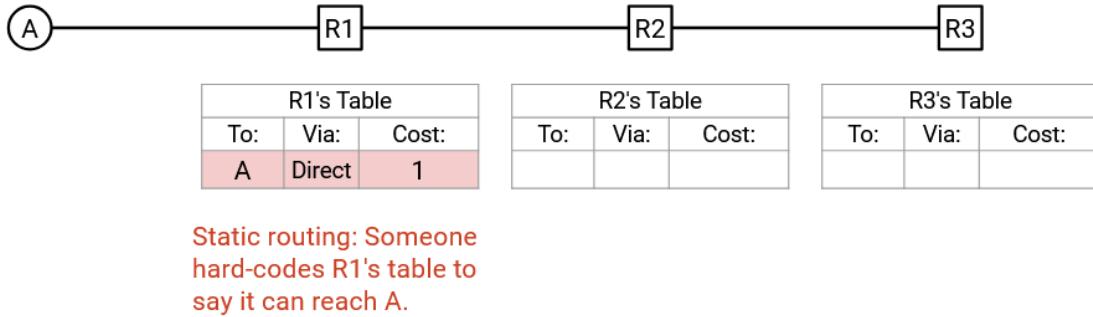
What if you receive an announcement from another router, and the destination isn't in your forwarding table? You don't have a best-known distance to this destination, because you don't know how to reach this destination yet. In this case, you can add a new entry to your forwarding table with the new destination, and the next hop and cost from the announcement.

When you change your forwarding table, that means that you've discovered a new path to the destination. In order to propagate this new path to the rest of the network, you will need to announce this new path (destination, your identity, and cost via you) to your adjacent routers.

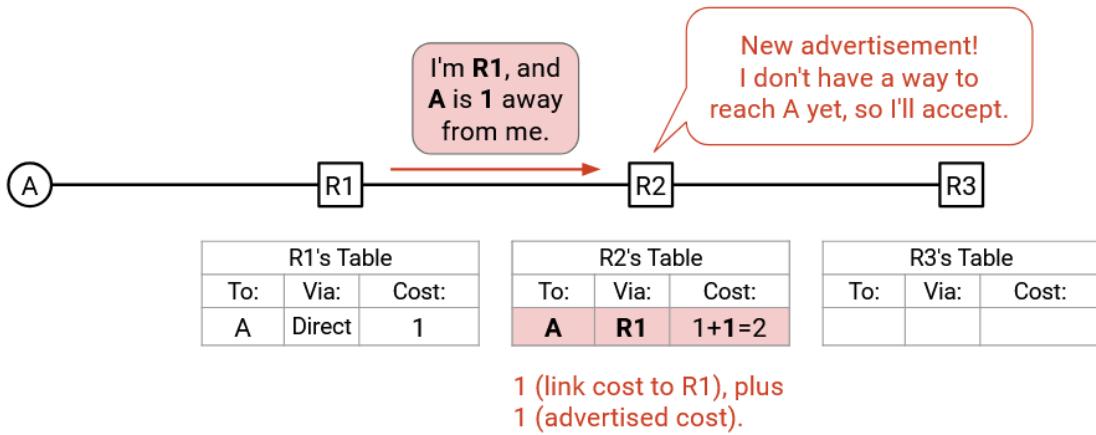
With this algorithm in mind, let's run through an example. In this network, we'll assume all edges have cost 1 since the edges are unlabeled. We want to populate the forwarding tables with routes to A, the one and only destination.



First, using static routing, we hard-code an entry in R1's forwarding table. To reach destination A, the next hop is A itself, and the cost of this path is 1.

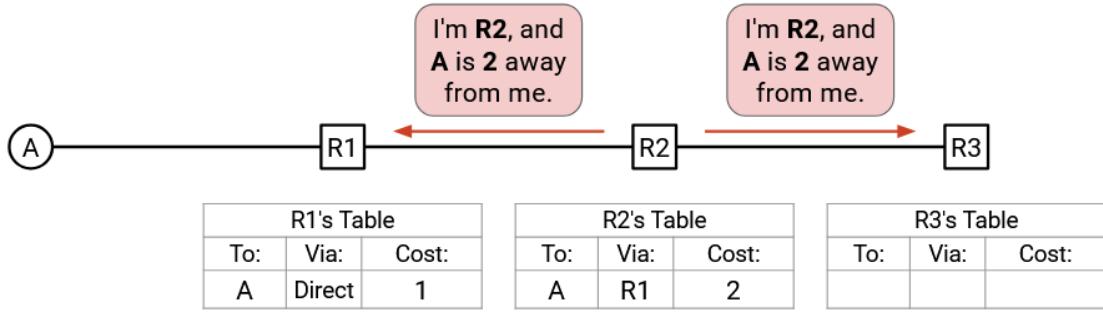


R1's forwarding table has changed, so R1 will create a new announcement with 3 values: the destination (A), the router's identity (R1), and the cost to the destination via this router (1). This announcement is sent to all of R1's adjacent routers, namely only R2.



R2 receives this announcement and looks in its forwarding table for an entry corresponding to destination A. The forwarding table is empty, so no such entry exists. Therefore, R2 will add a new entry with 3 values: the destination (A), the next hop (R1, from the announcement), and the cost to the destination via R1 (2, summing the cost in the announcement and the cost of the link to R1).

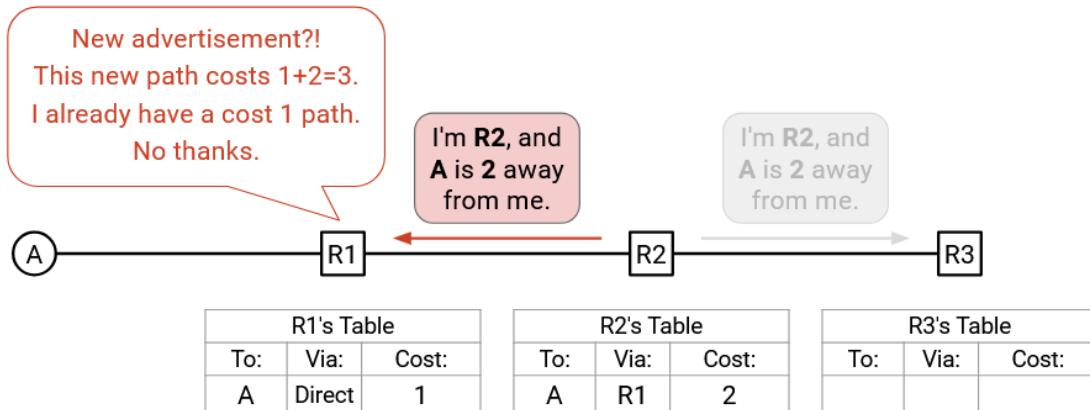
R2's forwarding table has changed, so R2 will make an announcement with 3 values: the destination (A), the router's identity (R2), and the cost to the destination via this router (2). This announcement is sent to all of R2's adjacent routers, namely R3 and R1.



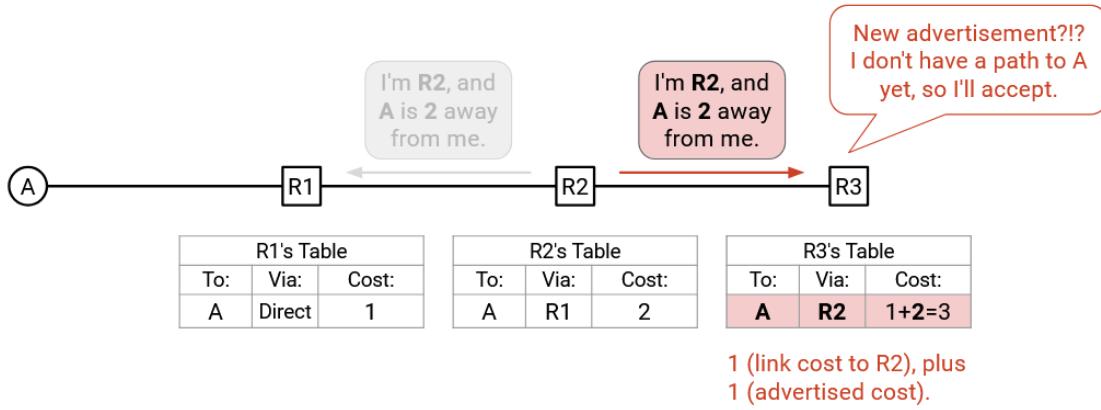
**Notice: R2's announcement doesn't include the next-hop.
Nobody else cares how R2 reaches A, just that R2 can reach A.**

Note that in our protocol so far, routers send announcements to all of their neighbors. This means that R2's announcement is sent to R1 as well. If this bothers you, stay tuned, we'll revisit it later.

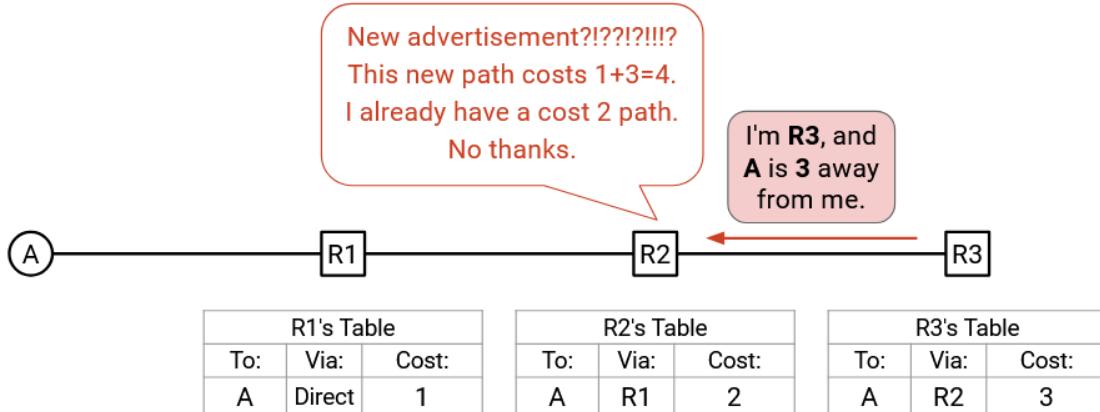
R1 receives this announcement. According to R1's forwarding table, the best-known way to reach A has cost 1. The path via R2 would instead cost 2 (from R2's announcement), plus 1 (link to R2), for a total of 3. This is a worse way to reach A, so R1 will ignore this announcement and leave its forwarding table unchanged.



R3 also receives the same announcement. R3's forwarding table is empty, so R3 will install a new entry with 3 values: the destination (A), the next hop (R2, from the announcement), and the cost to the destination via R2 (3 summing the cost from the announcement, and the cost of the R3-R2 link).



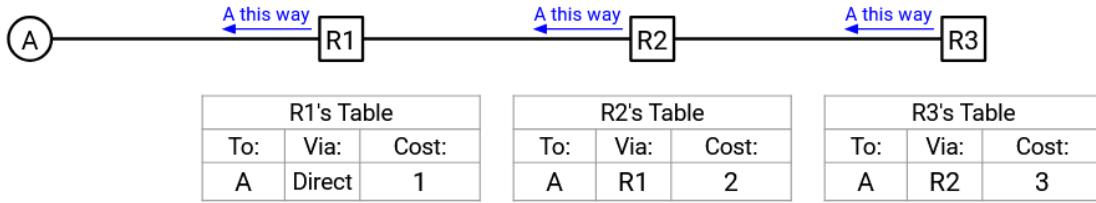
According to our rules so far, if you update your forwarding table, you need to send an announcement to all your neighbors. Even though we can see that this next announcement won't change anything, R3 doesn't have the same global view of the network that we have, so R3 will send an announcement to all of its neighbors, namely R2. The announcement contains: destination (A), next hop (R3), and cost via this next hop (3).



R2 receives this announcement. R2 knows of a way to reach A with cost 2, from the forwarding table. The announcement offers a path with cost 3 (from the announcement), plus 1 (cost of R2-R3 link), for a total cost of 4. This is worse than the cost in the forwarding table, so R2 ignores the announcement.

R2 did not update its forwarding table, so it does not make an announcement. At this point, no further announcements are made, and we can see that every router has populated its forwarding table with information about how to reach A. We can also see that the forwarding tables together form a valid, least-cost delivery tree with the shortest routes for reaching A.

We did it! Everybody has a way to reach A now.



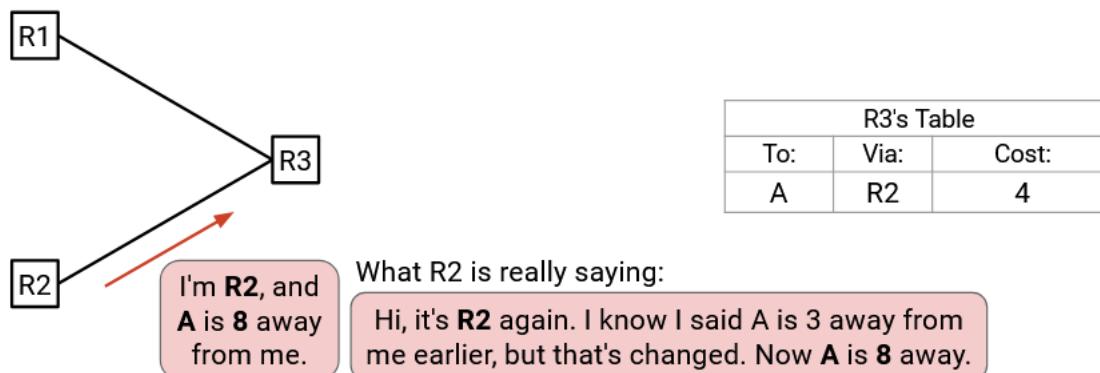
Rule 2: Updates From Next-Hop

Recall one of our routing challenges from the last section: The network topology can change.

Suppose that we hear an advertisement from R2, saying that A is 3 away from R2. If there's nothing in our table, we'll accept this advertisement and record a cost of $1+3=4$.



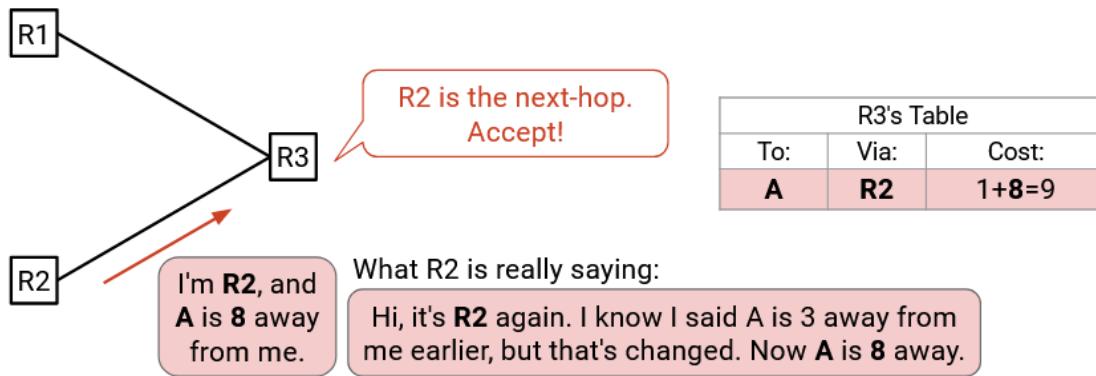
Later, we might hear a different advertisement from R2, saying that A is 8 away from R2. From the previous rule, we would reject this, because the advertised cost ($1+8=9$) is worse than our current cost (4).



However, we have to be careful about rejecting this advertisement. The router making the announcement (R2), was the same as the next hop router we were using. R2 is trying to say: "If you're using me as a next

hop, my distance to A is no longer 3, it's 8." But we ignored this message because we weren't thinking about the possibility that paths might change.

To fix this, we have to modify our update rule. If we hear an announcement from the next-hop router (the router with the best-known path that we were forwarding packets to), we should treat that announcement as an update, and edit our forwarding table. We should do this even if the announcement produces a worse path, because the next hop could be telling us that the path cost has changed and gotten worse.



Note that when this new rule applies, we don't update the destination or the next hop in the forwarding table, only the distance. In the example, packets at R3 destined for A are still forwarded to R2 (same destination, same next hop), but the cost via R2 changed.

Let's review our protocol so far.

For each destination:

- If you hear an advertisement for that destination, update the table if:
 - The destination isn't in the table.
 - The advertised cost, plus the link cost to the neighbor, is better than the best-known cost.
 - **The advertisement is from the current next-hop.**
- Then, tell all your neighbors.

In order to support changing topologies, routers will run the routing protocol indefinitely.

Suppose we ran the protocol indefinitely, with no topology changes. Initially, some relaxations will succeed and the forwarding tables will change. Eventually, the algorithm will **converge** when we have found all the least-cost routes through the network. At this point, if we continue relaxing the edges, the forwarding tables will not change. Every relaxation will be rejected, because the best-known paths to the goal are all the shortest paths, and we'll never find a better path to replace the current shortest paths. The state of the network at convergence is called **steady state**.

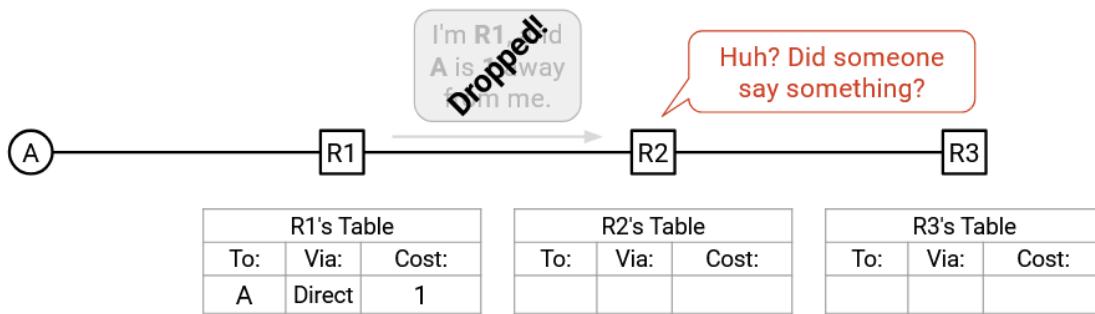
Later, suppose we change the topology (e.g. maybe a router fails). As we continue running the protocol, some relaxations might succeed again, since we've changed the underlying graph. After some time, the delivery tree will converge again on the new least-cost routes and stop changing until the next time the topology changes.

As an analogy, consider a pool of water. In the steady state, with no disturbances, the surface of the water is perfectly still. If you toss a rock in the water, there will be some ripples as the environment adjusts to the change you just made, but after some time, the surface of the water will become perfectly still again.

Rule 3: Resending

Recall another one of our routing challenges from the last section: Packets can get dropped.

For example, let's rewind to the very beginning of the example from earlier. R2 and R3 have empty forwarding tables, and R1 is updated with the hard-coded route to A. What if R1 issues an announcement, but the packet is dropped? R2 never hears an announcement, and the protocol fails.



You could try to design a more complicated scheme to ensure reliability (e.g. forcing recipients to send acknowledgements), but let's use something simple: If you have an announcement to make, re-send that announcement every few seconds. It turns out this simple approach works well with some of our later design choices, and nothing more complicated is necessary.

Formally, the protocol will define an **advertisement interval**. 30 seconds is a common interval used in practice. If the interval is X seconds, then every advertisement must be re-sent every X seconds.

As long as we wait long enough and re-send the packet enough times, the link will eventually successfully send the advertisement, as long as the link works some of the time. If the link was dropping every single packet, then there's no way for the advertisement to be sent (and maybe a link with 0% success rate probably shouldn't be in the graph anyway). Eventually, with enough re-sending, this protocol will still converge.

Let's review our protocol so far.

For each destination:

- If you hear an advertisement for that destination, update the table if:
 - The destination isn't in the table.
 - The advertised cost, plus the link cost to the neighbor, is better than the best-known cost.
 - **The advertisement is from the current next-hop.**
- Advertise to all your neighbors **when the table updates, and periodically (advertisement**

interval).

Note that re-sending at intervals can work in combination with our rule from earlier, where we sent an announcement any time the forwarding table changes. Announcements sent immediately after a change are called **triggered updates**.

The protocol would still converge if we only sent announcements at intervals. The table changes, we wait for the interval to expire, and send out the announcement. However, adding triggered updates in addition to interval updates is an optimization that can help the protocol converge quicker. As soon as we know the update, we might as well announce it, without waiting for the interval.

With this new rule, once the network converges, every router will continue to re-send announcements periodically, but none of the announcements will be accepted, because we're in steady state and everybody already has the shortest-cost routes.

In the example from earlier, after the network converges, R3 might decide to re-send its announcement, with destination A, next hop R3, and cost via R3 of 3. But R2 will ignore this announcement because its forwarding table has a cheaper route of cost 2 already (the announcement path costs $3 + 1 = 4$).

Rule 3: Expiring

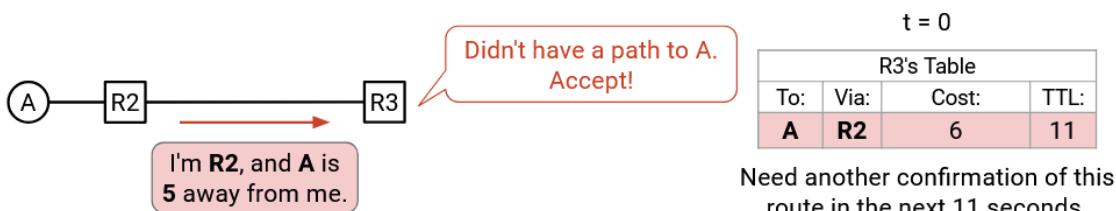
Recall our routing challenge from earlier: The network topology can change. In particular, links and routers can fail. If a router fails in the network, our route might become invalid. The failed router won't tell us about the problem (since it's failed), so we're stuck with this invalid route.

To solve this problem, we'll give every route (i.e. every table entry) a finite **time to live (TTL)**. This is a countdown timer, telling us how much longer we can keep this forwarding entry.

Periodic updates help us confirm that a route still exists. If we get an advertisement from the next-hop, we can reset ("recharge") the TTL to its original value.

If something in the network fails, we'll stop getting periodic updates. Eventually, the TTL will expire. If the TTL expires, we'll delete the entry from the table. Intuitively: We aren't getting updates anymore, so this route is probably no longer valid.

Here's an example of the TTL in action. In this example, we are R3. At time $t=0$, we hear an announcement: "I'm R2, and A is 5 away from me." Our table doesn't have an entry for A, so we'll accept this path, and set its TTL to 11. Notice that this TTL is associated with the specific table entry. If we had multiple table entries, they would each have their own TTL.



The TTL of 11 tells us that R2 must send us another confirmation of this route in the next 11 seconds.

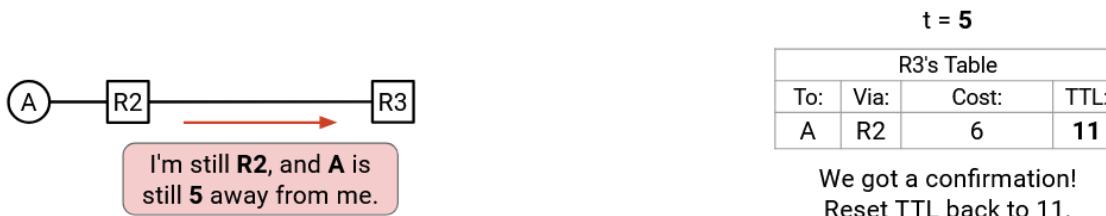
Otherwise, this table entry will be deleted. (Note: The initial TTL of 11 was chosen arbitrarily. In practice, this number would be set by the protocol or the person operating the router.)

Time passes. At $t=1$, the TTL is now 10. At $t=2$, the TTL is now 9. At $t=3$, the TTL is now 8. At $t=4$, the TTL is now 7.



At $t=5$, R2 does its periodic re-sending of the announcement: "I'm R2, and A is 5 away from me." We look in our table and realize that R2 is the current next-hop to A, so we should accept this advertisement (per Rule 2) and update the table.

Because we got a confirmation of this route still existing, the TTL can be reset back to its initial value of 11. We need to get another confirmation of this route from R2 in the next 11 seconds.



Suppose that a link goes down at $t=6$, and A is now unreachable. R2 removes its static route to A, and no longer sends any periodic updates.

At $t=16$ (11 seconds after the last update at $t=5$), the TTL in our table entry has decreased all the way to 0, so we'll delete the entry from our table.



Let's review our protocol so far.

For each destination:

- If you hear an advertisement for that destination, update the table **and reset the TTL** if:
 - The destination isn't in the table.

- The advertised cost, plus the link cost to the neighbor, is better than the best-known cost.
- The advertisement is from the current next-hop.
- Advertise to all your neighbors when the table updates, and periodically (advertisement interval).
- **If a table entry expires, delete it.**

Be careful not to confuse the various timers that the router must maintain.

The advertisement interval tells the router when to advertise routes to neighbors. This is usually a single timer for the entire table, so the router advertises all the routes in the table whenever the advertisement interval timer expires. In the example above, the advertisement interval timer was 5 seconds, since R2 sent advertisements at t=0 and t=5.

By contrast, the TTL tells the router when to delete a table entry. Each table entry has its own independent TTL, counting down for that specific entry. In the example above, the initial TTL was 11 seconds (reset to 11 when we accept an advertisement), and counted down for each table entry.

At this point, we have a mostly-functional routing protocol! Let's add some optimizations for faster convergence.

Rule 4: Poisoning Expired Routes

Waiting for routes to expire is slow. To see why, let's rewatch the demo from earlier.

In this example, we are R3. Assume that by t=5, we've learned a route to A, via R2, and this route has 11 seconds of TTL remaining.



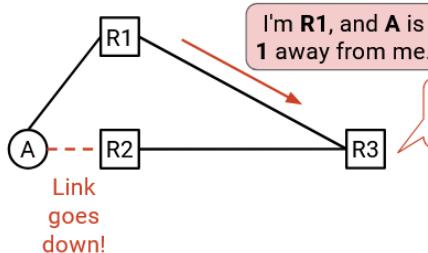
At t=6, the A-to-R2 link goes down! The table entry is now busted, because if we forwarded packets to R2, they wouldn't actually reach A. However, we don't know that this entry is busted yet. We have to wait another 10 seconds for this route to expire.

Also at t=6, we get a new announcement: "I'm R1, and A is 1 away from me." We look in our table, and we already have a way to reach A, so we reject this announcement. (Note: It's not important for this demo, but we're assuming we don't accept equal-cost paths here.)

If only we knew that our existing route is busted, we could accept this new advertisement right now. But instead, we're doomed to wait another 10 seconds of using this busted path.

Pause right here.

- At this point, we know the path via R2 is busted.
- But R3 won't know until the timeout 10s later.
- If R3 knew now, it could accept the new path. Instead, R3 rejects the new path, thinking the busted path is still valid.

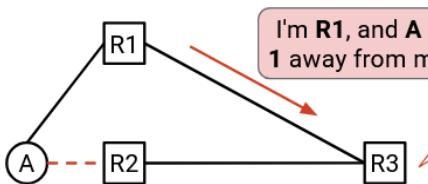


t = 6			
R3's Table			
To:	Via:	Cost:	TTL:
R2	A	2	10

Time passes. By $t=11$ (five seconds later), the busted route still has 5 seconds of TTL remaining.

At $t=11$, we get another announcement: "I'm R1, and A is 1 away from me." R1 is re-sending its announcement from earlier. Again, we look in our table, and we still have an entry for A, so we reject this announcement again.

Again, if only we had some way to know that our existing route is busted...then we could accept this new advertisement. With our current approach, however, we're doomed to keep using the busted path for the remaining 5 seconds.

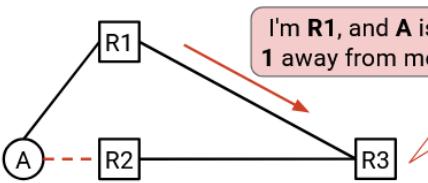


t = 11			
R3's Table			
To:	Via:	Cost:	TTL:
R2	A	2	5

Time passes. By $t=16$ (five seconds later), the busted route TTL finally reaches 0, and we can delete this entry from the table.

Also at $t=16$, R1 re-sends its announcement again: "I'm R1, and A is 1 away from me." Finally, our table doesn't have a route to A (the busted route just got deleted), so we can accept this announcement.

Can we alert R3 of the failure sooner, so it can delete the old busted path earlier (and start accepting new paths)?



t = 16			
R3's Table			
To:	Via:	Cost:	TTL:
R2	A	2	0

Timeout! Delete expired entry.

What just happened? At $t=6$, the failure occurred, and the entry in our table became busted. However, because there were 10 seconds of TTL remaining on the busted route, we were doomed to keep using the busted route for another 10 seconds. During this time, any packets to A will get lost, because we'll forward the packet along a busted path. Also, we might advertise this busted route to other people, causing them to lose packets as well. Finally, as we saw, we might reject new paths, thinking that the busted path is still valid.

The key problem here is: When something fails, it's not being reported, so we're forced to rely on timeouts to delete busted paths. This is slow. Is there any way we can detect failures earlier?

The solution is **poison**: When something fails, if possible, explicitly advertise that a path is busted.

In English, the new poison announcement that R2 sends would say: "I'm R2, and I no longer have a way to reach A." In the protocol, we encode this message by advertising a path with cost infinity: "I'm R2, and A is infinity away from me." This infinite-cost path represents a busted path.

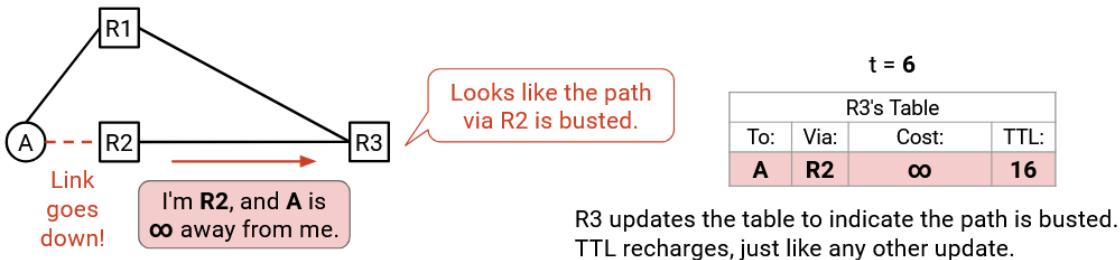
Poisoned paths propagate just like any other path. If we're forwarding packets to R2, and we get a poison message from R2, we update our forwarding table and replace the cost with infinity (per Rule 2). We can also advertise this infinite-cost poison to our neighbors, so they are also alerted of the busted path. This allows an invalid path to propagate through the network, which can be much faster than waiting for the path to time out.

Let's rewatch the demo from earlier, but with poisoning on route expiry. As before, assume that by $t=5$, we've learned a route to A, via R2, and this route has 11 seconds of TTL remaining.



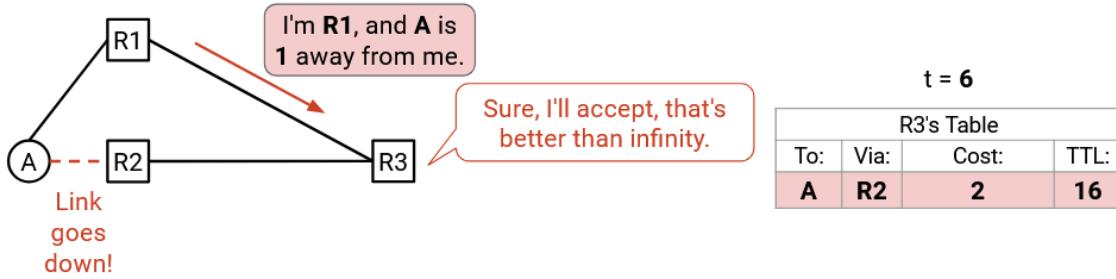
At $t=6$, the A-to-R2 link goes down! The table entry is now busted. However, we don't know that this entry is busted just yet.

With our modification, instead of saying nothing, R2 sends us a poison announcement: "I'm R2, and A is infinity away from me." Per Rule 2 (accept from next-hop), we notice that R2 is our next hop, so we accept this announcement and update our table.



Our table entry now encodes the fact that A is actually unreachable via R2. This entry has a TTL, just like any other table entry. Also, we can advertise this infinite-cost path to our neighbors, just like any other entry. This tells our neighbors that we can no longer reach A either.

Also at $t=6$, after our table update, we get a new announcement: "I'm R1, and A is 1 away from me." Using this route has distance 2 (1 from link, 1 from advertisement), which is better than infinity (from the table). We accept this advertisement and update the table. Now, packets for A are routed through R1 instead of R2.



In our earlier demo, at $t=6$, we were forced to wait 10 seconds for the busted route to expire. Thanks to the poison announcement, we were able to immediately invalidate that busted route at $t=6$, and accept the new path.

With poison, we were able to converge on a valid path sooner. Between $t=6$ and $t=16$, packets will now correctly reach A (whereas in the no-poison approach, packets in this time period would get lost). Also, thanks to the poison, we've avoided propagating a busted route to others in that time period. Even better, we can propagate the poison to others and let them know that the path to A via us (and R2) is busted.

Let's formalize the rules of poison. Poison originates from one of two sources: One or your routes times out, or you notice a local failure (e.g. one of your links goes down). When one of these occurs, you can update the appropriate table entry with cost infinity, reset the TTL, and advertise this poison to your neighbors.

How does poison propagate? When you receive a poison advertisement from your current next-hop, accept it. Your next-hop is telling you that the route no longer exists (similar to advertising worse paths in Rule 2), so you need to update your table. When you update the table, you reset the TTL, just like any other table update. You also advertise the poison to your neighbors, just like any other table update, so that your neighbors also know about the busted route.

One final modification: Now that our tables contain poison, we have to be careful not to forward packets along a poisoned route. If a table entry says that A is reachable via R1 with cost infinity, this really means that A is unreachable via R1. If we get a packet destined for A, we cannot forward it to R1.

To:	Via:	Cost:
A	R1	∞

← Don't forward to R1.

Let's review our protocol so far.

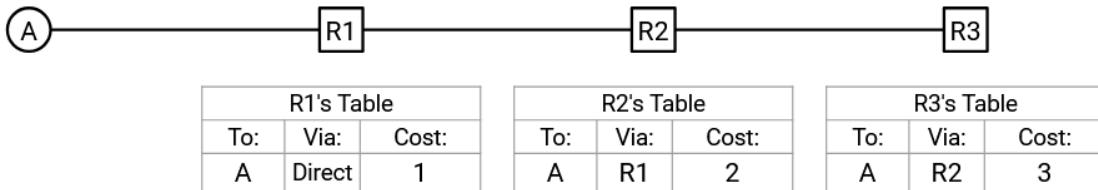
For each destination:

- If you hear an advertisement for that destination, update the table **and reset the TTL** if:
 - The destination isn't in the table.
 - The advertised cost, plus the link cost to the neighbor, is better than the best-known cost.
 - The advertisement is from the current next-hop. **Includes poison advertisements.**
- Advertise to all your neighbors when the table updates, and periodically (advertisement interval).
- If a table entry expires, **make the entry poison and advertise it.**

Rule 5A: Split Horizon

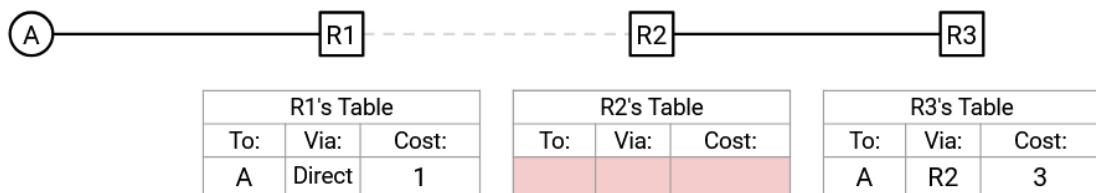
Let's go back to our favorite running example again to demonstrate another problem. Suppose we're in steady state, and the forwarding tables have the correct shortest routes to A. Announcements are being periodically re-sent, but all announcements are being rejected because we're in steady state.

We ran the algorithm for some time, and we converged to this steady-state.
All subsequent advertisements will be rejected.



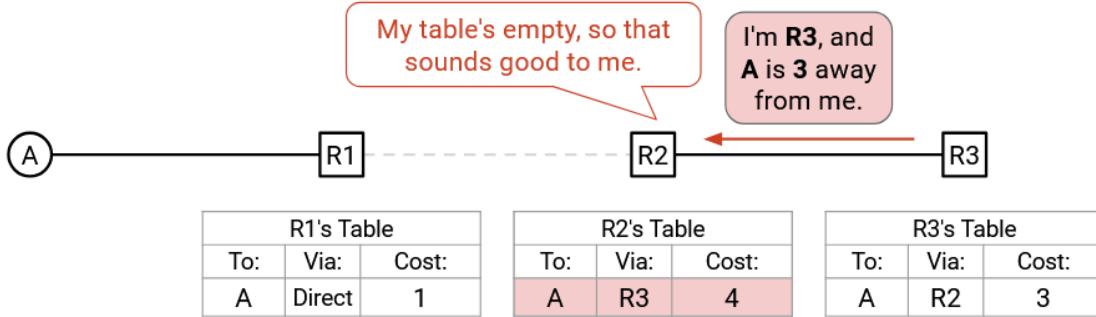
The R1-R2 link goes down, and R2's entry expires, because R1 stopped sending periodic announcements. R2 now has an empty forwarding table. What happens next?

A link goes down, and R2's entry expires (no more updates from R1).
What happens now?

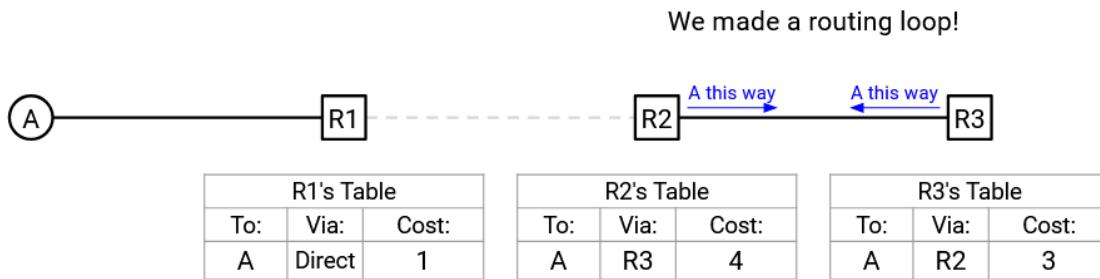


Eventually, R3 re-sends its announcement to R2, with destination (A), next hop (R3), and cost via next hop (3).

R2's table is empty, so it accepts this announcement and adds destination (A), next hop (R3), and cost via next hop ($3 + 1 = 4$).



We've created a routing loop! R2 will forward packets to R3, and R3 will forward packets to R2.



This problem can be tricky to spot at first, so let's restate it intuitively. Suppose I have accepted a route from Alice, which means that I'll be forwarding packets to Alice. What happens if I then offer this route back to Alice? If she accepts the route, she'll end up forwarding packets to me, and I'll forward the packet back to her.

If the network topology never changed, this advertisement is harmless. The path I'm offering to Alice goes from Alice, to me, back to Alice. This new path is definitely more expensive because it adds an unnecessary loop, so Alice will always reject this advertisement.

However, this advertisement is dangerous if Alice loses her route. Now, my advertisement is fooling Alice into thinking that she can send packets to me. But, my path relies on Alice herself, so if she accepts this path, we would create a loop where she sends packets to me, only for me to send the packet right back to her. The key problem here is: Alice thinks that the path I'm advertising is independent and never goes through Alice. But in fact, my path does go through Alice, so if she accepts my path, she'll end up forwarding packets back to herself.

To solve this problem, we need to avoid offering Alice a route that already involves herself. We never want Alice to accept a route that sends packets back to herself.

This leads us to a solution called **split horizon**, where we never advertise a route back to the person who gave us that route.

Let's review our protocol so far.

For each destination:

- If you hear an advertisement for that destination, update the table **and reset the TTL** if:

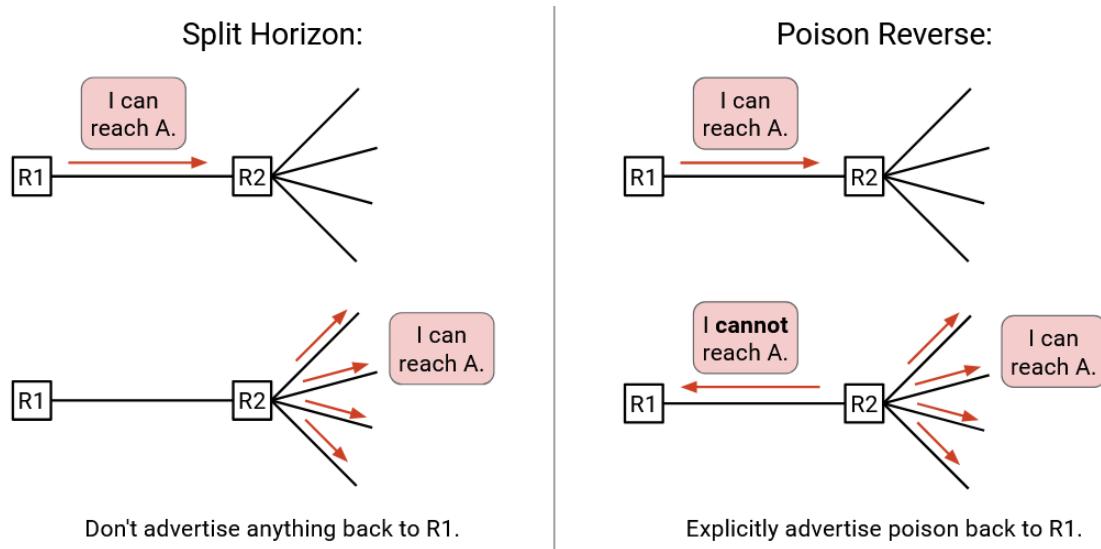
- The destination isn't in the table.
 - The advertised cost, plus the link cost to the neighbor, is better than the best-known cost.
 - The advertisement is from the current next-hop. Includes poison advertisements.
- Advertise to all your neighbors when the table updates, and periodically (advertisement interval).
 - **But don't advertise back to the next-hop.**
 - If a table entry expires, make the entry poison and advertise it.

Rule 5B: Poison Reverse

Poison reverse is an alternative way to avoid routing loops. We can use either split horizon or poison reverse to solve the problem from earlier (but not both).

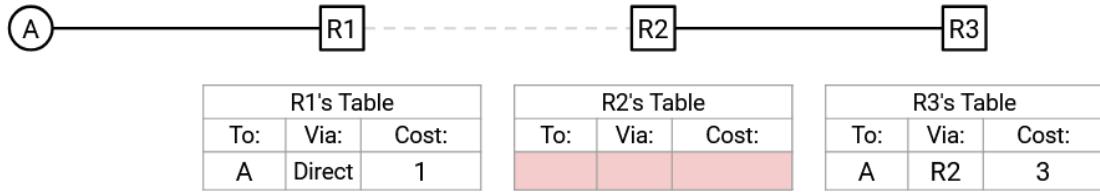
In split horizon, if someone gives me a route, I don't advertise the route back at them.

By contrast, in poison reverse, if someone gives me a route, I explicitly advertise poison back at them. In other words, I explicitly tell them, “Do not forward packets my way” (because I'd just forward them back to you).



Let's see the demo again, but using poison reverse instead of split horizon this time. As before, we reach steady state, then R1-R2 goes down, and R2 loses its table entry.

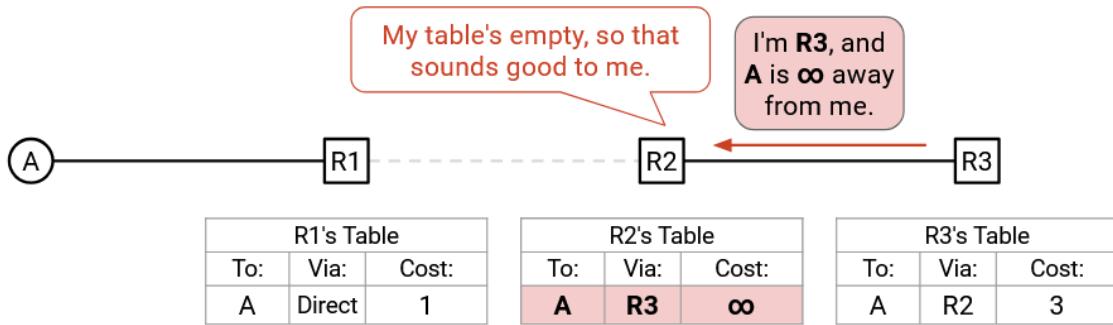
A link goes down, and R2's entry expires (no more updates from R1).
What happens now?



If we implemented neither fix, this is the point when R3 would advertise its route to R2, and R2 would accept a route going through itself.

If we implemented split horizon, R3 would not advertise its route back to R2 at this point.

In the poison reverse approach, R3 explicitly sends an advertisement back to R2: “I’m R3, and A is infinity away from me.”

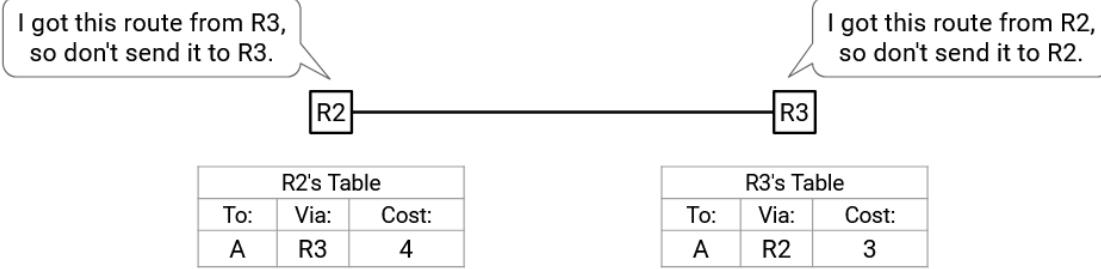


R2 doesn't have an entry for A (its old one expired), so it accepts this new, poisoned route. Now, R2's table explicitly says that it cannot reach A via R3. We've avoided the routing loop with the help of poison reverse!

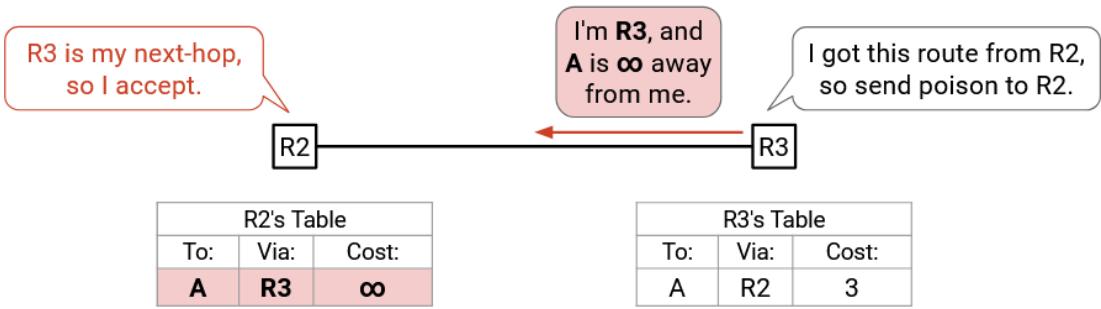
In our model of the network, split horizon and poison reverse will both help avoid routing loops. More generally, poison reverse can help eliminate routing loops sooner if they ever arise.

For example, suppose we end up with a routing loop somehow, where R2 and R3 are forwarding packets to each other.

In the split horizon approach, no poison gets sent. R2 got its route from R3, so it won't send anything to R3. Similarly, R3 got its route from R2, so it won't send anything to R2. The loop exists until the table entries expire. Until then, packets could get lost in the loop.



By contrast, if we used the poison reverse approach, R3 explicitly sends poison back to R2: “I’m R3, and A is infinity away from me.” R2 accepts this advertisement (Rule 2, route from its next-hop), and updates its table to invalidate the path via R3. The poison reverse advertisement immediately eliminates the routing loop.



Let’s review our protocol so far.

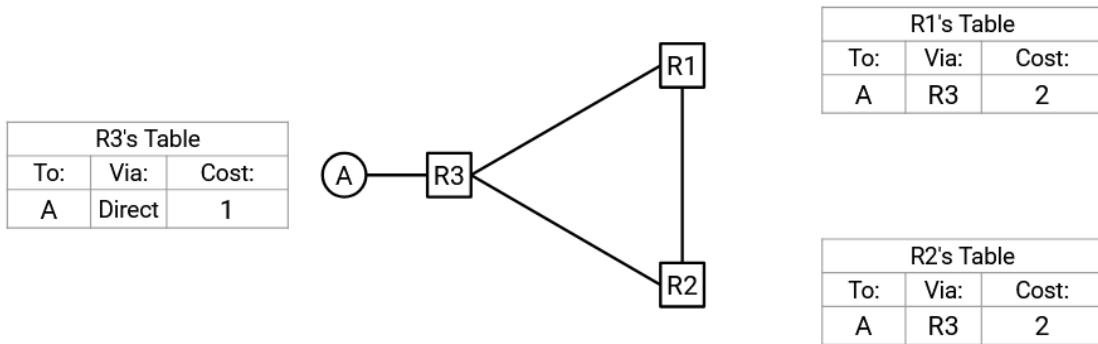
For each destination:

- If you hear an advertisement for that destination, update the table **and reset the TTL** if:
 - The destination isn’t in the table.
 - The advertised cost, plus the link cost to the neighbor, is better than the best-known cost.
 - The advertisement is from the current next-hop. Includes poison advertisements.
- Advertise to all your neighbors when the table updates, and periodically (advertisement interval).
 - But don’t advertise back to the next-hop.
 - ...Or, **advertise poison back to the next-hop**.
- If a table entry expires, make the entry poison and advertise it.

Note that split horizon and poison reverse are two choices, and you can pick exactly one to use (not both). Either you say nothing back to the next-hop, or you explicitly advertise poison back to the next-hop.

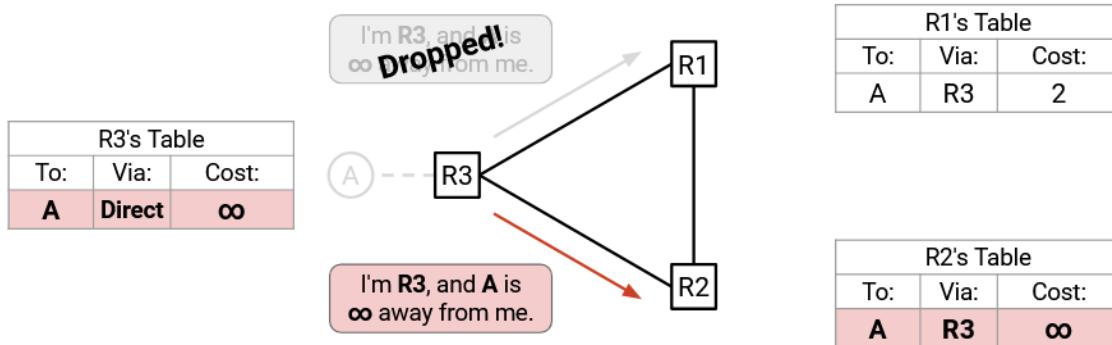
Rule 6: Count to Infinity

Split horizon or poison reverse helped us avoid length-2 loops, where R1 forwards to R2, and R2 forwards to R1. But we can still get routing loops involving 3 or more routers.



To see why, consider this network. Suppose the tables reach steady-state. R1 and R2 both forward to R3, which forwards to A.

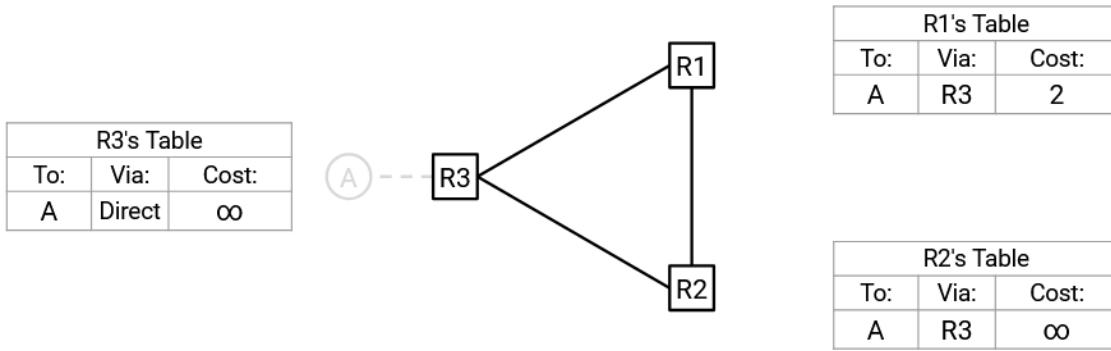
The A-R3 link goes down! A is now unreachable. Per Rule 4, R3 updates its table to show infinite cost to A, and sends this poison to both R2 and R1.



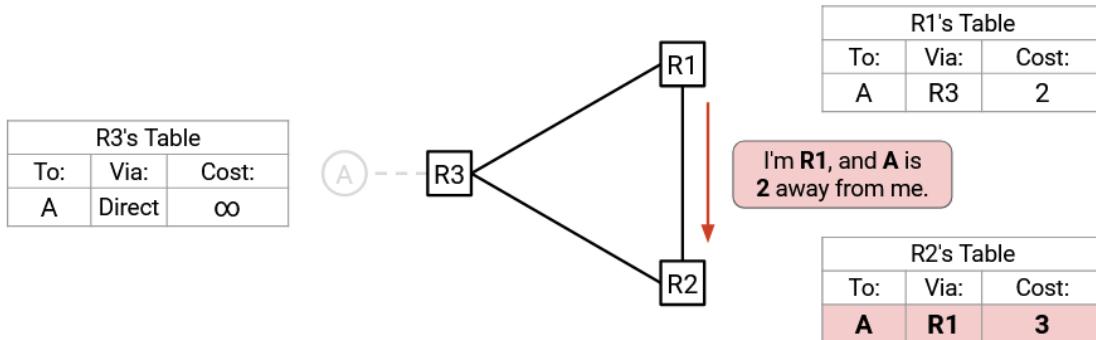
R2 gets the poison advertisement and updates its table (Rule 2, accept from next-hop). Now, both R2 and R3 know that A is unreachable.

The poison advertisement to R1 is dropped! R1 doesn't see the poison, so it still thinks it can reach A via R3. (The poison can get re-sent later, but for this demo, all the bad things that are about to happen will happen before the poison gets a chance to be re-sent.)

At this point, R2 and R3 can't reach A, but R1 thinks that it can still reach A.

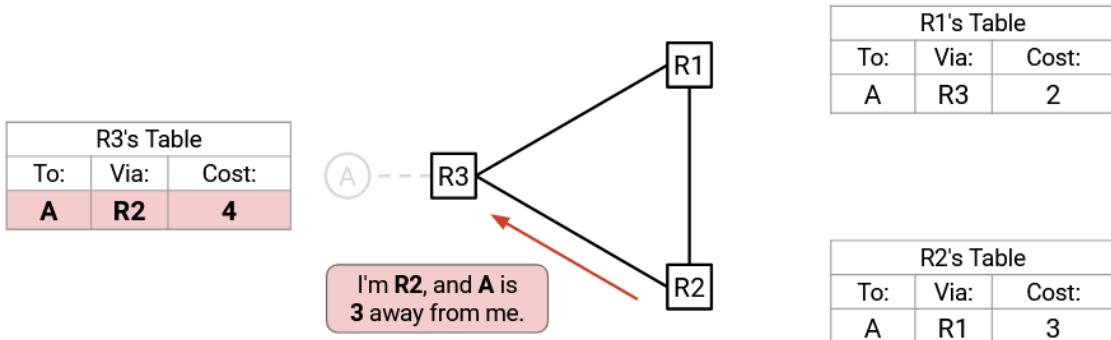


Eventually, R1 sends out an advertisement. R1's path to A is via R3, so by split horizon, it won't advertise to R3. However, R1 will still advertise to R2: "I'm R1, and A is 2 away from me."



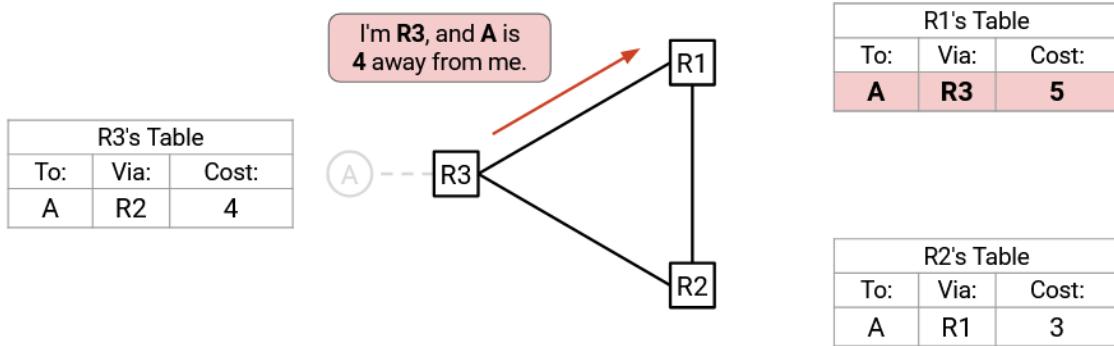
R2 doesn't have a way to reach A, so it accepts this route. Now, R2 is fooled into thinking it can reach A with cost 3.

R2 sends out an advertisement about its new route. Split horizon dictates that R2 won't advertise back to R1, but it will still advertise to R3: "I'm R2, and A is 3 away from me."



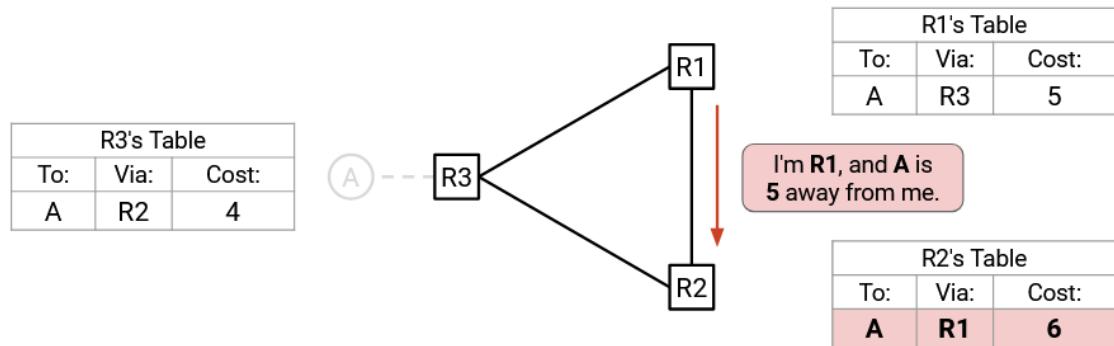
R3 doesn't have a way to reach A, so it accepts this route. Now, R3 is fooled into thinking it can reach A with cost 4.

Next, R3 sends out an advertisement to R1 (not R2, per split horizon): "I am R3, and A is 4 away from me."

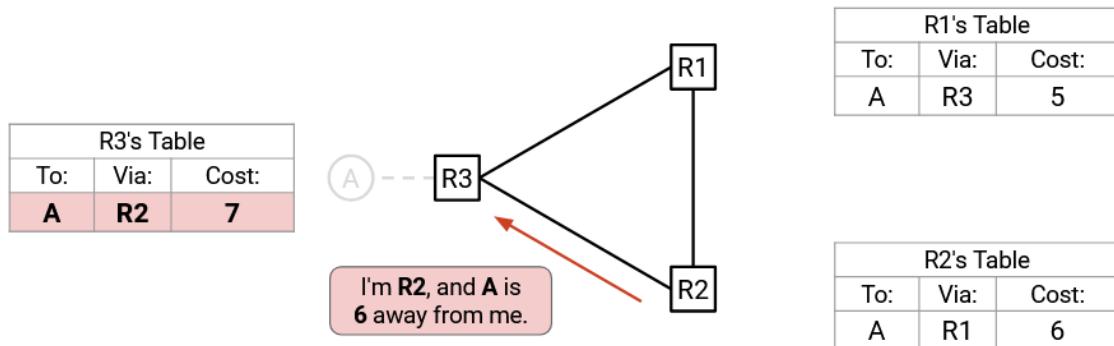


R1 will accept this advertisement (Rule 2, advertisement from next-hop) and update its table. Now, R1 thinks its cost to A is 5.

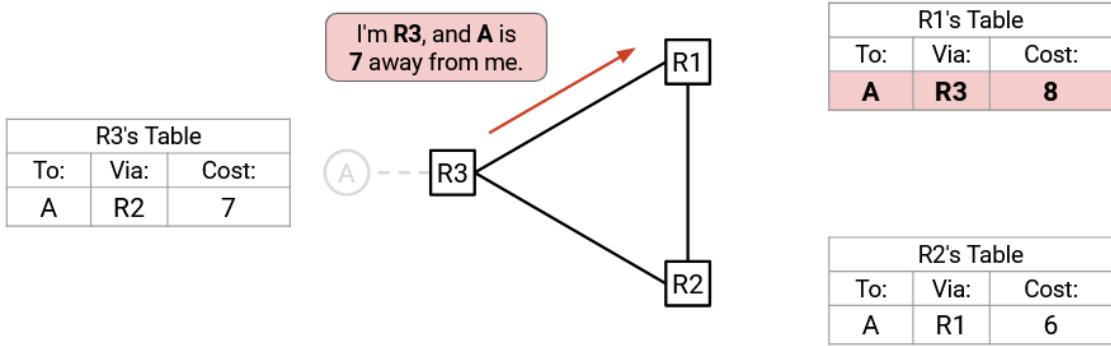
Maybe you're seeing where this is going. R1 advertises to R2 (not R3, per split horizon): "I'm R1, and A is 5 away from me."



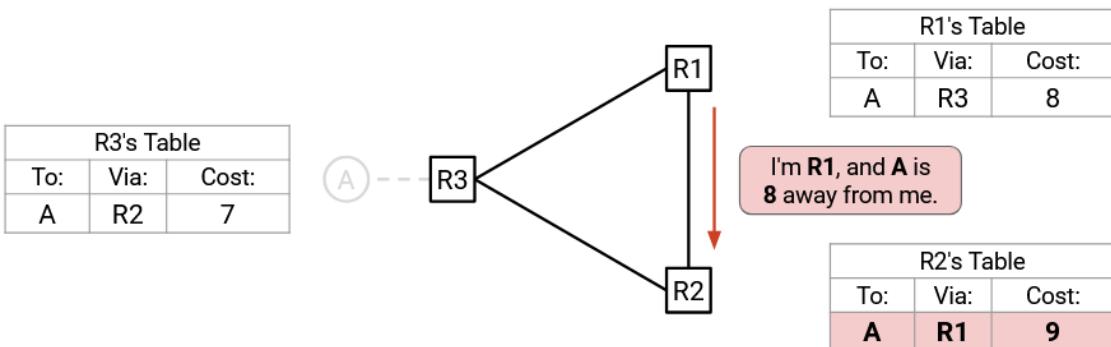
R2 accepts this advertisement (Rule 2), and thinks it can reach A with cost 6.



R2 advertises a cost of 6 to R3, who now thinks it can reach A with cost 7.



R3 advertises a cost of 7 to R1, who now thinks it can reach A with a cost of 8.



R1, R2, and R3 will keep sending advertisements to each other in a cycle, with progressively higher costs (which will all be accepted by Rule 2). Also, packets for A will get stuck in a forwarding loop between these routers.

Let's restate the problem again. The poison didn't correctly propagate to all hosts, so one of the routers still had a busted path in its table. Then, that busted path got advertised in a loop, and Rule 2 caused the costs to keep increasing, with no end in sight.

Why didn't split horizon rescue us? Remember, split horizon only stops a router from advertising back to its next-hop. But in this case, the loop is of length 3, and we were never advertising back to the next-hop.

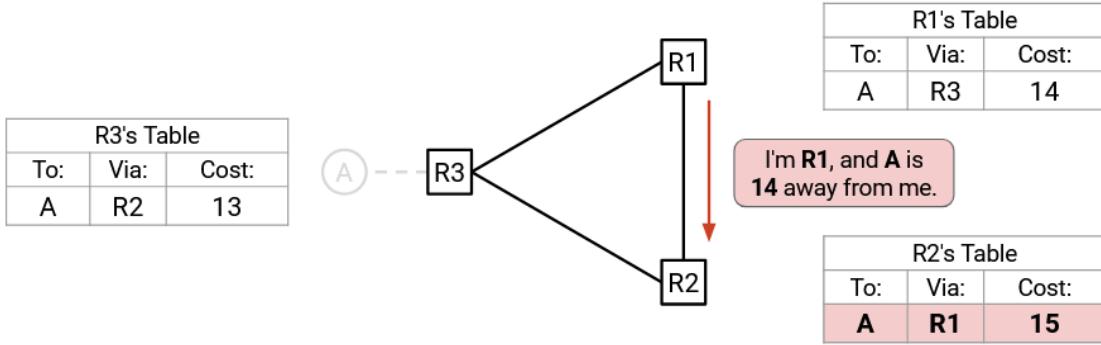
(Note: Poison reverse wouldn't rescue us either. If R3 advertises poison back to R2, then R2 would ignore that poison, because R2's next hop is R1, not R3.)

This is called the **count-to-infinity** problem, and none of our fixes so far (poison expired routes, split horizon, poison reverse) can solve it.

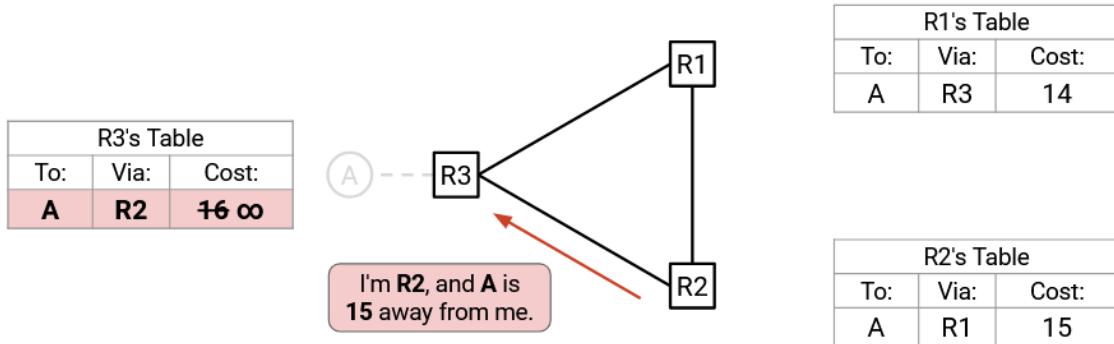
To solve this problem, we will enforce a maximum cost. In RIP, this value is 15. All costs greater than this maximum (i.e. 16 or above) are considered infinity.

With this fix, the loop will still exist for some time, but eventually, all the costs will reach 16 (infinity). Let's watch this in action.

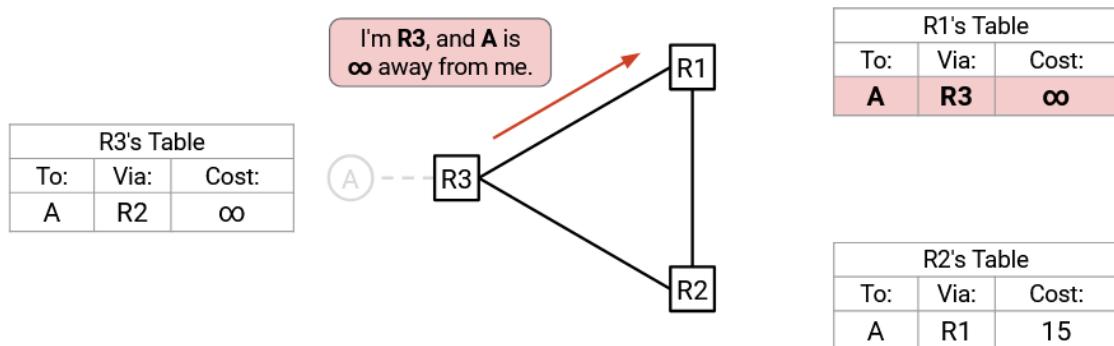
The costs are increasing with every advertisement. Eventually, R1 advertises to R2: "I'm R1, and A is 14 away from me." R2 accepts (per Rule 2) and updates its cost to 15.



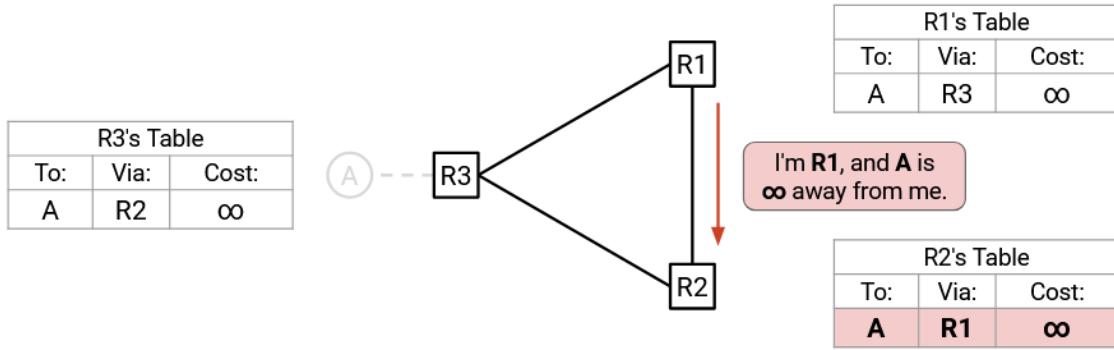
R2 advertises to R3: “I’m R2, and A is 15 away from me.” R3 accepts (per Rule 2), but instead of updating its cost to 16, the cost is updated to infinity.



Next, R3 advertises to R1: “I’m R3, and A is infinity away from me.” R1 accepts (per Rule 2), and now R1 also has a cost of infinity. (Note: This advertisement looks just like poison, though the infinity originated from counting to infinity instead of detecting a failure.)



Finally, R1 advertises to R2: “I’m R1, and A is infinity away from me.” R2 accepts (per Rule 2), and now all the routers have a cost of infinity.



We've reached steady-state again! Any future advertisements would all be advertising infinite cost, and they won't change the tables. Eventually, the infinite-cost entries would all expire. Or, if another route to A appears, it would replace the infinite-cost entry.

Let's review our protocol so far.

For each destination:

- If you hear an advertisement for that destination, update the table **and reset the TTL** if:
 - The destination isn't in the table.
 - The advertised cost, plus the link cost to the neighbor, is better than the best-known cost.
 - The advertisement is from the current next-hop. Includes poison advertisements.
- Advertise to all your neighbors when the table updates, and periodically (advertisement interval).
 - But don't advertise back to the next-hop.
 - ...Or, advertise poison back to the next-hop.
 - **Any cost greater than or equal to 16 is advertised as infinity.**
- If a table entry expires, make the entry poison and advertise it.

Eventful Updates

There are three occasions where a router might want to send advertisements:

1. Send advertisements when the table changes. These are called **triggered updates**. The table might change when we accept a new advertisement, or when a new link is added (e.g. new static route), or when a link goes down (e.g. route gets poisoned).
2. Send advertisements periodically, once every advertisement interval.
3. Send advertisements when a table entry expires (and gets replaced by poison).

Note that triggered updates are an optimization. Instead of advertising every time the table changes, we could just wait for the next advertisement interval to advertise the changes. This protocol would still be

correct. However, triggered updates, in addition to the periodic updates, help our protocol converge on correct routes faster, because we propagate new information the instant we learn about it.

Link-State Protocols

Introduction to Link-State Protocols

Recall that there are different classes of routing protocols, depending on their underlying algorithm. In the previous section, we saw the distance-vector class of protocols. In this section, we'll discuss **link-state**, another major class of protocols.

Recall that protocols can also be classified as exterior gateway protocols (operating between networks) and interior gateway protocols (operating within networks). Like distance-vector, link-state protocols are usually interior gateway protocols.

IS-IS (Intermediate System to Intermediate System) and OSPF (Open Shortest Path First) are two major examples of link-state protocols. Both are widely deployed today.

Link-State Overview

Distance-vector performed a distributed, cooperative computation. Each node computes its own piece of the solution, based on results computed by its neighbors. The computation across all nodes collectively forms the full solution. Each node only needs local information from its neighbors in the computation (nodes don't know the full network graph).

By contrast, link-state protocols perform a local computation. Each node computes the full solution independently and from scratch, without using any computation results from neighbors. However, to do this, each node needs global information from all parts of the network.

Link-state protocols in one sentence: Every router learns the full network graph, and then runs shortest-paths on the graph to populate the forwarding table.

There are two major steps that we have to implement. First, the router needs to somehow learn the full network graph, including the state of every link (up or down), the cost of every link, and the location of every destination.

Then, the router needs to run some algorithm on that graph to learn how to forward packets to every destination.

We'll think about the second step first (shortest paths), then think about the first step (learning the graph).

Computing Paths

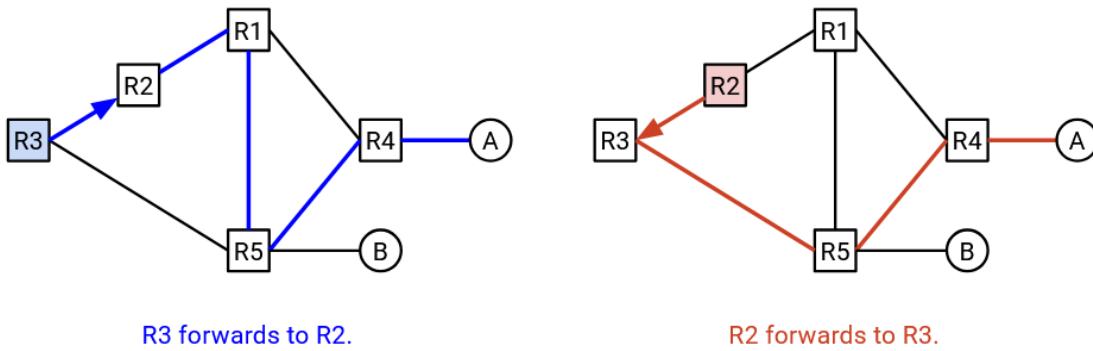
Once the router has a global view of the network, it can easily compute paths through the network using some shortest-path algorithm.

In particular, the router should compute the shortest path to every destination. Then, for each destination, the router records the next hop along the shortest path, just like in distance-vector protocols. The rest of the path is not needed during forwarding.

Many single-source shortest-path algorithms can be used in this step. For example, the Bellman-Ford

algorithm (serial version, with none of the distance-vector changes) and Dijkstra's algorithm both efficiently compute the shortest path from a single source to all destinations. We could also consider alternate solutions like breadth-first search, or algorithms that can run in parallel.

One thing we have to be careful about is inconsistencies between routers.



Remember, every router is computing the shortest paths independently, and deciding on a next hop accordingly. Each router only controls its own next hop, and cannot influence what the next hop will do.

For example, suppose R5 computes this shortest path to A, and decides to forward packets to R2. Then, R2 computes this shortest path to A, and decides to forward packets to R5. Both routers computed valid shortest paths, but their decisions resulted in a routing loop.

To avoid this problem, we have to make sure that all routers are producing forwarding decisions that are compatible with each other. What are the requirements for all routers to produce compatible decisions?

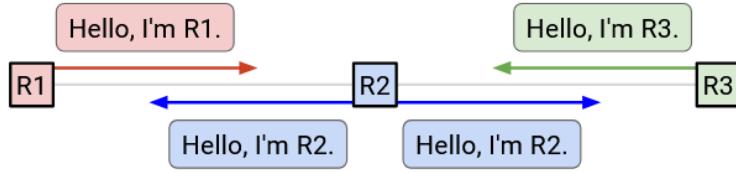
1. All routers have to agree on the network topology. Suppose a link failed, but only one router knows about it. Then different routers are computing paths on totally different graphs, and might produce inconsistent results.
2. All routers are finding least-cost paths through the path. If one router preferred more expensive paths for some reason, we would get inconsistent results.
3. All costs are positive. Negative costs could produce negative-weight cycles.
4. All routers use the same tiebreaking rules. If we assumed shortest paths are unique, then the previous two conditions are sufficient to ensure everybody picks the same path. This condition additionally ensures that if there are multiple paths tied as the shortest, everyone chooses the same one.

With these four conditions, routers could use different shortest-path algorithms, and they would still all compute the same paths and produce compatible decisions. In practice, though, routers usually all use the same algorithm for simplicity.

Learning About Graph Topology

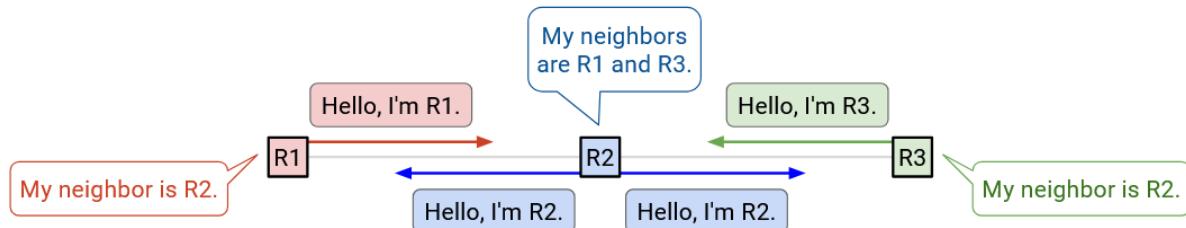
How do routers learn about the full network graph? First, we need to learn who our neighbors are (both routers and destinations). Then, we need to distribute that information through the whole network. We also need routers to glue together all the information it receives into a graph topology.

To discover neighbors, every router sends a hello message to all of its neighbors.



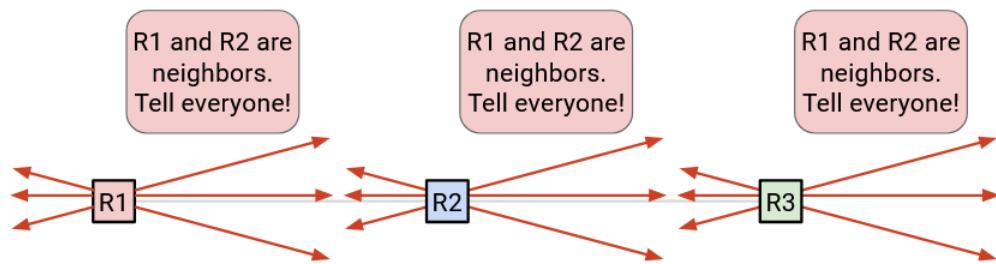
For example, in this network, E sends to both of its neighbors: “Hello, I’m E.” Now, B knows that it’s connected to E, and C also knows that it’s connected to E. Similarly, B says hello to E, so now E knows about B. Likewise, C says hello to E, so E also knows about C.

As a result, everybody now knows who their immediate neighbors are. Note that B does not know about C, because B and C are not neighbors.



We also want to know if links go down. To support this, we’ll periodically re-send the hello message. If a neighbor stops saying hello (e.g. misses some number of hellos), we assume they disappeared.

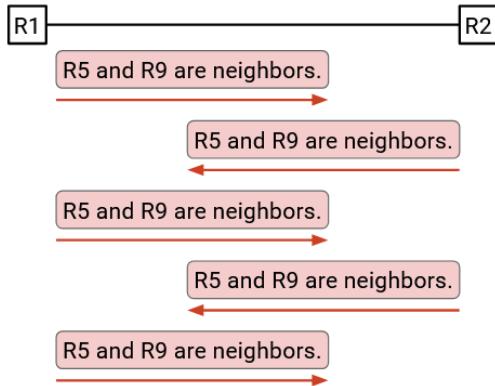
Now that we know about our neighbors, we should announce that fact to everybody. To make a global announcement, we send the announcement to all of our neighbors. Also, if we ever receive an announcement, we should send it to all of our neighbors as well. This ensures that every message gets propagated throughout the network. This is known as **flooding** information across the network. If any information changes (e.g. a neighbor disappears), we should flood that information as well.



We also need to make sure that messages don’t get dropped. Otherwise, other routers might miss an update and compute paths on the wrong graph. To fix this problem, we use the same trick as we used in distance-vector, and periodically re-send the message. As long as the link is functioning, our message should get sent after enough tries.

Avoiding Infinite Flooding

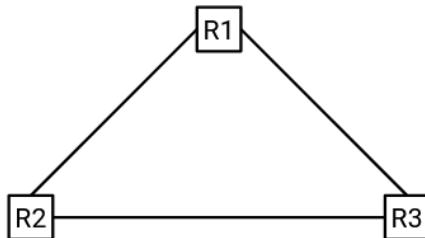
We have to be careful about how we flood announcements through the network.



R2 learns some information and announces it to its neighbor R3. When R3 receives this information, it makes an announcement to its neighbor R2. When R2 receives this information, it makes an announcement to its neighbor R3. These two routers are stuck making announcements to each other, wasting bandwidth, even though there's no new information.

Note that this is not the same as periodically re-sending messages for reliability. For reliability, we might re-send a message once every 5 seconds. In this infinite loop, the routers are receiving and re-sending duplicate announcements at maximum rate (e.g. millions of times per second).

The problem is even worse if our network contains a loop:



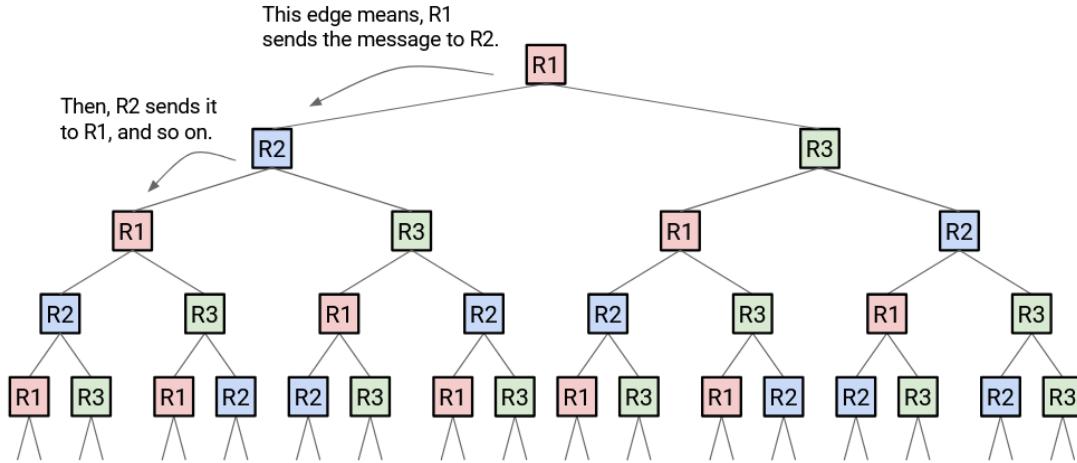
Time step 1: R1 broadcasts to R2 and R3.

Time step 2: R2 broadcasts to R1 and R3. R3 broadcasts to R1 and R2.

Time step 3: R1, R1, R2, and R3 all make broadcasts to (R2, R3), (R2, R3), (R1, R3), and (R1, R2) respectively. Note that R1 received two messages at time step 2, so it makes two broadcasts.

Time step 4: R1, R1, R2, R2, R2, R3, R3, R3 all make broadcasts to (R2, R3), (R2, R3), (R1, R3), (R1, R3), (R1, R3), (R1, R2), (R1, R2), (R1, R2), respectively.

Time step 5: R1 makes 6 broadcasts, R2 makes 5 broadcasts, R3 makes 5 broadcasts.



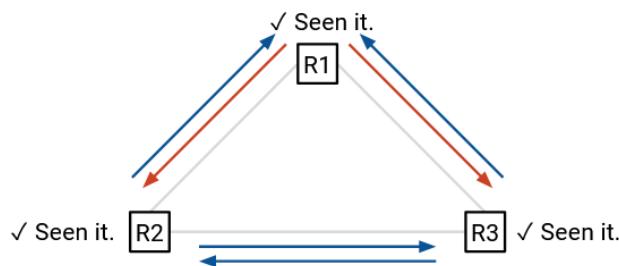
All the new information was learned at time step 1. But, everybody keeps re-sending the same information, and duplicate announcements multiply exponentially and eventually overwhelm the network.

To fix this problem, we need to make sure that routers don't send the same information twice.

When we see a message for the first time, send that message to all neighbors, and write down that we've seen that message. (We have to write down this message anyway, since we're trying to use this information to build up the network graph.) Then, if we ever see that same message again, don't send it a second time.

To uniquely identify a message, we can introduce a timestamp (or some other counter that's unique to every message).

Now, if we go back to the example from earlier:



Time step 1: R1 broadcasts to R2 and R3.

Time step 2: R2 broadcasts to R1 and R3. R3 broadcasts to R1 and R2.

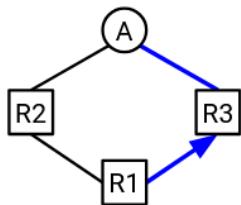
Time step 3: At this point, R1, R2, and R3 have all seen the message before, so they don't send it again. No further duplicate messages are sent.

Note that duplicate messages are still sometimes sent with this modification, but we've avoided duplicate messages being sent infinitely.

Convergence

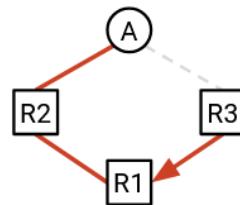
Link-state converges on a valid least-cost routing state after every router learns the full network topology and computes its forwarding table accordingly. Convergence relies on every node using the same graph. After convergence, the routing state remains valid as long as the network topology doesn't change.

As soon as the network topology changes, it can take some time for the network to converge again. We have to wait for the change to be detected (e.g. a link failure). Then, we have to wait for the new information to be propagated through the network, and for routers to re-compute forwarding table entries. While the network is converging, we might be in an invalid routing state, because some routers are using the old graph, while others are using the updated graph. The routing state could have dead-ends, loops, or paths that are not least-cost.



Link is down, but R1 doesn't know!

R1 forwards to R3.



R3 knows about the link failure!

R3 forwards to R1.

For example, suppose the R3-A link has failed. R3 knows about this, but the other routers do not. R3 will forward packets to R1. However, R1 will still forward packets to R3.

Much of the complexity in link-state protocols is in the small details. To ensure faster convergence and avoid invalid routing as much as possible, we can make minor optimizations and adjustments in the protocol.

Link-State vs. Distance-Vector

What are some pros and cons of link-state protocols compared to distance-vector protocols?

In distance-vector, when we receive an announcement, we don't necessarily know all the details about the path we're accepting. We have to trust whatever our neighbor claims in the announcement. By contrast, in link-state, we know the full topology of the graph, so we know more about the paths that packets are taking.

Depending on implementation, distance-vector could be slower to converge. If the network changes, we have to wait for our neighbor to recompute and readvertise a path, before we can update our forwarding table. Then, all of our neighbors have to wait for us, and so on. By contrast, in link-state, everybody can quickly flood the new information and recompute at the same time.

Link-state protocols are good for small local networks, but don't scale well to the global Internet. In particular, link-state requires every router to know about the entire network. On the global Internet, operators might not want to reveal their network topology (e.g. where their routers are located, the bandwidth of their links) to competitors.

In practice, most networks deploy a combination of distance-vector and link-state protocols.

Addressing

Scaling Routing

So far, our forwarding table has one entry per destination. This won't scale to the entire Internet.

If we ran distance-vector on the entire Internet, we'd have to send an announcement for every host on the Internet. If we ran link-state on the entire Internet, every router would have to know the full Internet network graph. In both cases, if any host joins or leaves the Internet, we'd have to re-do computations to converge on a new routing state.

The trick to scale routing is how we address hosts. So far, we've called every host and router by some name (e.g. R1, R2, A, B), but in practice, we'll use a smarter addressing scheme.

R3's Table	
Destination	Port
A	0
B	1
C	1
D	2
...	...

The trick: Use more informative names than A, B, C, D for destinations.

One entry for every possible destination.

IP Addressing

Recall that in our postal service analogy, we had different addressing schemes for different contexts. The mailman used a street address like 2551 Hearst Ave. The secretary inside the building used a room number like 413 Soda Hall. Addresses are assigned in some structured way. For example, all third-floor room number start with the digit 3, and all fourth-floor rooms number start with the digit 4.

Just like the postal system, the Internet uses different addressing schemes at each layer. In this section, we'll focus on IP addresses, which can be used for routing at Layer 3.

Every host on the network (e.g. your computer, Google's server) is assigned an IP address. For this section, you can assume every host has a unique IP address.

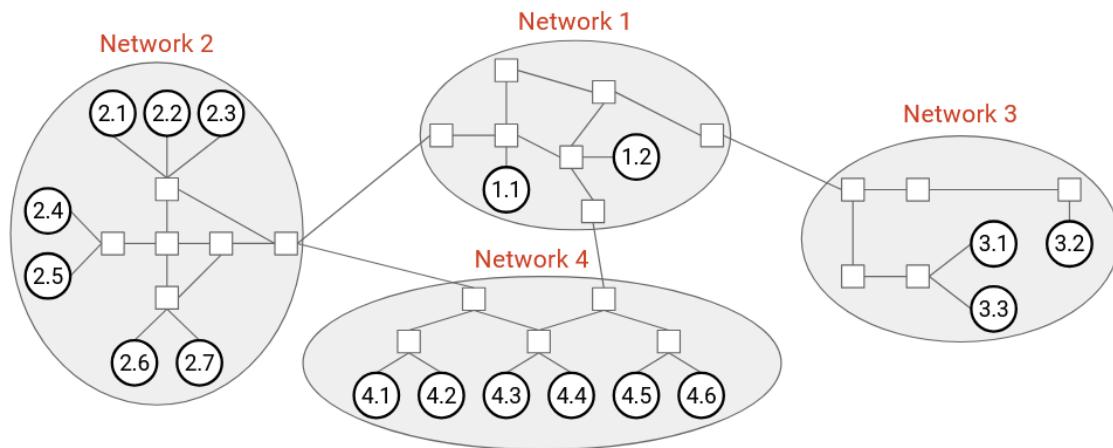
An **IP address** is a number that uniquely identifies a host. Just like the postal system, the number is chosen to contain some context about where the host is located.

Note that IP addresses are not necessarily static. In the analogy, if you move to a different house, your address changes. Similarly, if your computer moves to a different location, it may be assigned a different IP address when it joins the network (and your old IP address will eventually expire).

The length of an IP address depends on the version of IP being used. IPv4 addresses are 32 bits, and IPv6 addresses are 128 bits. The routing concepts are similar for both versions, but we'll use IPv4 when possible, because smaller addresses are easier to read.

Hierarchical Addressing

Recall that the Internet is a network of networks. There are many local networks, and we add links between local networks to form the wider Internet. This gives us a natural hierarchy that we can use to organize our addressing scheme.

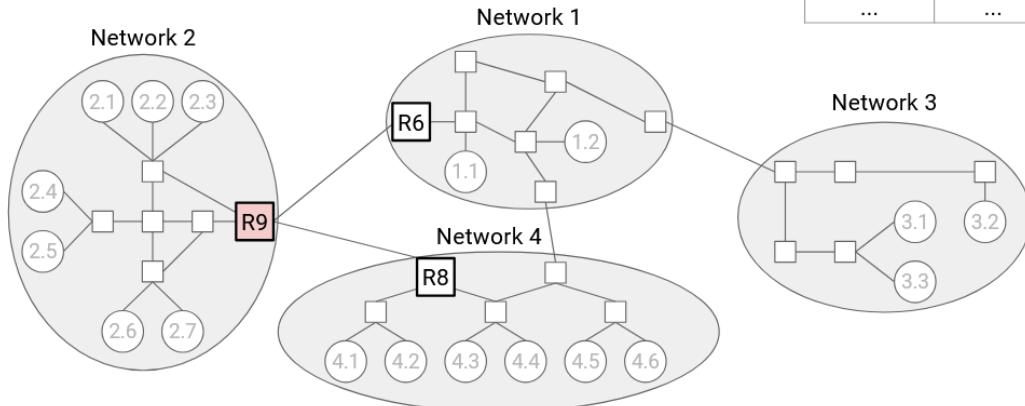


Here's an intuitive picture of addressing. We could assign a number to every network. Then, within network 3, we could assign host numbers 3.1, 3.2, 3.3, etc., and similar for hosts in the other networks.

R9 can summarize all hosts in another network with a single table entry.

Huge scaling improvement! Tables are smaller now.

R9's Table	
Destination	Next Hop
1.*	R6
3.*	R6
4.*	R8
...	...

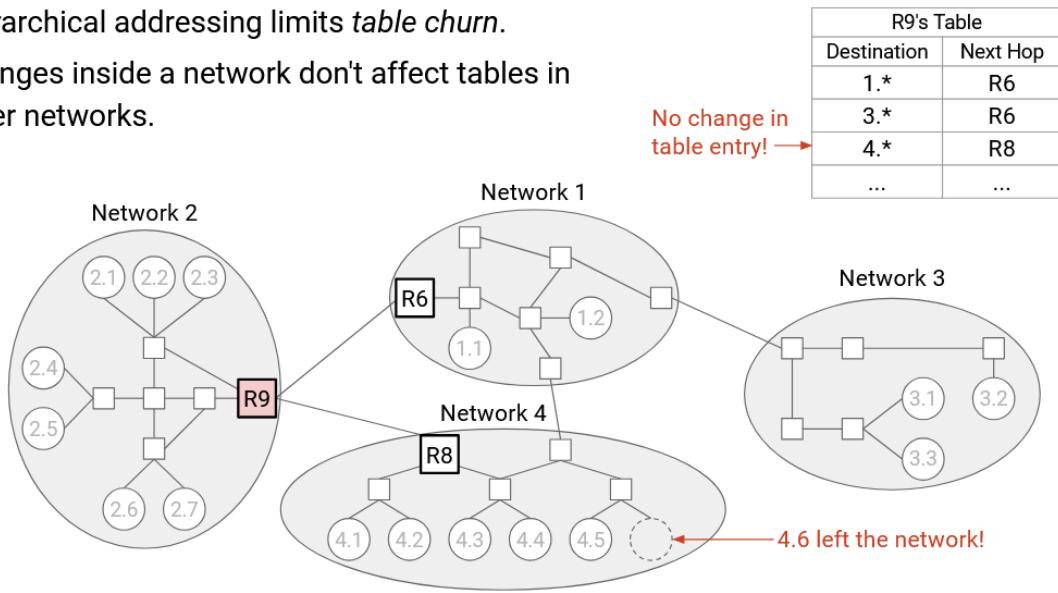


Now, consider the forwarding table in router R9. Before, we would have one entry for every host in network 1, and they would all have the same next hop of R6. With our hierarchical addressing, we could instead have a single entry for the entire local network, saying that all 1.* addresses (where the * represents any number) have a next hop of R6. We could also say that all 2.* addresses have a next hop of R8.

This hierarchical model, where we use wildcard matches to summarize routes, makes our forwarding tables smaller.

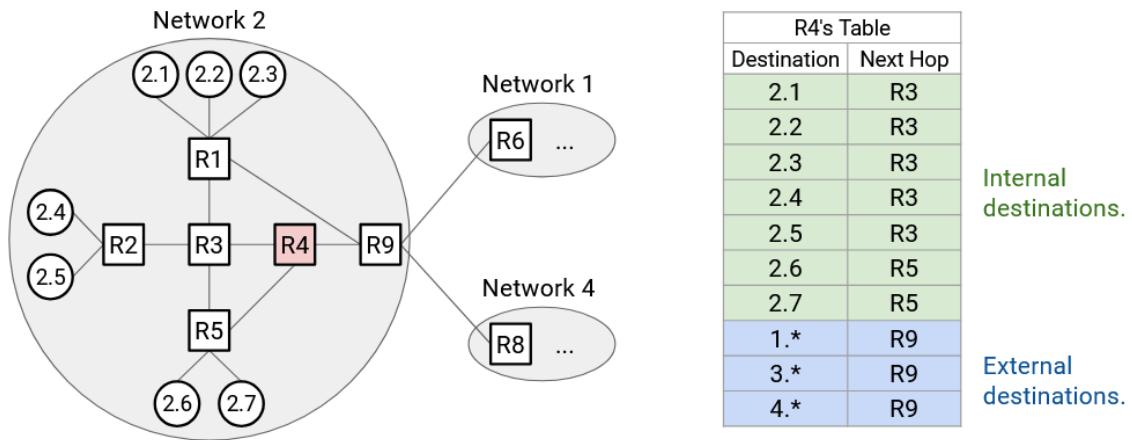
In addition, this model also makes our tables more stable.

Hierarchical addressing limits *table churn*.
Changes inside a network don't affect tables in other networks.



If the topology inside network 1 changes, we don't need to update R9's forwarding table (or any other tables in other networks). In practice, changes within a local network (e.g. new host joins the network) happens much more often than changes between networks (e.g. new underground cable installed), so it's a good thing that local changes only affect local tables.

More generally, our addresses have two parts: a network ID, and a host ID. This allows inter-domain routing protocols to focus on the network ID to find routes between networks, and intra-domain routing protocols to focus on the host ID to find routes inside networks. This also makes our routing protocols more stable as the network changes. Inter-domain protocols don't care about changes inside networks, and intra-domain protocols don't care about changes in other networks.

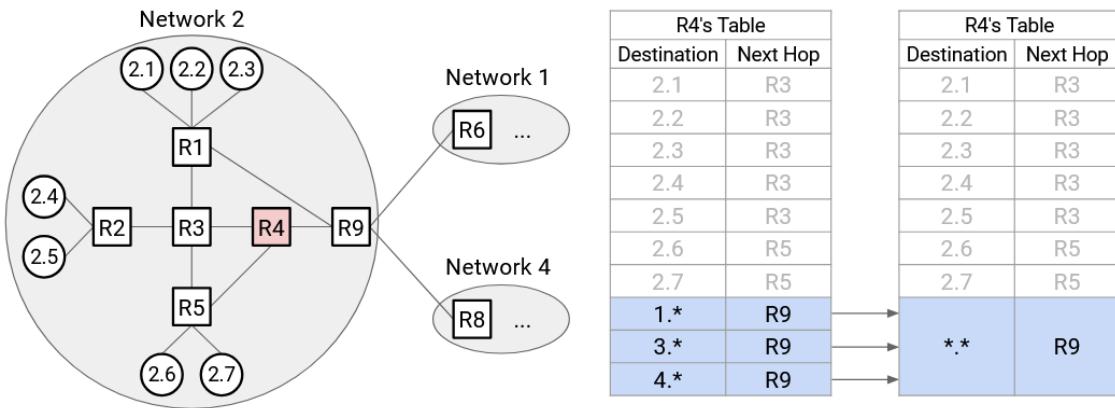


Note that the forwarding table in R9 still needs entries for each individual host inside its own network (i.e. network 3).

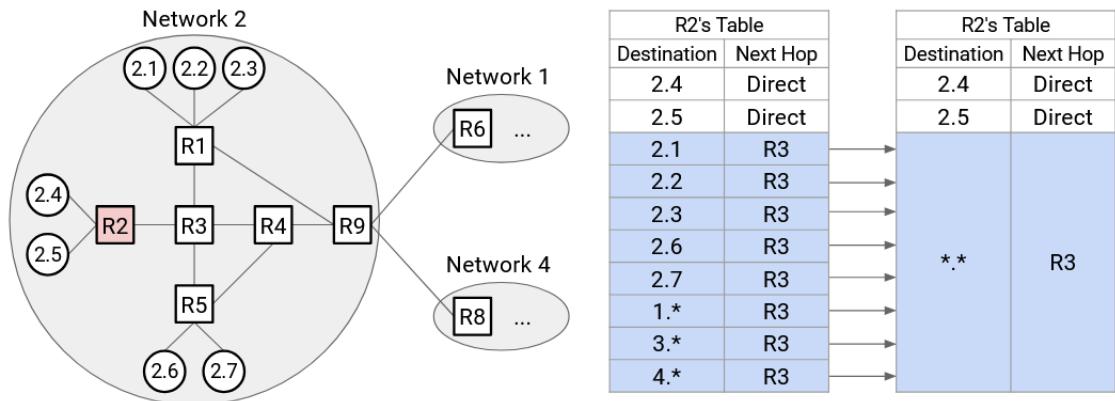
Similarly, R4, an internal router with no connections to other networks, needs both entries for individual hosts inside network 3, and aggregated entries for other networks (e.g. 2.* has a next hop of R9). The scale of a forwarding table depends on the number of internal hosts in the same network, plus the number of external networks.

Default Routes

We now know that our entries can represent entire ranges of addresses, instead of always representing a single address. We can extend this idea even further to improve scale.



Consider R4. It has an entry for every external network (1.*, 2.*, etc.), all with the same next hop of R3. We could aggregate every external network into a single entry. We'll still have entries for every internal host (3.1, 3.2, etc.), but at the end, we'll say: For all other hosts not in the forwarding table, the next hop is R9.



We can use more aggressive aggregation at R2. Again, all external networks have a next hop of R3. But, 3.1, 3.2, 3.3, 3.6, and 3.7 also have a next hop of R3. Therefore, the forwarding table only needs static entries for

3.4 and 3.5. Then, we can say, for all other hosts not in the forwarding table (including some internal and some external hosts), the next hop is R3.

To represent all hosts not in the table, we can use a wildcard ** that matches everything. When forwarding toward a given destination, the router first checks specific hosts (e.g. 3.1) or ranges (e.g. 2 *) for matches. If the router can't find any matches, it will eventually match the ** wildcard. This is called the **default route**.

Most hosts only have a single hard-coded default route. For example, host 2.4's forwarding table has a single entry, saying to send everything to R2. In practice, your home computer has a single entry, saying to send everything to your home router. This is why hosts don't need to participate in routing protocols.

Assigning Hierarchical IP Addresses: Early Internet

In order to get more scalable routing, we need to assign addresses in some hierarchical way. The addresses need to contain some information about their location (e.g. nearby hosts need to share some part of their address).

In the early Internet, IPv4 addresses had an 8-bit network ID and a 24-bit host ID, just like in our intuitive version.



For example, AT&T has network ID 12, Apple has network ID 17, and the US Department of Defense has 13 different network IDs.

The 8-bit network ID means we can only assign 256 different network IDs, but in real life, there are way more than 256 organizations that might operate their own local network. Also, our 24-bit host ID means that every network gets $2^{24} = 16,777,216$ addresses. A small network (e.g. a company with 10 employees) probably doesn't need 16 million addresses. As the Internet grew larger, a new approach to addressing was needed.

Assigning Hierarchical IP Addresses: Classful Addressing

The first attempt to fix this was **classful addressing**, which allocates different network sizes based on need. In this approach, there are 3 classes of addresses, each with a different number of bits allocated to the network ID and host ID. The first 1-3 bits identify which class is being used.

Class A:		Network (7 bits)	Host (24 bits)
		~128 networks	~16m hosts per network
Class B:		Network (14 bits)	Host (16 bits)
		~16k networks	~65k hosts per network
Class C:		Network (21 bits)	Host (8 bits)
		~2m networks	~256 hosts per network

Class A addresses start with leading bit 0. The next 7 bits are the network ID (128 networks), and the next 24 bits are the host ID (16 million hosts).

Class B addresses start with leading bits 10. The next 14 bits are the network ID (16,000 networks), and the next 16 bits are the host ID (65,000 hosts).

Class C addresses start with leading bits 110. The next 21 bits are the network ID (2 million networks), and the next 8 bits are the host ID (256 hosts).

In this approach, we can now have 2 million + 16,000 + 128 different local networks. Larger organizations with more hosts could receive a Class A network, and smaller organizations could receive a Class B or Class C network. As before, within a single network, the leading class bit(s) and network ID bits are the same, and each host gets a different host ID.

One major problem with classful addressing is the size of each class. Class A (16 million hosts) is way too big for most organizations, and Class C (256 hosts) is way too small for most organizations. As a result, most networks need to be in Class B.

Unfortunately, there are only 16,000 Class B network IDs, and by 1994, we were running out of Class B networks. Again, a new approach to addressing was needed.

Note: Classful addressing is now obsolete on the modern Internet.

Note: Technically, the number of hosts per network is off by 2, because the all-zeroes address and all-ones address are reserved for special purposes. For example, in Class C, there are actually 254 hosts per network, not 256.

Assigning Hierarchical IP Addresses: CIDR

Our third approach to hierarchical addressing, and the one still used on the modern Internet, is **CIDR** (Classless Inter-Domain Routing). In CIDR, we still have variable-length network IDs, but instead of only 3 different network ID lengths (Class A, B, C), we make the number of fixed bits arbitrary.

For example, consider the tiny company with 10 employees from earlier. In classful addressing, they would get a Class C network with 256 host addresses. If they only need 10 host addresses, we could allocate fewer addresses by giving them a longer network ID.

If we allocated a 28-bit network ID, the host ID would be 4 bits long (16 possible addresses). If we allocated a 29-bit network ID, the host ID would be 3 bits long (8 possible addresses). We can't allocate exactly 10 addresses, but a 28-bit network ID would be sufficient for this company's purposes. There's a little bit of waste (6 unused addresses), but this is still way better than allocating 256 addresses.

As another example, consider an organization that needs 450 host addresses. In classful addressing, Class C (256 addresses) isn't sufficient, so they would receive a Class B network with 65,000 host addresses, and most of the addresses would go unused. With arbitrary-length network IDs, we can assign a 23-bit network ID, which gives 9 bits for host addressing (512 addresses). This meets the organization's needs and wastes far fewer addresses.

Multi-Layered Hierarchical Assignment

In real life, hierarchies can be multi-layered. For example, inside a network, an organization can choose to assign specific ranges of addresses to specific sub-organizations (e.g. departments in a company or university).

In practice, we exploit real-life multi-layered organizational and geographical hierarchies to assign addresses. ICANN (Internet Corporation for Names and Numbers) is the global organization that owns all the IP addresses.

ICANN gives out blocks of addresses to Regional Internet Registries (RIRs) representing specific countries or continents. For example, RIPE gets all addresses for the European Union, ARIN gets North American addresses, APNIC gets Asia/Pacific addresses, LACNIC gets South American addresses, and AFRINIC gets African addresses. Example: ICANN gives ARIN all addresses starting with 1101.

Each RIR then gives out portions of their ranges to large organizations (e.g. companies, universities) or ISPs. These organizations or ISPs are called Local Internet Registries. Example: ARIN controls all addresses starting with 1101, and gives AT&T all addresses starting with 1101 11001.

Finally, each local Internet registry assigns individual IPs to specific hosts. For additional hierarchy, local registries can also assign IP ranges to small organizations, and the small organizations can in turn assign individual IPs.

ICANN owns all addresses:
ARIN (North America) owns:	1101 4 bits fixed, $2^{28} \approx 268m$ addresses.
AT&T (large ISP) owns:	110111001 9 bits fixed, $2^{23} \approx 8m$ addresses.
UC Berkeley owns:	110111001110100010 18 bits fixed, $2^{14} \approx 16k$ addresses.
Soda Hall owns:	110111001110100010011010 24 bits fixed, $2^8 \approx 256$ addresses.
Prof. Ratnasamy owns:	11011100111010001001101001011101 All bits fixed, 1 address.

At each level, the number of additional bits fixed is determined by the number of addresses to be allocated. For example, ARIN might want to give AT&T 8 million addresses, and computes that fixing 9 bits results in 8 million host addresses. ARIN had 4 bits fixed already, so it fixes another 5 bits and assigns AT&T all addresses starting with those 9 bits. AT&T might then give the prefix 1101 11001 110100010 to give 16,000 addresses to UC Berkeley. As we allocate addresses to sub-organizations, more bits are fixed, always keeping the fixed bits from parent organizations.

Writing IP Addresses

We could write IP addresses as a 32-bit sequence of 1s and 0s, or as a single big integer. In practice, for readability, we take each sequence of 8 bits and write it as an integer (between 0 and 255). For example, the IP address 00010001 00100010 10011110 00000101 can be written as 17.34.158.5. This is sometimes called a **dotted quad** representation.

So far, we've been writing ranges of addresses as bits (e.g. all IPs starting with 1101). To write a range of addresses, we can use **slash notation**. We write the fixed prefix, then we write 0s for all remaining unfixed bits, and we convert the resulting 32-bit value into an dotted quad IP address. Then, after the slash, we write the number of fixed bits.

For example, if the prefix is 11000000, we add zeros for all the unfixed bits to get 11000000 00000000 00000000 00000000. As a 32-bit address, this is 192.0.0.0. Then, because 24 bits were fixed, we write the range as 192.0.0.0/24.

To write an individual address as a range, we could write something like 192.168.1.1/32, which indicates that all 32 bits are fixed. Also, the default route ** can be written as 0.0.0.0/0.

Slash notation can sometimes look a little confusing because we're using arbitrary 8-bit divisions and writing numbers in decimal. For example, the 8-bit prefix 11000000 and the 12-bit prefix 11000000 0000 would be written as 192.0.0.0/24 and 192.0.0.0/20 (same IP address representing different ranges). As another example, if I owned the 4-bit prefix 1100, I could assign the 5-bit prefix 11001. As ranges, these are written as 192.0.0.0/24 and 200.0.0.0/24. At first glance, it's not clear that the second range is actually a subset of the first one, and we'd have to write out the bits to confirm.

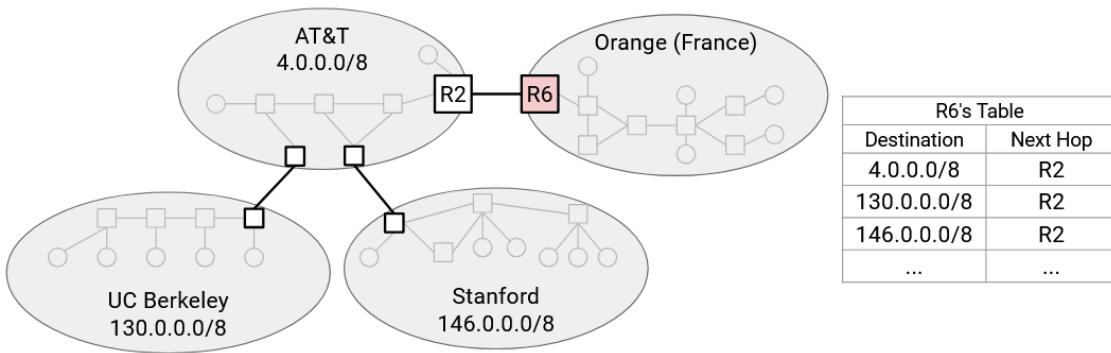
An alternative to the slash (e.g. /16) in the slash notation is a **netmask**. Just like the number after the slash, the netmask tells us how which bits are fixed. To write a netmask, we write 1s for all fixed bits and 0 for all unfixed bits, and convert the result into a dotted quad. For example, if we had the range 192.168.1.0/29, we could write 29 ones (fixed bits) and 3 zeros (unfixed bits). 11111111 11111111 11111111 11111000 as a dotted quad is 255.255.255.248. The range in netmask notation is 192.168.1.0, with netmask 255.255.255.248 (replaced the slash with a netmask).

In these notes, we'll usually use slash notations because they're more convenient to read. In practice, netmasks can be useful because given a specific IP address, if you perform a bitwise AND between the IP address and the netmask, all the host bits will get zeroed out, and only the network bits will remain.

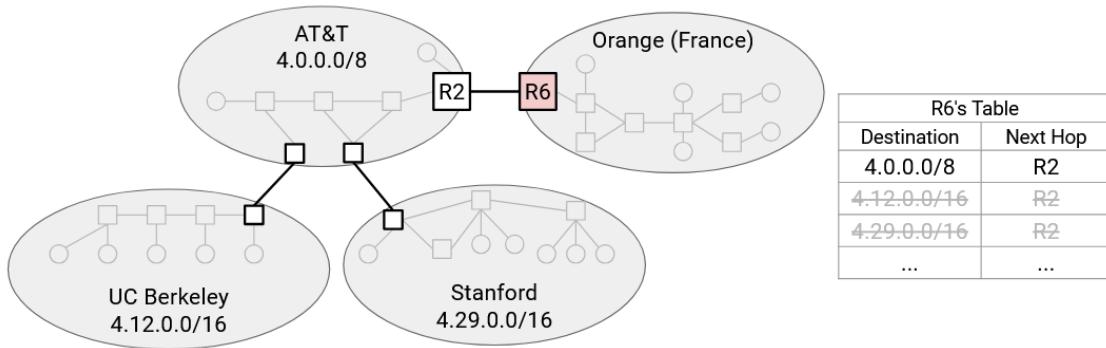
Aggregating Routes with CIDR

In our original model with a network ID and host ID, we could aggregate all hosts inside the same network into a single route in the forwarding table (e.g. 2.* for everything in network 2).

Multi-layered hierarchical addressing means that we can also aggregate multiple networks into a single route.



Consider this diagram of networks. In our original model, R6 needs a separate forwarding entry for AT&T, UCB, and Stanford.

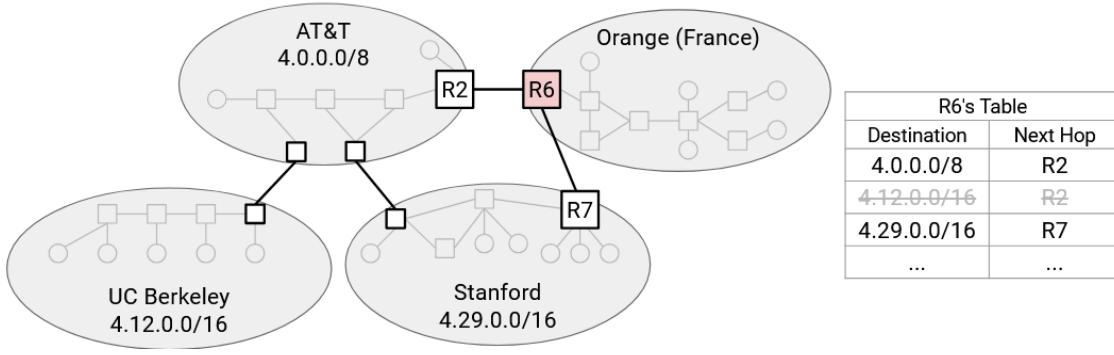


However, if we used hierarchical addressing, then UCB's range (4.12.0.0/16) and UCM's range (4.29.0.0/16) are both subsets of AT&T's range (4.0.0.0/8). This could happen if AT&T allocated those ranges to its subordinate customers UCB and UCM.

Now, R6 only needs a single entry for AT&T, UCB, and UCM. We've aggregated the two smaller ranges into the wider range that they both belong to.

Multi-Homing

Aggregating ranges doesn't always work. Suppose we added a link from R6 directly to Stanford.



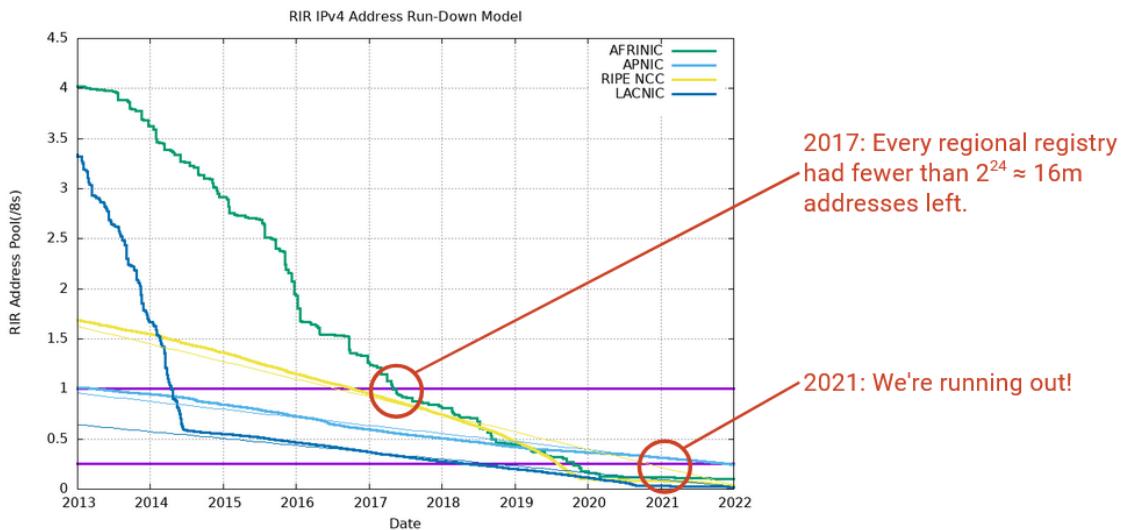
Our aggregated route says that all packets to AT&T (and its subordinates) have a next hop of R2. We need to add an additional entry saying that Stanford has a next hop of R7.

Notice that our forwarding table now has ranges that overlap. A destination could match multiple ranges. To pick a route, we'll run **longest prefix matching**, which means we'll use the most specific range that matches our destination IP address. For example, if we had a packet destined for a UCM host, we would use the UCM-specific entry because it has the longer 19-bit prefix. Even though the 9-bit AT&T entry also matches the destination, its prefix is shorter, so we don't use this route.

If instead we had a packet destined for a UCB host, we can't use the Stanford-specific entry, because the 16-bit prefix won't match the UCB host. But we can still use the 8-bit AT&T entry, which will match the destination.

Brief History of IPv6

IPv4 addresses are 32 bits, which means we have roughly 4 billion addresses available. Is this enough?



This graph plots the number of remaining unallocated IP addresses (y-axis) for each regional registry over time.

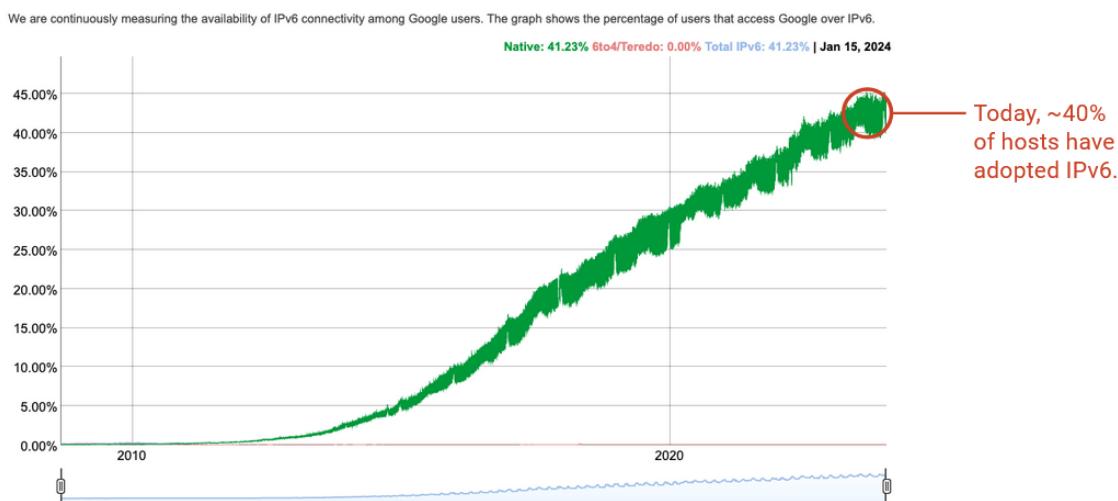
By 2017, everybody had less than one /8 block (i.e. less than $2^{24} = 16$ million addresses) available. Each regional registry held a spare /8 block of addresses just in case, but by 2017, everybody had to start using their spare supply of addresses. By 2021, even the spare supply of addresses was running out.

Fun fact: In February 2011, there was an in-person ceremony when the final /8 block was allocated. There was even a special paper certificate issued.

As the Internet grew, we started to realize that we would eventually run out of addresses. Luckily, this was realized early on, and IPv6 was developed in 1998 as a response to IP address exhaustion.

Fundamentally, IPv6 addressing structure is the same as IPv4. There are some minor implementation changes needed for IPv6, though they aren't relevant here.

The main new feature in IPv6 is longer addresses. IPv6 addresses are 128 bits long, which means there are roughly 3.4×10^{38} possible addresses. This is an astronomically big number, so we'll never run out. The universe is 10^{21} seconds old, so we could assign an address to every second and still have only used 0.0000000000000001% of all available addresses.



IPv6 was developed in the 1990s, but was not immediately adopted by all computers. Even in 2010, basically nobody used IPv6. As of 2024, IPv6 is used by around 45% of end users, and most of these users are located in developed countries with wider Internet adoption. The main reason why IPv6 is becoming more widely-adopted is because we're running out of IPv4 addresses.

Why is IPv6 adoption so slow? Users, servers, and Internet operators have to upgrade their software and hardware (e.g. routers, links, device drivers on computers) to support IPv6. Routers now need two forwarding tables, one with IPv4 addresses and one with IPv6 addresses.

IPv6 upgrades have to be backwards-compatible. If a server only had an IPv6 address, users on older computers that only support IPv4 can't use this server. IPv4 and IPv6 are essentially separate addressing systems, and there's no way to convert between IPv4 and IPv6 addresses. As of 2024, many computers still don't support IPv6, so many services need to support both IPv4 and IPv6.

Computers that do support both IPv4 and IPv6 also have to think about which one to use. Is one better than the other? In practice, IPv6 is faster, but many other implementation details could affect your choice.

IPv6 Address Notation

IPv6 addresses are usually written in hexadecimal instead of decimal. For example:

2001:0D08:CAFE:BEEF:DEAD:1234:5678:9012

is an IPv6 address (32 hex digits = 128 bits). We add colons in between every 4 hex digits (16 bits) for readability.

For readability, we can omit leading zeros within a 4-digit block. For example:

2001:0DB8:0000:0000:0000:0000:0000:0001

can be shortened to 2001:DB8:0:0:0:0:1.

For readability, we can also omit a long string of zeroes, e.g. 2001:DB8::1. The double colon says to fill in all missing 4-digit blocks with 0000. This can only be done once per address. (Omitting two ranges creates ambiguity, because we don't know how many zeroes go in each range.)

Slash notation can still be used in IPv6. An individual address has /128 (all bits fixed). A 32-bit prefix might look like 2001:0DB8::/32.

Because the address space is so large, in IPv6, you could fix the network ID to be 64 bits and the host ID to be 64 bits, and still never run out of network IDs or host IDs. In fact, special protocols exist where networks and hosts can pick their own 64-bit network ID and host ID (and check that no one else is using it), without an organization needing to allocate specific IDs.

In practice, regional registries typically allocate 32-bit prefixes to ISPs, and ISPs typically allocate 48-bit prefixes to organizations. The organization can then allocate 64-bit prefixes to smaller sub-networks. In IPv6, we usually don't see prefixes longer than /64. Even the smallest sub-networks inside an organization have a 64-bit prefix, and 64 bits for addressing specific hosts. Using these standardized prefix sizes allows prefixes to be more informative. For example, in IPv6, it's not clear what a /19 prefix represents, but in IPv6, we know a /32 prefix typically represents an ISP. TODO double check this

Router Hardware

What Do Routers Do?

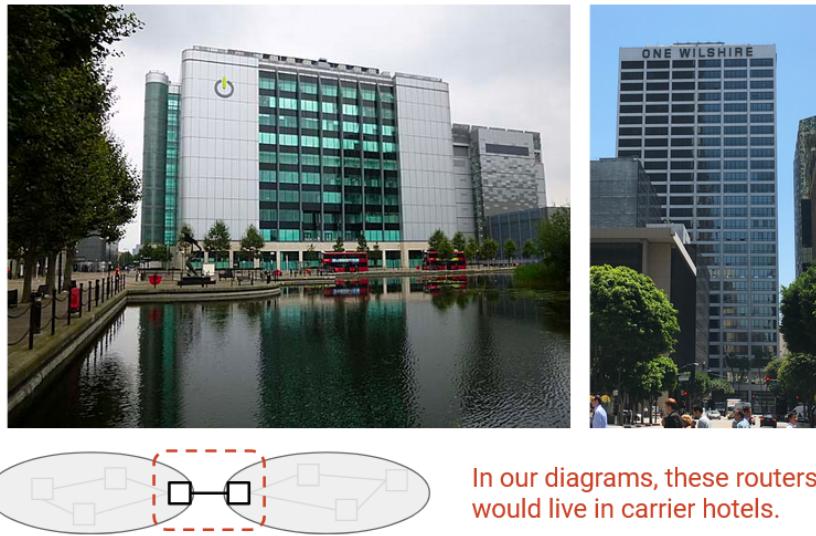
A router runs some routing protocol to populate the forwarding table.

Then, when a packet comes in, the router looks at its destination IP and uses the forwarding table to select a link to forward the packet along. Remember, the forwarding table could contain ranges of addresses.

So far, we've drawn routers as boxes on a diagram. In reality, a router is a specialized computer optimized for performing routing and forwarding tasks. In this section, we'll explore the hardware inside routers.

Where Are Routers?

In real life, homes and offices have small routers to connect hosts to the Internet. Where do all these routers all connect to each other?



Colocation facilities or **carrier hotels** are buildings where multiple ISPs install routers to connect to each other. These buildings are specially designed to have power and cooling infrastructure, and ISPs can rent space to install routers and connect them to other routers in the same building.

Inside a carrier hotel, routers are stacked together into racks (6-7 feet tall, 19 inches wide).

Router Sizes and Capacities

Routers come in all sizes, depending on the user requirements. Home routers only forward traffic for a few users, and the forwarding table has a single default entry. Industrial routers might need to forward traffic from thousands of customers, with a huge forwarding table.



There are different ways we can measure the size of a router. We could consider its physical size, the number of physical ports it has, and its bandwidth.

We can measure a router's capacity as the number of physical ports, multiplied by the bandwidth of each physical port. The speed or bandwidth of a physical port is often called its **line rate**.

Not all physical ports need to have the same line rate. For example, a modern home router might have 4 physical ports that can send at 100 Mbps, and 1 physical port that can send at 1 Gbps. The total capacity of this router is 1.4 Gbps.



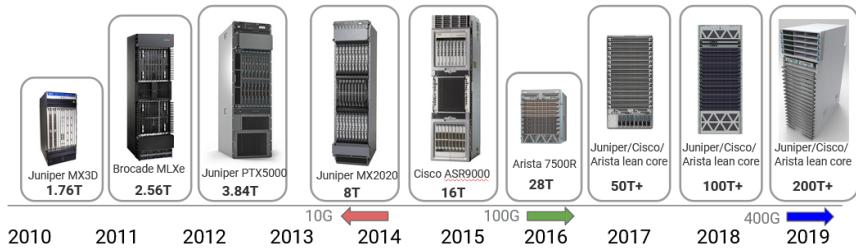
A modern state-of-the-art router used by ISPs might have a line rate of up to 400 Gbps per physical port.

This router contains multiple removable **line cards**, where each line card contains a set of physical ports. A modern router might have 8 line cards, with 36 physical ports per line card, for a total of 288 physical ports.

288 physical ports, each with 400 Gbps bandwidth, gives our router a total capacity of 115.2 Tbps.

This router could cost upwards of \$1 million. Breaking up a router into line cards allows us to install more line cards as more capacity is needed.

In the future, next-generation routers will have 800 Gbps physical ports. Physical space for routers is constrained, so modern improvements are focused on improving the speed per port, instead of increasing the number of ports. (Stuffing more ports into the same space is also difficult because of power and cooling constraints.)



Router capacity has increased over the years in response to the growth in user demand (e.g. video quality has increased from 720p to 8K = 8000p). In 2010, state-of-the-art routers had 1.7 Tbps capacity, and that's increased by a factor of 100 in the past decade. Much of this improvement came from increasing the link speed, from 10 Gbps in 2010 to 100 Gbps around 2016 to 400 Gbps today. These improvements are starting to slow down because of constraints like Moore's law slowing and physical challenges with sending signals at high rate. The next improvement to 800 Gbps is only a 2x increase (compared to the earlier 10x and 4x increases).

Data, Control, Management Planes

The hardware and software components of the router can conceptually be split into three planes. The **data plane** is mainly responsible for forwarding packets. The data plane is used every time a packet arrives and needs to be forwarded. The data plane operates locally, without coordinating with other routers.

The **control plane** is mainly responsible for communicating with other routers and running routing protocols. The result of those routing protocols (e.g. the forwarding table) can then be used by the data plane. The control plane is used every time the topology of the network changes (e.g. when links are added or removed).

Because the data plane and control plane operate at different time scales, and are running different protocols, the hardware and software of a router are optimized for different tasks. In practice, packets arrive much more frequently than the network topology changing. Therefore, the data plane is optimized for performing very simple tasks (table lookup and forwarding) very quickly. By contrast, the control plane is optimized for more complex tasks (re-computing paths in the network).

The **management plane** is used to tell routers what to do, and see what they are doing. Systems and humans interact with the management plane to configure and monitor the router. This is where operators can configure the device functionality. What costs should be assigned to each link? What routing protocol should be run? These need to be manually decided by the operator.

In addition to configuration, the management plane also provides monitoring tools. How much traffic is being carried over each link? Has any physical component of the router failed? This information can be relayed back to the operator.

The management plane is the main place where operators access and interact with the router from outside the device. If the operator is using some piece of code to interact with the router, we usually consider that part of the management plane as well.

The data plane and control plane operate in real-time, receiving and processing packets on the order of nanoseconds (data) and seconds (control). By contrast, the management plane works on the order of tens

to hundreds of seconds. If the operator changes a configuration, the router might spend time performing validation checks and processing the configuration before fully applying the update.

The **network management system (NMS)** is some piece of software run by the operator to interact with the routers. This software computes a network configuration (maybe with the help of manual operator input), and then applies that configuration to the routers. The router publishes some API that the system can use to talk to the router.

The network management system also allows telemetry (statistics and running state) to be read from routers.

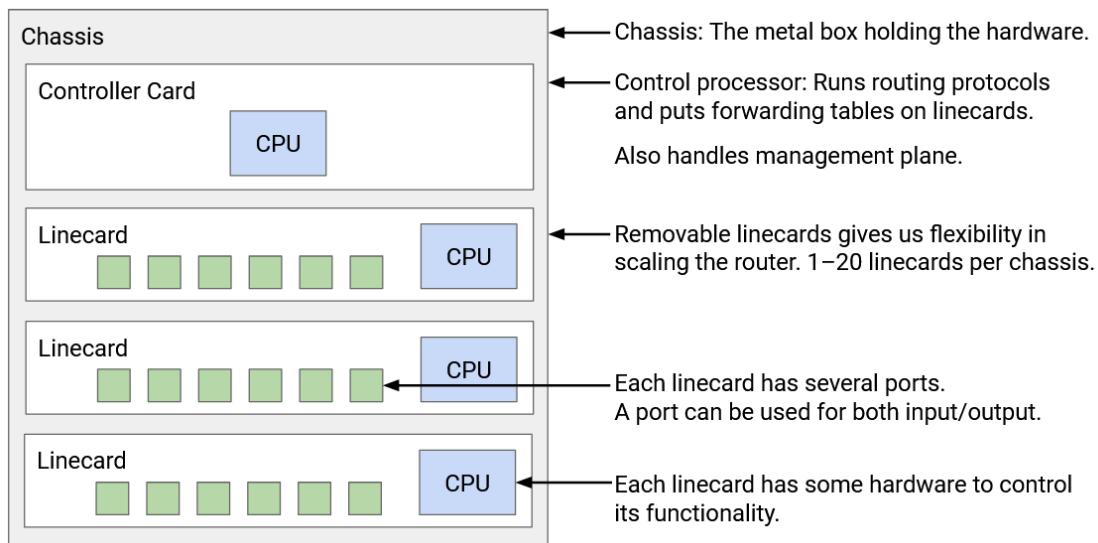
The complexity of the network management system depends on what the operator is trying to achieve.

All three planes are needed to run a router. If we only had the data plane and no control plane, we could forward packets, but we wouldn't know where to forward them.

What's Inside a Router?

We defined a router as a computer that performs routing tasks, but in reality, inside the router, there are many smaller computers (e.g. CPUs, specialized chips) that work together to perform routing tasks.

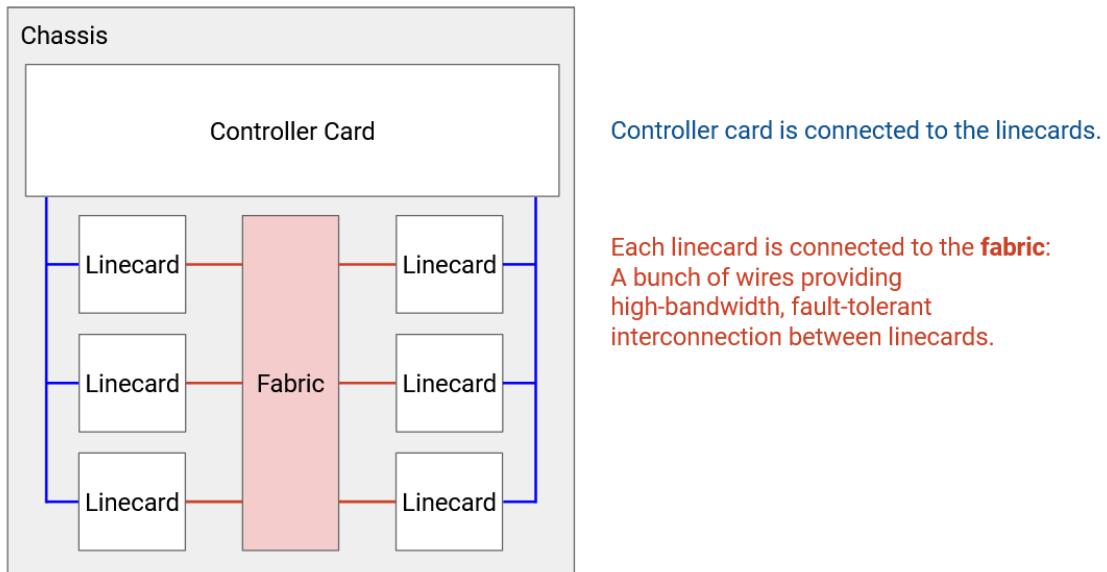
The physical shelf that makes up an industrial-size router is called a **chassis**. Inside the chassis, we install many **line cards**, and we have several physical ports on each line card. Each physical port can be used for either input or output.



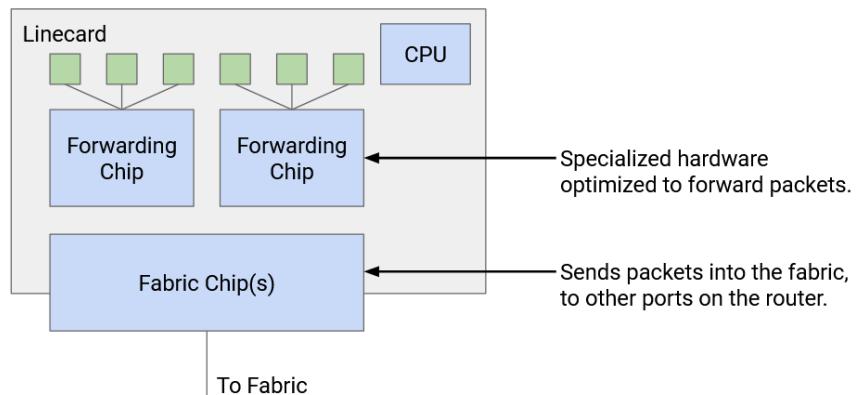
Every physical port has to be connected to every other physical port in the router (both in the same linecard and other linecards). You might receive a packet through one port, and need to forward it out of a port on a different linecard.

It would be pretty inefficient to physically wire each port to every other port. Instead, we have a fabric of wires to connect linecards together. Each linecard also has chips to facilitate connections to the fabric.

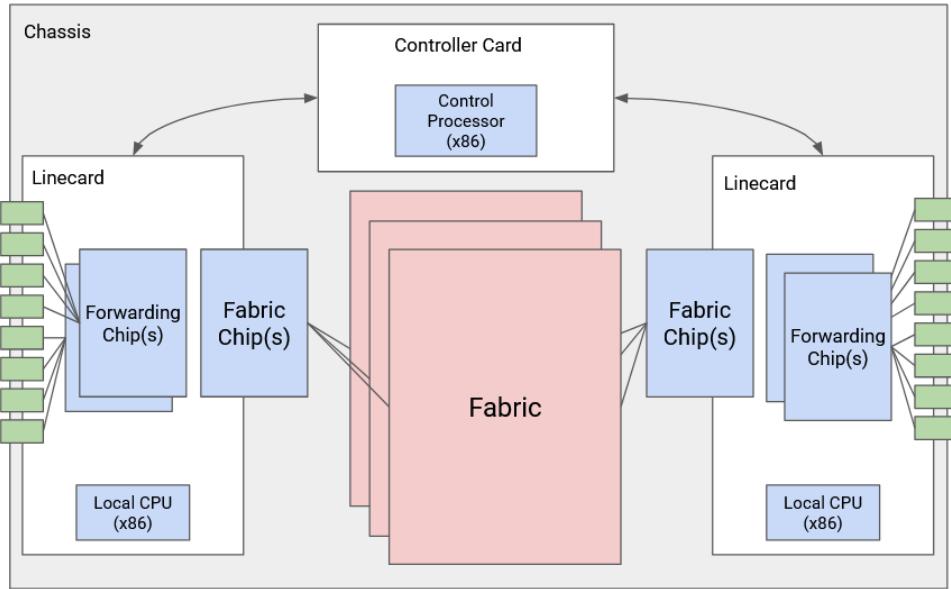
Separate from all the linecards, we have a controller card with its own CPU, which talks with other routers to perform routing protocols. After running some algorithm to compute paths, the controller programs the forwarding chips with the correct forwarding table entries.



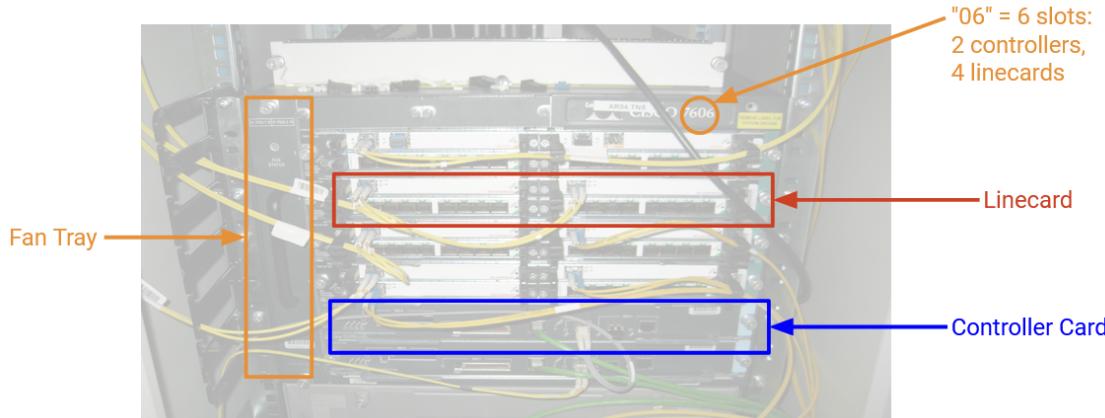
Each linecard has its own local CPU to control linecard functions (e.g. populate the forwarding table). The linecard also has hardware for basic processing of packets (e.g. updating its TTL before sending it out). The linecard contains one or more chips specifically optimized for forwarding.



We can also categorize the router components by the different planes. The data plane is supported by forwarding chips on linecards, the fabric connecting linecards, and the fabric chips connecting the linecards to the fabric. The control plane and management plane are supported by the controller card.

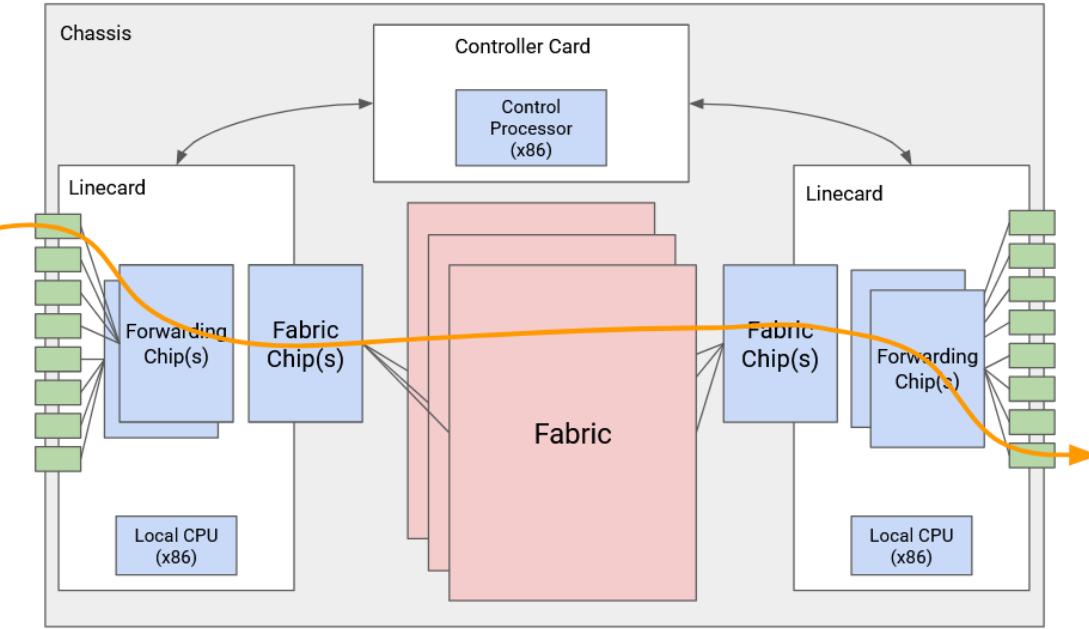


Here's a picture of an industrial router. This router has 6 slots, where 4 of them have line cards, and the other 2 have controller cards. There's also a fan tray for cooling. The fabric connecting linecards is in the back (not pictured).



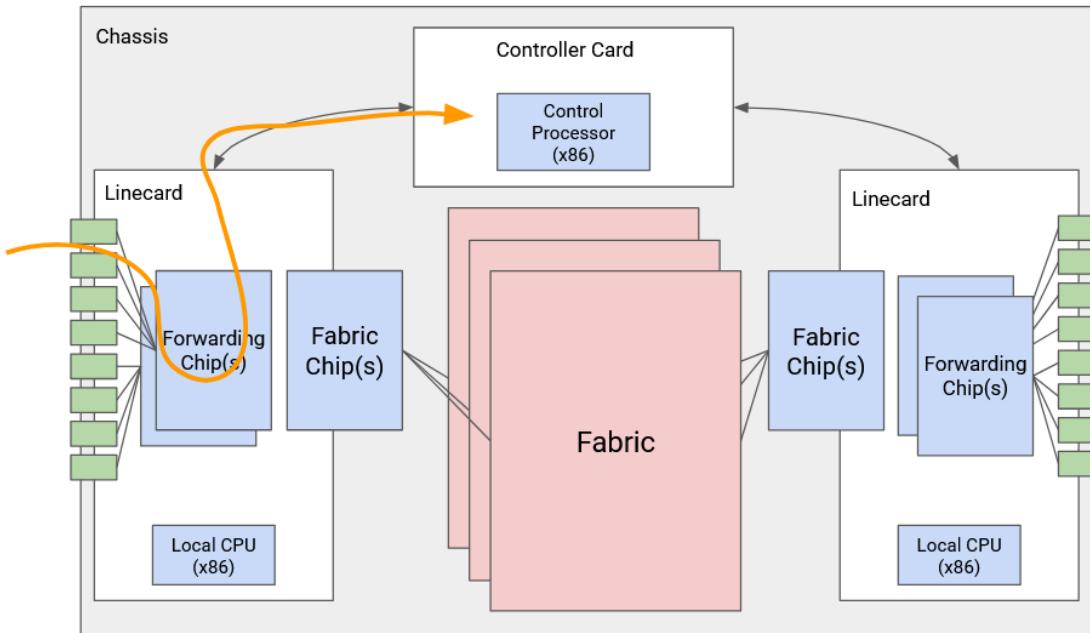
Types of Packets

The most common packet is a **user packet**, containing data from an end host. When the router receives this packet, the forwarding chip first reads the destination field in the header and looks up the appropriate port. If that port is on a different linecard, the packet is sent through the fabric to the appropriate linecard. Once the packet reaches the correct linecard, the packet is sent along the appropriate port.



Some packets are **control-plane traffic**, which are destined for the router itself. In particular, when we run routing protocols, advertisements are sent to the router itself. When the router receives this packet, the forwarding chip sends the packet up to the controller card. The CPU on the controller card processes the packet accordingly.

The last type of traffic is **punt traffic**. These are user packets, but they require some additional special processing. For example, if we receive a packet with a TTL of 1, the packet has expired, and we shouldn't forward it. We might also need to send an error message back to the sender. When the router receives a punt packet, the forwarding chip “punts” the packet to the controller card for special processing.



Scaling Routers

Why is our router broken down into this specific architecture, with forwarding chips and controller cards? Couldn't we run everything on a general-purpose CPU?

The problem is, state-of-the-art routers need to run at enormous scale. At modern speeds of 400 Gbps per second, and assuming 64-byte packets, we have to process 781 million packets per second, per port. Across 36 ports, the entire router has to process 56 billion packets per second. (In practice, the numbers might be slightly lower if some packets are larger.)

This scale is not achievable in software on a general-purpose CPU. To get a sense of scale, if we tried writing a program for forwarding packets, and we ran that program on a CPU, it would be pretty impressive if we could forward one packet every 10 microseconds = 0.00001 seconds. A state-of-the-art router needs to process one packet roughly every 10 nanoseconds = 0.00000001 seconds. Even the most optimized software cannot process packets at this scale. Instead, we need to implement router functionality directly on hardware.

By splitting the router into specialized data plane linecards and control plane controller cards, we create a fast path and slow path. The fast path only involves forwarding hardware and is optimized for forwarding packets at very high rate. The slow path with the control CPU is only used when necessary, and most packets are sent through the fast path. These specialized components make routers much more efficient (uses less power, cheaper, uses less physical space).

Linecard Functionality

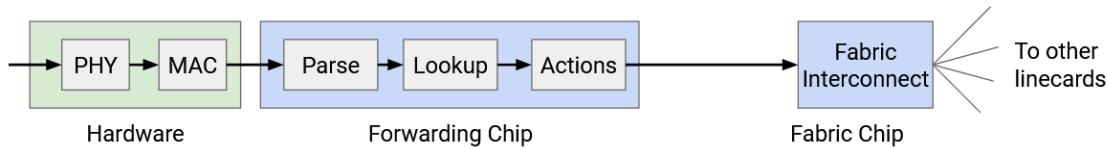
What specific tasks does a linecard need to do when it receives a packet?

First, the linecard needs to take the signal (e.g. optical, electrical) and decode this signal into ones and zeros that make up the packet. This is the **PHY** part of the linecard, which handles the physical layer (Layer 1) functionality.

Once we have a sequence of ones and zeros, we have to read those bits and parse them (e.g. find out which bits correspond to the IP header). We might also have to perform other link-layer operations (e.g. if a link is connected to more than 2 machines). The **MAC** part of the linecard handles the link layer functionality (Layer 2).

Now that we have an IP packet, we have to parse the packet. For example, we need to check if the packet is IPv4 or IPv6. Then, we have to read the destination address and perform a lookup for forwarding (or discover that we need to punt the packet).

We may also need to update various IP header fields. We have to decrease the TTL. Since we updated the header, we also need to update the checksum in the header. We might also need to update other fields like options and fragment (discussed in more detail in the IP header section).



All of this functionality has to happen in a matter of nanoseconds. Even if we somehow did all the processing in one clock cycle, the linecard still has to operate at 0.2 GHz. In practice, all these operations will take more than one clock cycle. Also, we have to do all this processing for every port on the linecard (one forwarding chip supports all the ports).

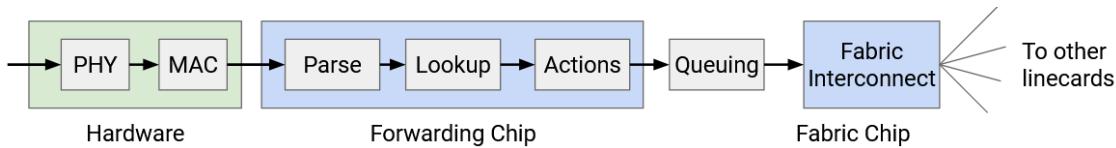
In order to make these operations fast, forwarding chips are extremely specialized for the limited tasks that they perform (e.g. reading packet header, table lookup). You can't write a general-purpose program and run it on a forwarding chip. If a packet requires functionality that the forwarding chip can't support, we can always punt the packet to the general-purpose CPU on the controller card.

Simple operations, like decrementing the TTL, are easy to implement in hardware. More complex operations, like special options, usually require punting to the controller card. In the modern Internet, we avoid special options whenever possible, in order to maximize use of the fast path and avoid punting (if we punted everything, controller cards would be overwhelmed).

The fabric interconnect chips are also similarly specialized. These chips help send packets across the fabric to other linecards. These chips tend to be the most specialized and most high-performance chips in the whole router.

Packet Queuing

TODO



Efficient Forwarding Table Lookup

We now know that routers need to perform lookups in forwarding tables at extremely high rates. One major challenge is that our table entries can contain ranges of IP addresses (192.0.1.0/24) in addition to individual IP addresses. Also, these ranges could be overlapping (a destination could match multiple ranges). How can we make lookups extremely fast?

Ideally, for maximum speed, the forwarding table could contain one entry per destination, with no ranges. Then, we just need to take the destination in the packet, and look up an exact match to learn the next hop.

To achieve this ideal approach, we could expand every range into its individual IP addresses. For example, an entry for the 24-bit prefix 192.0.1.0/24 would be expanded into 256 entries.

R2's Table	
Destination	Port
2.1.1.0/24	5

R2's Table	
Destination	Port
2.1.1.0	5
2.1.1.1	5
2.1.1.2	5
2.1.1.3	5
2.1.1.4	5
...	...
2.1.1.252	5
2.1.1.253	5
2.1.1.254	5
2.1.1.255	5

This is space-inefficient (remember, this is being implemented in hardware). Also, if a route changes, we'd have to update tons of entries in the table. Expanding routes isn't going to work, so we'll have to work with ranges.

Recall that forwarding table lookup is done using longest prefix matching. If multiple ranges match the destination, we pick the most specific range (most prefix bits fixed). If none of the ranges match, we pick the default route (**, 0.0.0.0/32, matches all destinations). If there's no default route, we drop the packet.

How do we implement longest prefix matching in hardware efficiently?

These two prefixes match.
The first one is longer.

R2's Table				
Destination				Port
11101000	01100101	111.....	5
11101000	01100...	9
11101100	01100101	111.....	7
11111...	2

Destination: 11101000 01100101 11101011 11000110

TODO rewrite this to match the diagram

First, for readability, we rewrite all the ranges and the destination in binary. Then, we scan the destination bits, one by one. For the first 21 bits, all four ranges match, so all four ranges are still in play. Then, the 22nd bit is a 1. The first row has a 0 in the 22nd bit, so we can eliminate this row (not a match). The other three rows still match in the first 22 bits, so they're still in play.

Next, we check the 23rd bit, which is also a 1. The second and third rows have a 0 in the 23rd bit, so we eliminate them (not a match). The fourth row is still a match.

At this point, we can confirm that the fourth row is a full match, because it's a 23-bit prefix, and all 23 bits match. No further checking of this row is needed.

We continue checking bit-by-bit, eliminating rows that don't match, and confirming rows that are full matches. Eventually, we have one or more rows that match, and we pick the match with the longest prefix.

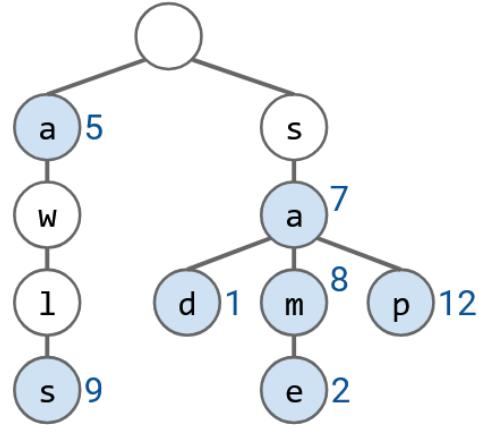
If we implemented this naively, then for every bit, we would have to match that bit against every entry in the forwarding table. The asymptotic runtime would scale with the number of entries in the forwarding table. Can we do any better?

Efficient Lookup with Tries

Thinking back to a data structures class (like CS 61B in UC Berkeley), you might remember that tries are a data structure that efficiently store maps where the keys are strings (in this case, bitstrings). Tries store the key-value pairs by writing out the keys one character (bit) at a time, which enables efficient longest prefix matching.

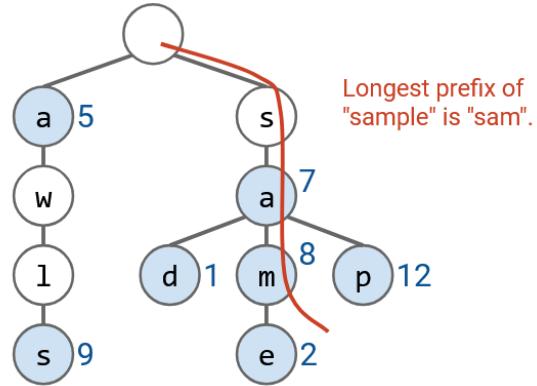
For example, this trie stores a map of words to numbers. If you don't remember tries, it's okay.

Key	Value
a	5
awls	9
sa	7
sad	1
sam	8
same	2
sap	12



If we want to find the longest prefix, just like before, we read the word one letter at a time. This allows us to trace a path down the tree, from the root to a leaf. Along this path, we look for all prefixes in the table (nodes with colors), and pick the longest prefix.

Key	Value
a	5
awls	9
sa	7
sad	1
sam	8
same	2
sap	12

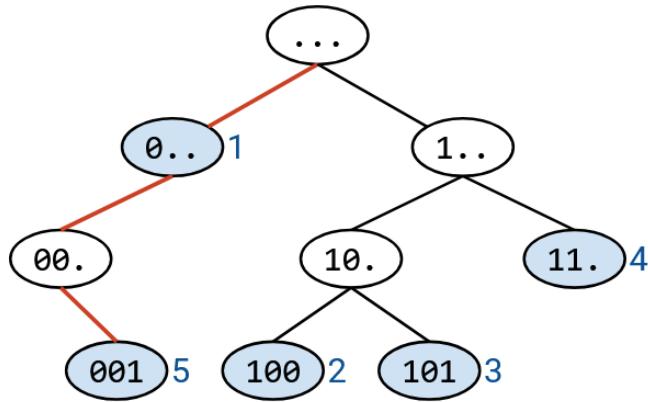


We can use a similar approach for our forwarding table. Each layer of the trie represents one of the digits in the IP address. The zeroth layer is the root (empty string), the first layer represents the first bit, the second layer represents the second bit, etc.

Each node in the trie represents a prefix. For example, the 2-bit prefix 11* is at the second layer of the tree, and the 3-bit prefix 100 is at the third layer of the tree. The trie has all possible 3-bit prefixes. If a prefix is in the forwarding table, at the corresponding node, we write the next hop. If the prefix is not in the forwarding table, we don't write anything in the node (in the picture, colored white).

Key	Value
0..	1
100	2
101	3
11.	4
001	5

Longest prefix of 00100 is 001.



Tracing the path down the tree can be done in constant time. We visit one node per bit of the destination address, and the destination address is always 32 bits (constant). Even if the forwarding table had millions of entries, we'd still pick out 32 nodes.

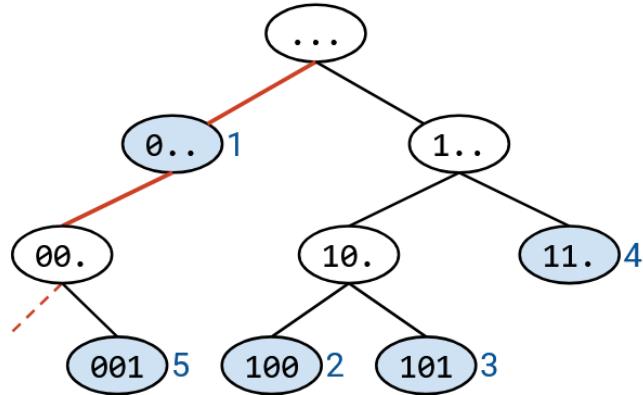
If there's no overlapping ranges, every valid prefix corresponds to a leaf node. If ranges are overlapping, a non-leaf node could also be a valid prefix.

As before, we use the destination address to trace a path down the tree. If we fall off the tree, we stop early and pick the longest prefix out of the nodes we visited.

As a slight optimization, as we walk down the tree, we could keep track of the longest prefix match seen so far. This will always be the most recent match, because the prefixes get longer as we move down the tree. If we fall off the tree, we use the longest prefix match (the most recent match we found).

Key	Value
0..	1
100	2
101	3
11.	4
001	5

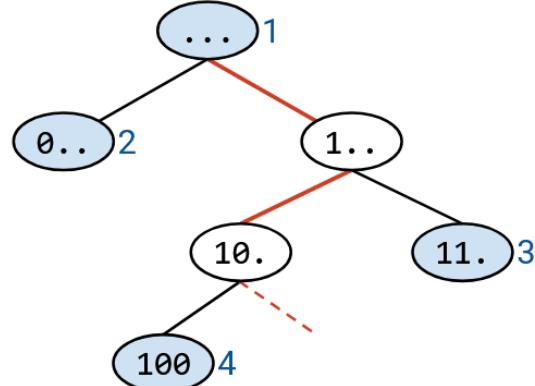
Longest prefix of 00100 is 0.



Note that the default route would be stored in the root node (0-length prefix). Our algorithm of walking down the tree ensures that we only use the default route if no other prefixes match.

Key	Value
...	1
0..	2
11.	3
100	4

Longest prefix of 10100 is default.



All routers have some form of longest prefix matching functionality, but some use more advanced solutions than others. For example, we could add heuristics and optimizations based on real-world Internet assumptions. Some destinations might be more popular, so we might want to look them up more efficiently. Some ports might be used for more ranges. The modern Internet has some conventions for prefix sizes (e.g. the longest IPv4 prefix for routes to other networks is 24 bits). We could also make optimizations for updating the forwarding tables.

Model for Inter-Domain Routing

Inter-Domain Routing

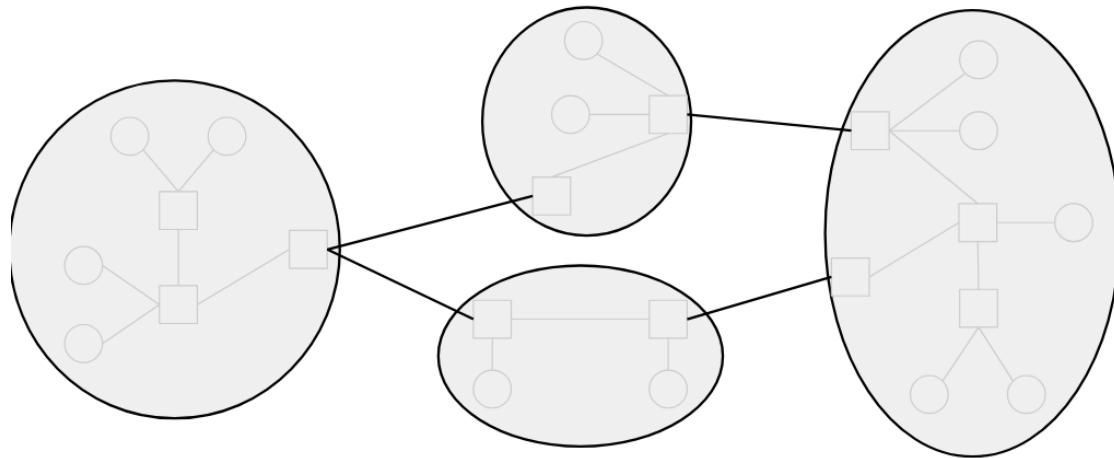
Recall from earlier that routing is performed in a network of networks. We've seen distance-vector and link-state protocols that can be used to implement intra-domain routing, which allows packets to be sent within a local network.

In this section, we'll build a model that will allow us to define inter-domain routing protocols, which can send packets between different local networks. We'll also see how inter-domain and intra-domain routing protocols combine to allow packets to be sent to any host in any network.

Defining Autonomous Systems

We can formalize the notion of a local network by defining an **autonomous system (AS)**, which is one or more local network(s) all run by the same operator. For example, within a company like Google, there might be a local network for employee computers, and another local network for data centers, but both networks are controlled by the same company. The operator can deploy a single intra-domain routing protocol to send messages between machines on any of those local networks. Sometimes, the term **domain** is used to informally refer to an AS, though this term is also used in other protocols, so we will say AS when possible.

To think about routing packets between autonomous systems, we can abstract away all of the individual routers and hosts within the AS, and treat the AS as a single entity. Then, we can draw a graph where each node represents an AS, and edges between two ASes represent a connection between them. This graph is sometimes called the **inter-domain topology** or an **AS graph**.



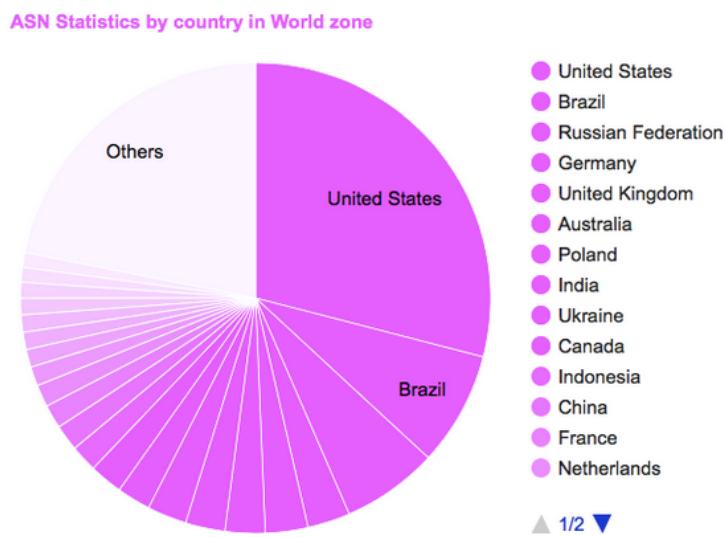
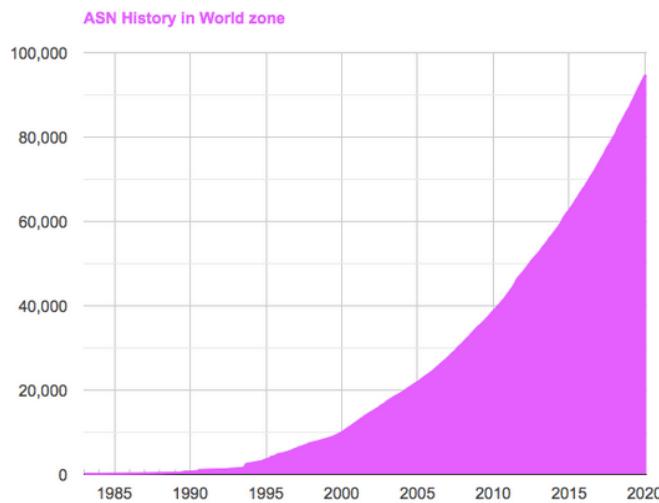
Brief History of Autonomous Systems

In real life, an organization called the Internet Assigned Numbers Authority (IANA) manages a global list of all autonomous systems that exist in the Internet. In order to be an AS, you must register with this

organization and receive a unique autonomous system number (ASN).

Fun fact: In the early days, the IANA was administered manually by a single person, Jon Postel. This meant that anybody in the world who wanted to register a new AS would have to ask for his approval.

Today, there are over 90,000 autonomous systems, with the United States having the most ASes of any country.



Fun fact: UC Berkeley has ASN 25, which is a remarkably low number given that there are so many ASNs. This reflects the fact that UC Berkeley received its ASN very early in the Internet's history (in the 1980s).

Types of ASes

Recall that when modeling the network for intra-domain routing, we made a distinction between end hosts and routers. We'll make a similar distinction for inter-domain routing by defining two types of ASes.

A **stub autonomous system** only exists to provide Internet connectivity to the hosts in its local networks. A stub AS only sends and receives packets on behalf of hosts that are inside the AS, and does not forward packets between different ASes. These are analogous to the end hosts in our intra-domain routing model, which only sent and received their own packets and did not forward other people's packets.

Real-life examples of stub ASes include non-Internet companies (e.g. a bank offering connectivity to its employees) or universities (e.g. UC Berkeley offering connectivity to its students and employees). These organizations are not responsible for carrying Internet traffic from other organizations. The vast majority of ASes in the world are stub ASes.

By contrast, a **transit autonomous system** forwards packets on behalf of other ASes. A transit AS could carry a packet between two different ASes by receiving and forwarding that packet.

Transit ASes correspond to real-life companies whose business includes selling Internet connectivity to other organizations. Real-life examples of transit ASes include AT&T and Verizon, which are companies that you can pay to offer you Internet connectivity. Some transit ASes like AT&T are global, with infrastructure around the world. Others might be specific to a region, like Sonic, an Internet service provider that only forwards traffic to and from California.

Note that a transit AS can still contain end hosts that send and receive packets of their own. Nevertheless, a transit AS is similar to the routers in our intra-domain routing model, which received other users' packets and forwarded them on behalf of users.

This model of stub and transit ASes is what we'll use in these notes, though sometimes, the classification in real life can be less well-defined. For example, major tech companies like Google, Microsoft, and Amazon control massive ASes that carry as much traffic as transit ASes (and maybe even more). Because their primary role is to carry traffic to and from their services (e.g. receive Google search requests and send search results), they could be classified as stub ASes. In recent years, though, these companies have also offered to carry traffic between ASes, so they could arguably be classified as transit ASes as well.

Inter-Domain Topology Is Defined by Business Relationships

In our inter-domain topology, we draw an edge between two ASes if they exchange traffic. What causes two real-life organizations, such as a local bank and Verizon, to agree to exchange traffic? The edges in the AS are defined by real-world business relationships between ASes.

There are two possible ways that a pair of ASes could be related.

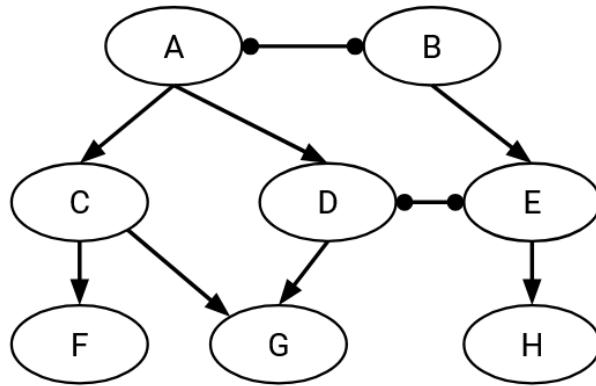
A pair of ASes could be involved in a customer-provider relationship. In real life, the **customer** is paying for service, and the **provider** is offering connectivity in exchange for money. For example, the local bank AS could be the customer, paying the provider, Verizon, for Internet services.

A pair of ASes could also be involved in a **peer** relationship. Two peer ASes usually send each other a roughly equal amount of traffic. In real life, two ASes could agree to become peers by signing a legal contract between companies. Usually, the two peers agree to not pay each other for connectivity services,

as long as the traffic sent in either direction is roughly equal.

AS Graph with Business Relationships

We can draw these relationships into the AS graph by adding arrows to the graph. A directed edge points from the provider to the customer. An undirected edge connects two peers. Note that the graph can contain both directed and undirected edges (not all edges need to have arrows).



Stub ASes in the graph are only customers. They have incoming edges, showing who provides them with connectivity. However, they don't have any outgoing edges, because they don't provide connectivity to others.

By contrast, transit ASes in the graph are the providers. Their outgoing arrows show that they are selling connectivity to other organizations.

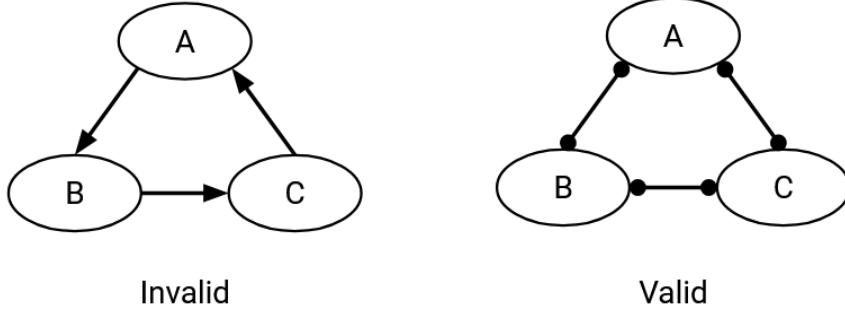
Note that the direction of the arrow does not tell us anything about what direction the packets are being sent. In fact, packets can be sent in both directions even along a directed edge. The customer often pays the provider for the ability to send packets to and receive packets from the rest of the Internet.

AS Graphs are Acyclic

The graph of customer-provider relationships is acyclic. The graph does not contain any cycles consisting of directed edges.

This acyclic property exists because of the real-world implications of having a cycle. In real life, a cycle would mean that A pays B, B pays C, and then C pays A, and it doesn't make sense for money to flow from somebody back to themselves. Also, this cycle would mean that A provides service to C, which provides service to B, which provides service to A. It also doesn't make sense for somebody to provide connectivity to themselves.

To use an analogy, imagine if you paid UC Berkeley tuition for classes, then UC Berkeley paid the UC system for classes, and then the UC system paid you for classes. This business relationship doesn't make any sense!

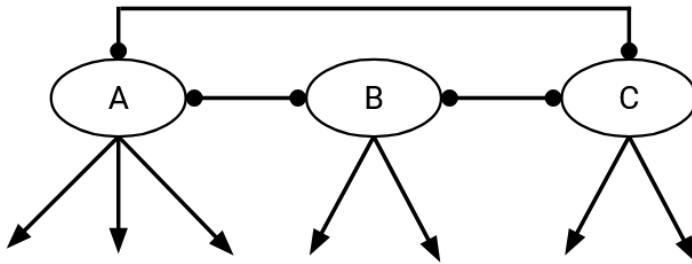


Note that the acyclic property only applies to customer-provider relationships. It is okay if peering relationships form a cycle. For example, it's okay if A-B, B-C, and C-A are all peers. None of them are sending each other money, so we don't have an ill-defined business relationship.

Provider Hierarchy and Tier 1 ASes

A consequence of the graph being acyclic is, we can form a hierarchy of providers. In other words, we can arrange the nodes such that all the arrows point downward. The stub ASes are at the bottom, the providers are at the top. Service flows from higher to lower nodes. The lower nodes pay money up to higher nodes.

At the very top of the hierarchy, there are **Tier 1 autonomous systems**, which have no providers (no incoming edges). Every Tier 1 AS has a peering relationship with every other Tier 1 AS.



A consequence of this hierarchy is: Every non-Tier 1 AS has at least one provider (incoming edge). This makes sense in real life, since you have to pay somebody to offer you connectivity.

In this hierarchy, starting from any AS, and following the uphill chain of providers, always leads to a Tier 1 AS. This also makes sense in real life. The Tier 1 ASes all peering with each other is why the entire Internet is connected (as opposed to, say, two disconnected subgraphs representing two separate Internets where you can only talk to hosts in your own half). In order to guarantee having a path to every other AS in the graph, every AS must have a path upwards that eventually leads to a Tier 1 AS.

TODO-diagram

Some real-world examples of Tier 1 ASes in the AT&T and Verizon (US-based), France Telecom and Telecom Italia (Europe-based), and NTT Communications (Japan-based). There are around 20 ASes that are Tier 1 or nearly Tier 1 in real life. These Tier 1 ASes usually own infrastructure spanning multiple continents (e.g. undersea cables).

The hierarchy structure of the AS graph is defined by real-world business and political motivations. In theory, it would be possible to draw an AS graph that looks like a tree, with a single Tier 1 AS at the root providing services to every stub AS. However, this means that a single real-life entity controls the entire world's Internet access, which may be undesirable for political reasons.

Policy-Based Routing

Recall that in intra-domain routing, our goal was to find paths that are valid (no loops and no dead-ends) and good (least cost).

In inter-domain routing, we still want paths to be valid. However, unlike in intra-domain routing, where there was nothing special about one router over another, each autonomous system has its own business goals and relationships with other ASes (e.g. customer, provider, peer). Therefore, we will need to re-define "good" to reflect the real-world business goals and preferences of ASes.

In order to allow each AS to carry traffic in a way that's compatible with its real-world goals, our routing protocol will allow each AS to set its own policy. Then, the paths computed by the protocol should properly respect each AS's policy.

In theory, ASes can set any sort of policy that they like, although standard conventions do exist (which we'll discuss next). Here are some examples of policies that an AS could set:

- "I don't want to carry AS#2046's traffic through my network." (Defining how I will handle traffic from other ASes.)
- "I prefer if my traffic was carried by AS#10 instead of AS#4." (Defining how other ASes should handle my traffic.)
- "Don't send my traffic through AS#54 unless absolutely necessary."
- "I prefer AS#12 on weekdays, and AS#13 on weekends." (Policies can change over time!)

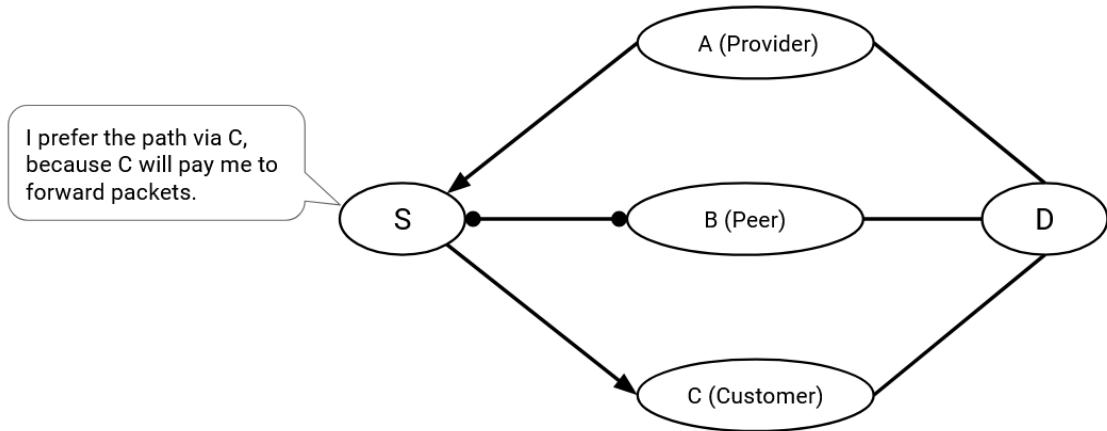
The routing protocol doesn't care why the AS has these preferences. Perhaps I'm refusing to carry traffic from AS#2046 because it's a rival company, but the protocol doesn't need to know that.

Our least-cost routing protocols so far have no way of supporting these policies. Least-cost was a global minimization problem, where every router was trying to solve the same problem. By contrast, in policy-based routing, each AS only cares about its own policy, and there isn't a global problem that everybody is cooperating to solve.

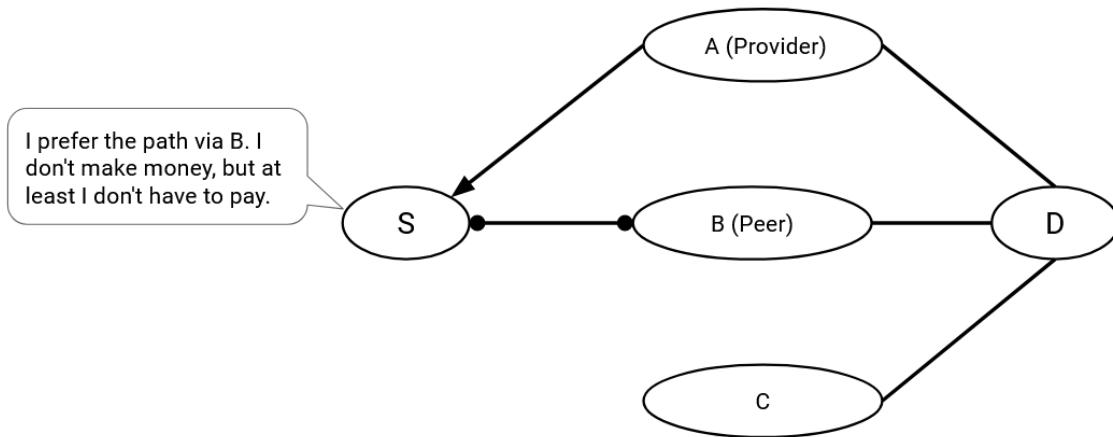
Gao-Rexford Rules for Routing Policies

Although our routing protocol allows each AS to set any arbitrary policy they like, in practice, most ASes set their policies according to some standard conventions, known as the **Gao-Rexford rules**. These conventions are based in the assumption that real-world organizations like making money, and dislike losing money.

There are two broad rules that ASes typically follow. First, when an AS has a choice of multiple routes, the AS prefers to forward packets to the most profitable next hop. Specifically, the AS prefers a route with a next hop that is a customer. If there are no such routes, the AS prefers a route with a next hop that is a peer. The AS will only select a route with a next hop that is a provider if it's forced to do so, because there are no better routes.



This principle dictates the routes that the AS selects. You can think of this principle as a preference-based version of selecting paths in the distance-vector protocol. Instead of selecting the shortest route I know about, I select the route where the next hop makes me money (customer best), or saves me money (if no customers, then peer), and avoids losing money (if no customers or peers, then provider).

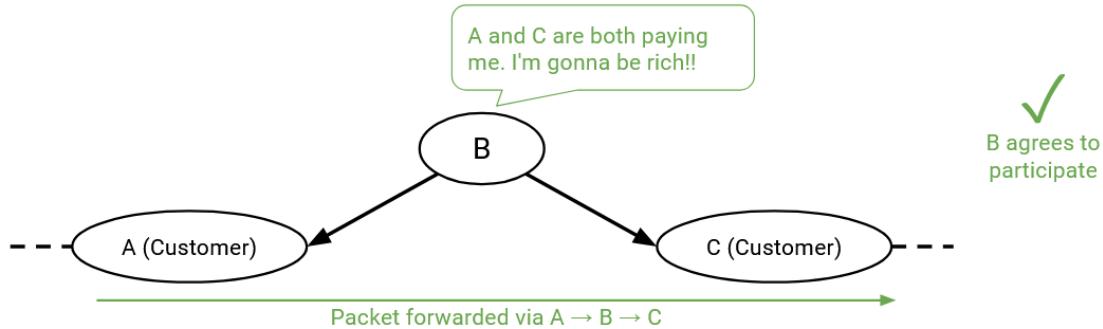


Second, ASes only carry traffic if they're getting paid for it. There's no incentive for ASes to perform free labor. This principle dictates the paths that the AS is willing to participate in. You can think of this principle as a more restrictive version of announcing paths in the distance-vector protocol. Instead of advertising a route to every neighbor, allowing anybody to forward packets through me, I only advertise routes in which I'm paid to forward packets.

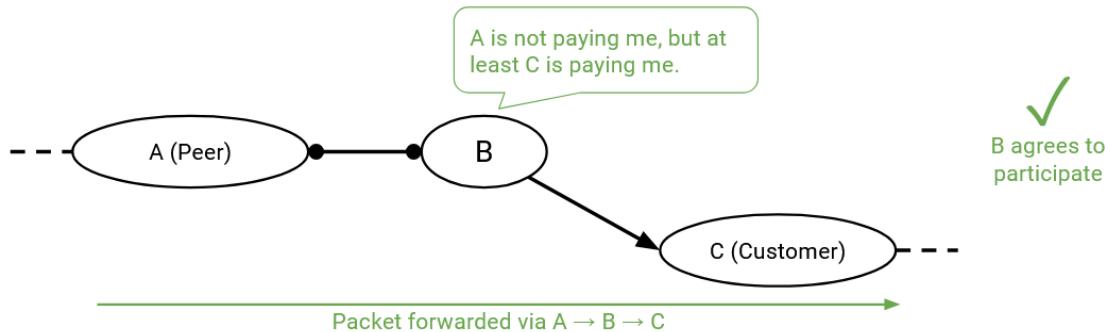
A consequence of this second principle is: As an AS, the traffic I carry should come from a customer, or go to a customer. In other words, for any route going through me, one of my neighbors must be a customer.

Let's go through all the specific cases.

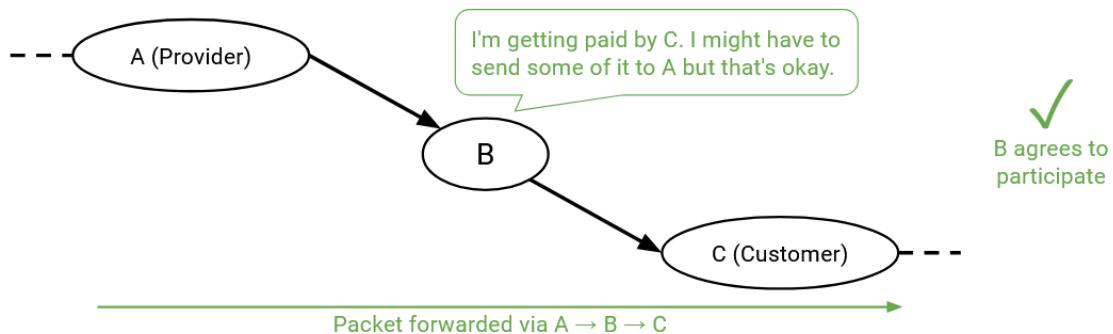
Routes where both of my neighbors are customers are good, because I am getting paid by the two customers to forward packets.



Similarly, routes where one of my neighbors is a customer, and one of my neighbors is a peer are good, because even though the peer doesn't pay me, the customer does.

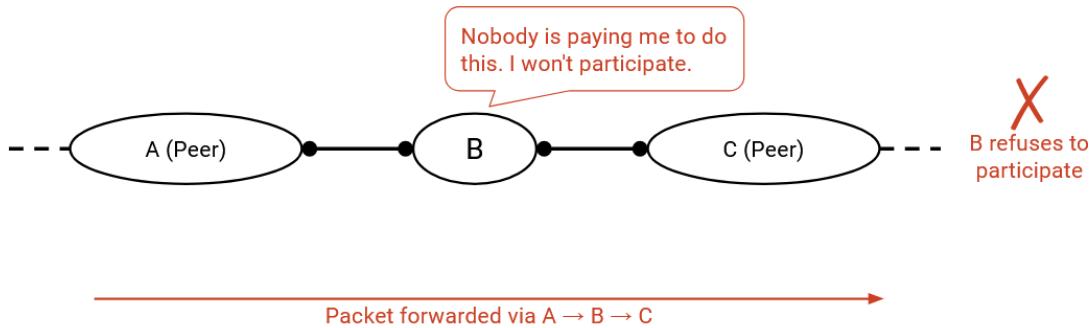


Routes where one of my neighbors is a customer, and the other is a provider are good. At first, it might seem like this path is bad, because the customer is paying me, and then I'm paying the provider. Isn't it possible that I make no money, or lose money from this transaction? That may be true, but if we didn't participate in these routes, we would be a useless AS with no customers. An AS's job is to give connectivity to its users, and participating in these customer-AS-provider routes unlocks more routes to the rest of the Internet.



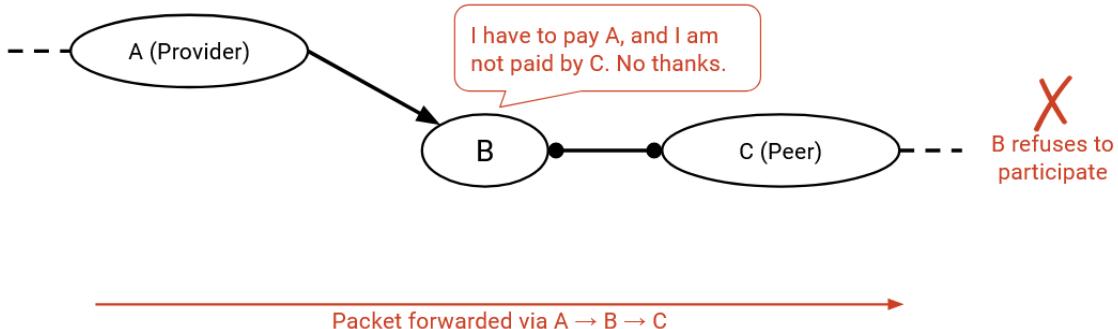
Routes where both of my neighbors are peers are bad, because neither side is paying me to forward packets.

More generally, peers do not provide transit between other peers. Thinking in terms of the hierarchy structure, a path should not stay at a given level for multiple hops.

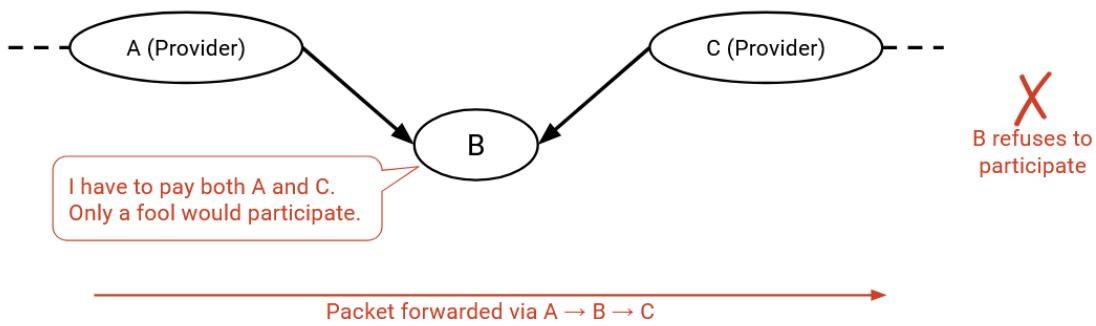


Routes where one of my neighbors is a peer, and the other is a provider are also bad, because again, neither side is paying me to forward packets.

More generally, if an AS has a peering link, that link will only carry traffic to/from its own customers. In other words, when packets arrive at B via that peering link, B's only profitable option is to forward the packet to a customer (not a provider, and not another peer). Similarly, packets from customers can be forwarded through the peering link (customer pays), but packets from providers and peers cannot be forwarded through the peering link (nobody is paying).

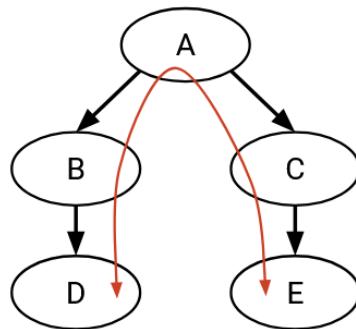


Similarly, routes where both of my neighbors are providers are bad, because I have to pay both sides to forward the packet, and nobody is paying me to do this.



Examples of Gao-Rexford Rules

The policy for selecting routes (customer best, provider worst), and the policy for announcing routes (only announce and participate in routes where one of my neighbors is a customer) will be used in our modified protocol to compute routes that respect each AS's policy. We haven't said how to compute routes yet, but given a route, we can check if it satisfies these two policies.



In this example, suppose that a computer in D (a stub AS) wants to talk to a computer in E (another stub AS). D and E might want to exchange messages (remember, arrows represent customer/provider relationships, not direction of packets).

One possible path for the traffic is D, B, A, C, E (and reverse for messages from E to D).

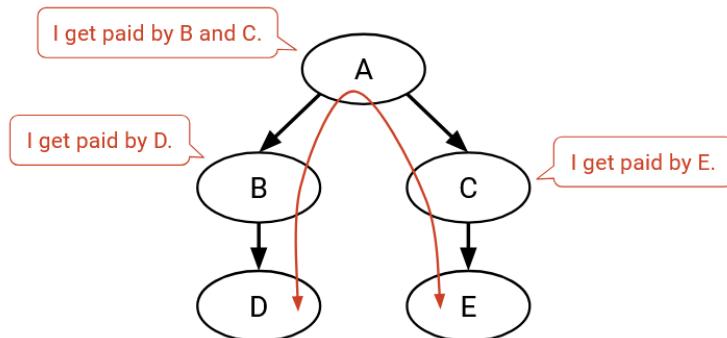
Who is paying whom in this path? Since traffic is being sent along the D-B link, the customer (D) must pay the provider (B). Similarly, E must pay C, and B and C must both pay A.

Will the transit ASes A, B, and C agree to announce and participate in this route? Let's check each of their neighbors.

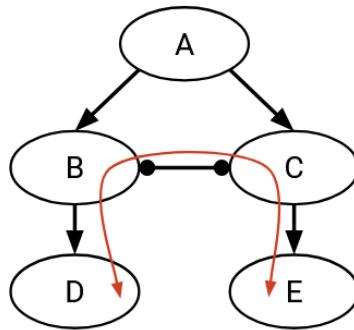
A's neighbors along this path are both customers, so A is making money, and thinks this path is good.

B's neighbors are a customer (D) and a provider (A). B is making money from the customer (D), and thinks this path is good. (Remember, paths with one customer neighbor and one provider neighbor are good, even if the AS has net profit of 0, because they enable greater connectivity.)

Similarly, C has at least one customer neighbor (E), so it also thinks this route is good.

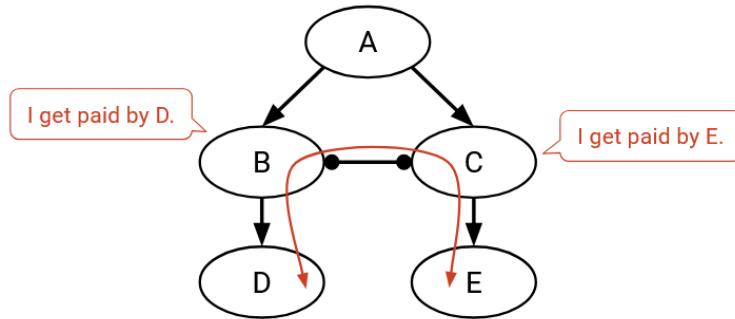


Instead of B and C both paying A, perhaps they choose to establish a peer relationship, which causes the AS graph to change:



Now, another possible path for the traffic is D, B, C, E. Now, D still needs to pay B, and E still has to pay C. However, B and C no longer need to pay A, and they don't pay each other (peering relationship).

Again, we can check if the transit ASes on this path, namely B and C, will agree to announce and participate in this route. B's neighbors are a customer (D) and a provider (C). B is making money from the customer (D), and thinks this path is good. Similarly, C has at least one customer neighbor (E), so C also thinks this path is good.



We've just reasoned that there are two good paths that can be used to send messages from D to E. Now, B must decide to forward through either path B-A-C-E, or path B-C-E. Which path should B choose? According to our first principle, B prefers the most profitable path (not the shortest path). In B-A-C-E, the next hop is provider A (who we'd have to pay), and in B-C-E, the next hop is peer C (no payment needed). Therefore, B will select the path through C, giving final path D-B-C-E.

Note: It seems like B and C are saving money with the additional peering relationship, so why wouldn't every AS establish peer relationships to save money? In real life, establishing a link also requires installing physical infrastructure (e.g. laying cables underground), so there's a cost trade-off to establishing new relationships between ASes, in exchange for cheaper routes.

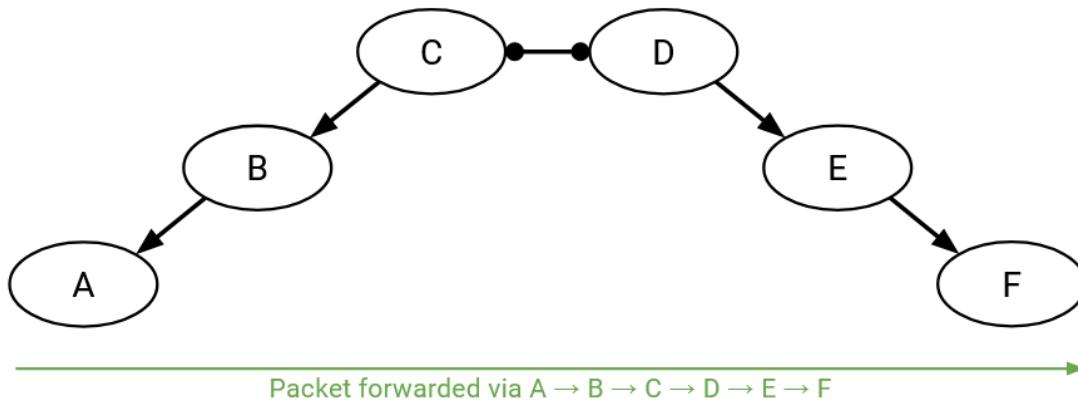
Routes are Valley-Free

More generally, paths in the AS graph are always **valley-free**.

Thinking in terms of the hierarchy structure, if a path includes a lateral hop via a peering link, the immediate next hop needs to go downhill to a customer. The next hop cannot be lateral again (both neighbors peers), and the next hop cannot be uphill to a provider (peer and provider neighbors).

Thinking in terms of the hierarchy structure, if a path includes a downward hop from provider to customer, the immediate next hop must continue to go downhill to one of its customers. The next hop cannot be lateral again (neighbors are provider and peer), and the next hop cannot be uphill (neighbors are both providers).

If a downhill link must be followed by another downhill link, then we can conclude that as soon as you have a downhill link in a path, all subsequent links must also be downhill. A valley is a path that goes downhill, and then turns around to start going uphill. Paths cannot contain valleys, because once you start going downhill, you must continue downhill all the way to the destination.



In summary, here are the rules we've derived (though it's better to understand them in terms of respecting AS money preferences, instead of memorizing them):

- An uphill link can be followed by peering link, a downhill link, or another uphill link. (If the previous hop pays me money, I'm happy to forward the packet to anybody.)
- A peering link can only be followed by a downhill link. (If the previous hop isn't paying me, I need the next hop to be a customer that pays me.)
- A downhill link can only be followed by a downhill link. (If the previous hop is a provider I'm paying, I need the next hop to be a customer that pays me.)

These rules mean that routes are always valley-free and single-peaked. A route can start 0 or more climbing uphill links. Eventually, it will reach a single peak, and traverse 0 or 1 peering links. Then, the route must start going downhill all the way to the destination (no more lateral or uphill moves).

Paths cannot have valleys (going downhill and then turning to go back uphill). Also, paths cannot have lateral moves anywhere except the peak. As soon as you make a lateral move, you must turn around and go back down. You cannot continue traveling laterally or uphill.

ASes Want Autonomy and Privacy

When designing a protocol for computing inter-domain routes, our protocol should respect the autonomy and privacy of each AS.

ASes want **autonomy**, the freedom to choose their own arbitrary policies, without coordinating with other ASes, or worrying about what policies the protocol allows. In practice, the policies usually follow the money-based principles we described, but the protocol shouldn't force the AS to follow any specific policy.

ASes also want **privacy**. ASes don't want to have to explicitly tell others in the network about their preferences and policies. For example, an AS shouldn't need to explicitly tell everybody about whether its neighbors are peers, customers, or providers. This reflects real-world business strategies. As a company, you might not want to reveal information about your customers and providers to your rivals.

Note that our definition of privacy says that ASes shouldn't need to *explicitly* reveal their policies. In practice, ASes still need to coordinate with the rest of the network to agree on paths through the network, so some amount of information leakage is inevitable. Reverse-engineering techniques exist to trace the routes packets are taking through the network.

For example, it's unavoidable that others on the network can discover what route a packet is taking. However, our protocol shouldn't force an AS to tell the world "I liked this path more than this other path." We also shouldn't force an AS to disclose who their providers, peers, and customers are.

Border Gateway Protocol (BGP)

Brief History of BGP

Least-cost routing protocols are closely related to the shortest-paths problem in graph theory, which has been studied in computer science even before the Internet existed. Dijkstra's algorithm is from 1956, and the Bellman-Ford algorithm is from 1958. When developing early routing protocols, designers could adapt the ideas from these algorithms.

In the early days, the Internet was a government-funded project, where the network was centrally controlled by the US Department of Defense. The notion of autonomous systems didn't exist yet, and least-cost algorithms could be scaled up to the small size of the early Internet. Eventually, as the Internet grew, the government transferred control over to different commercial entities, who had to develop inter-domain routing protocols on the fly.

Unlike the early least-cost routing protocols, the notion of autonomous systems each having their own private policies had no precedent in computer science. The ideas behind inter-domain routing protocols had to be developed on the fly in response to the needs of these new Internet companies.

BGP was created in 1989-1995, and its ad-hoc development process means that the protocol isn't perfect. If we could rewrite the protocol from scratch today, the result might look different. However, the protocol has proven to be effective and resilient, and is still the inter-domain routing protocol in use today. (Remember, everybody has to agree to use the same inter-domain routing protocol, so there is only one.)

BGP is based on Distance-Vector

Recall that we saw two classes of intra-domain routing algorithms: distance-vector algorithms and link-state algorithms. When designing BGP, which class of algorithm would be a better starting point for our design?

Remember that in BGP, we need to respect the privacy of individual ASes. If we used a link-state protocol, then every AS has to tell the entire network about its policies, so that everybody has the full knowledge to compute routes by themselves.

Also, in BGP, we need to respect autonomy and allow each AS to make its own policy decisions. However, a link-state protocol requires everybody to compute routes in some consistent way (e.g. everyone agrees to use least-cost paths).

Link-state algorithm don't respect the privacy or autonomy of ASes, so link-state would be a poor choice of algorithm to design BGP around. By contrast, distance-vector would allow every individual AS to make its own decisions about what routes to accept/reject, and what routes to announce. Also, because distance-vector is not a global protocol, each AS doesn't need to know about everyone else's policies in order to compute valid routes.

Many core ideas in distance-vector protocols will still apply in BGP. The advertisements that we send and receive will still be specific to one destination. Just like in previous sections, we'll think about advertisements and routes for a single destination, but know that the protocol is being run for multiple destinations simultaneously.

In both distance-vector protocols and BGP, each AS computes routes using only information from the advertisements it receives, without seeing a global picture of the network topology. Also, in both types of protocols, the AS will send and receive advertisements indefinitely, until everybody has converged on a set of routes.

BGP follows the same core idea as distance-vector protocols, but with a slight change in terminology. Instead of saying that each AS announces or advertises routes, we say that the AS is **exporting** routes. Then, each AS listens to advertisements and selects its preferred route, which we'll call **importing** routes.

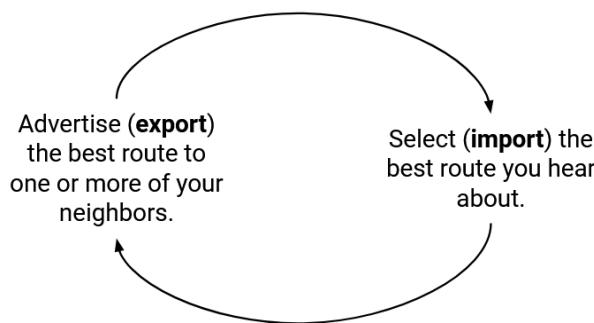
Distance-vector is a good starting point, but what's missing?

Distance-vector protocols are designed to find least-cost routes, but in BGP, we want routes to be decided based on each AS's individual policies.

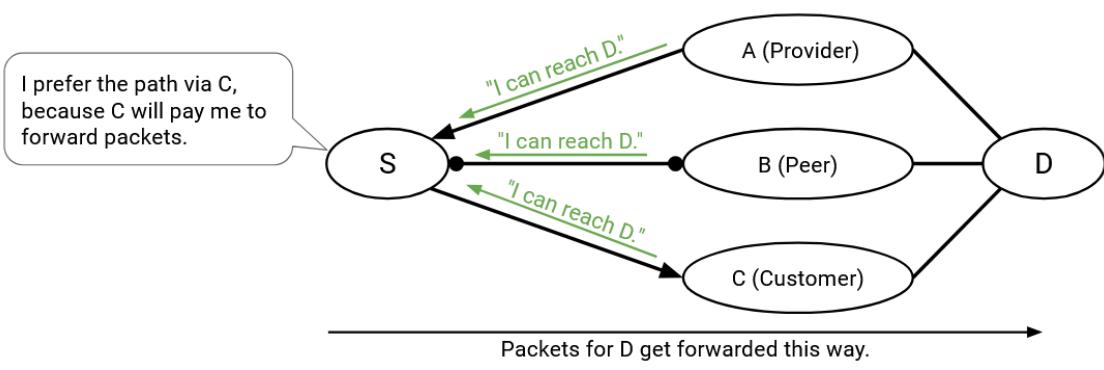
Policy-Based Importing and Exporting

At a high level, in order to support policies, we will change the rules for importing and exporting routes. Each AS will only export (advertise) routes that the AS likes (according to its policy). Also, when importing (selecting) routes, the AS will select the best route according to policy, not distance.

When an AS receives multiple advertisements for the same destination, instead of picking the shortest route, the AS now selects (imports) a route based on policy.



Remember that advertisements propagate outward from the destination, and messages are forwarded closer to the destination (the opposite direction from advertisements). The import decision dictates where an AS is sending its outbound traffic. For example, if S hears advertisements from A, B, and C about the same destination, S's import decision (A or B or C) determines where packets for that destination will be forwarded.



In the distance-vector protocol, when I receive an announcement and install a new route, I always announce this new route to all my neighbors.

Now that ASes have their own policies, they can choose whether or not they want to participate in a route. If an AS has a route it potentially dislikes, it can now choose to not export that route to certain neighbors.

For example, suppose my policy is that I don't want to carry C's traffic. This could be because of monetary reasons, or it could be some other policy decision by me. When I accept an advertisement and install a route, it's okay if I don't advertise that route to C.

Again, remember that data flows in the opposite direction from advertisements. The export decision dictates what inbound traffic an AS is willing to carry. If I export a route, I am agreeing to participate in this route and let other people forward packets to me along this route.

A consequence of this rule is, even if the underlying graph is connected (a path exists between any two nodes), it is not guaranteed that every AS can reach every other AS. In practice, we'll be able to guarantee reachability by establishing some conventions about the ASes' policies and the structure of the AS graph.

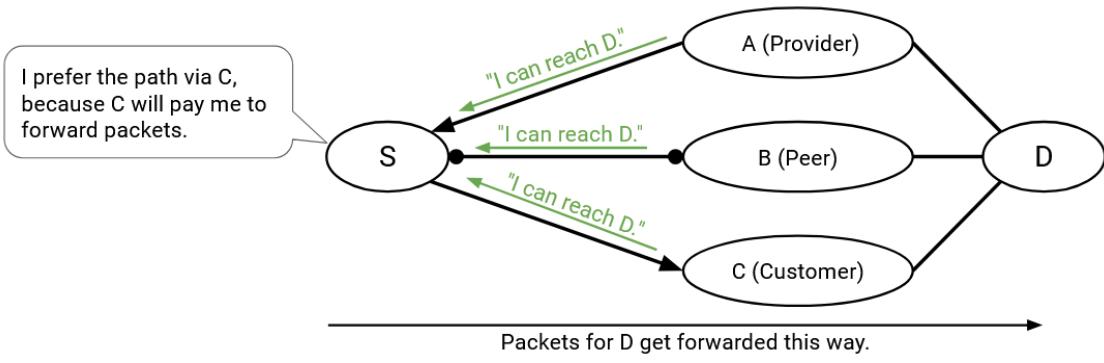
Implementing Gao-Rexford Rules

In general, BGP supports arbitrary policies, but arbitrary policies don't give us any guarantees that the Internet is fully connected (packets can go from any source to any destination).

Recall that the **Gao-Rexford rules** enforce a more restrictive set of policies, based on common money-based import and export policies. Nobody enforces that an AS must follow these rules. However, if ASes agree to follow these rules, we can make stronger assumptions about Internet connectivity.

Brief history: The rules are named for Lixin Gao and Jennifer Rexford at AT&T in the 1990s. Back then, each AS made up their own policies on the fly. Gao and Rexford surveyed ASes about their policies to come up with these rules, and used them to prove guarantees about the Internet.

When importing routes, the Gao-Rexford rules say that the AS prefers to import a route advertised by a customer, over a route advertised by a peer, over a route advertised by a provider.

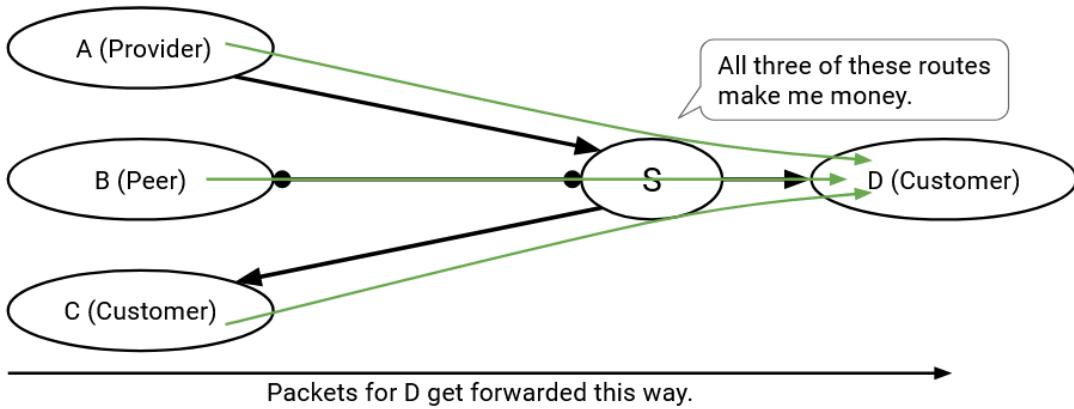


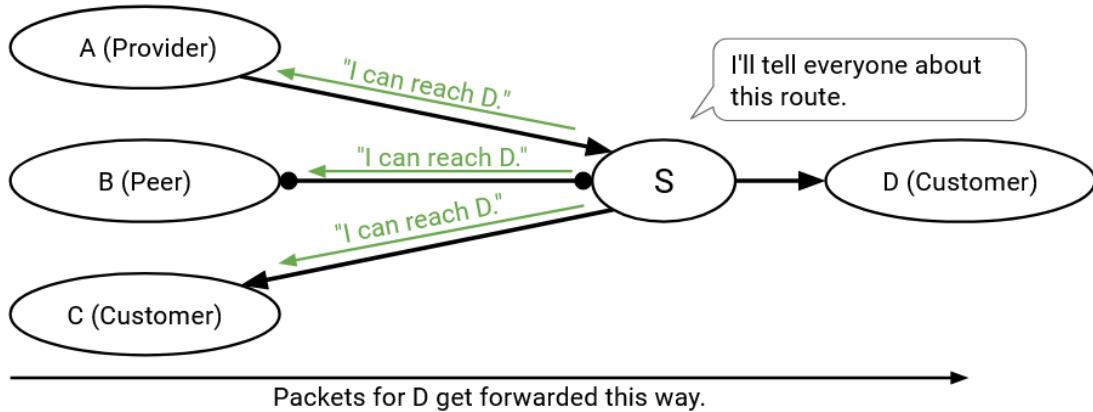
In practice, ASes also implement additional tiebreaking rules in addition to the Gao-Rexford rules. For example, if I receive advertisements from two customers, I need some additional tiebreaker to prefer one of them. Performance is a common tiebreaker, where we pick routes with higher bandwidth or shorter paths.

Based on the Gao-Rexford rules, how should we export paths? Recall that an AS agrees to participate in a route if at least one neighbor is a customer. Therefore, the AS should only advertise routes if the resulting route, if accepted, has a neighbor on one side.

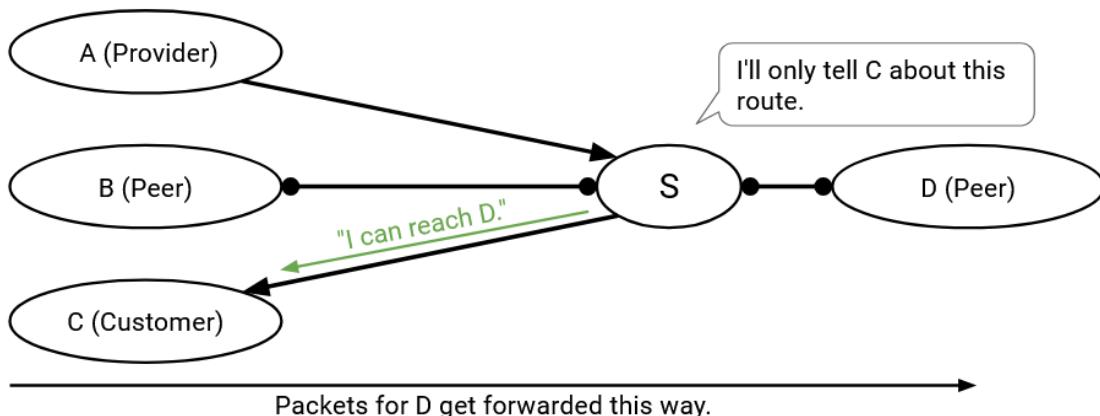
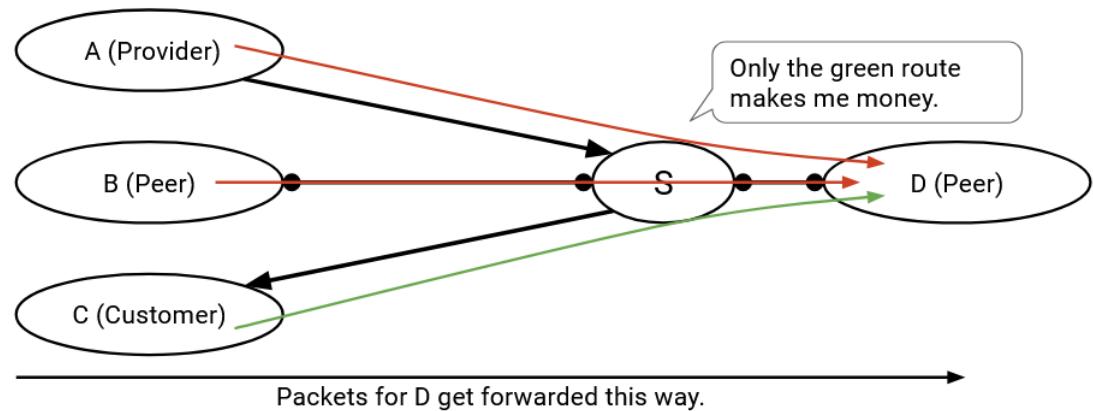
Let's go through all the specific cases.

I receive and install a route from a customer. This means that the next hop on this route is that customer. Who should I export this route to? I've already guaranteed that there's a customer on one side paying me, so I can export this route to everybody (customers, providers, and peers).

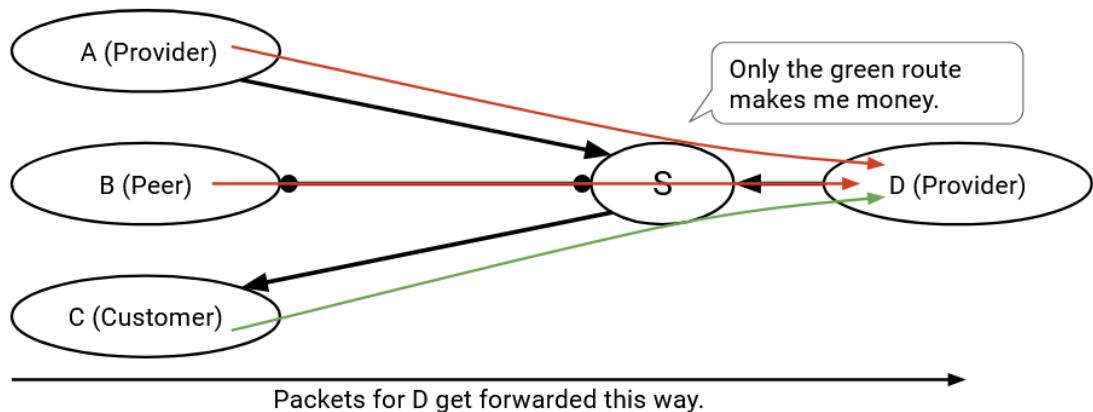
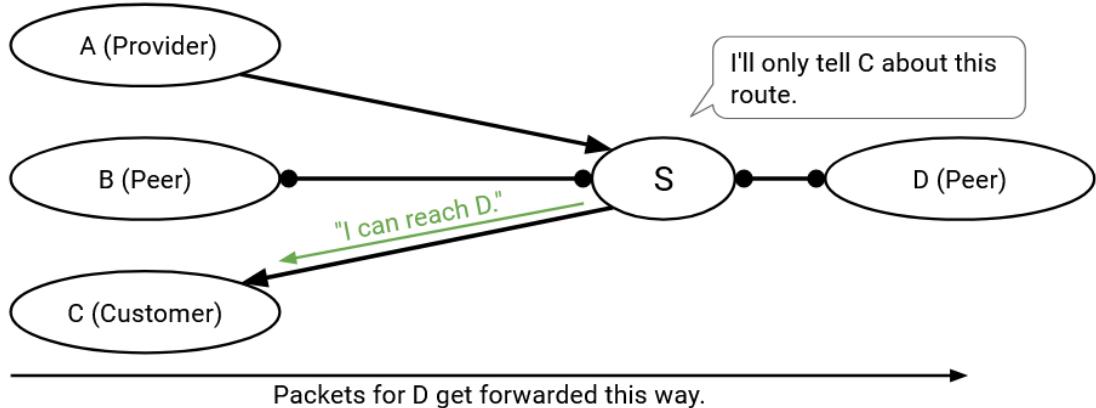




I receive and install a route from a peer (the next hop is a peer). Who should I export this route to? Nobody is paying me yet, so I should only export this route to customers. If I export this route to a peer or provider who accepts, then I've created a route where neither side is paying me.



Similarly, if I receive and install a route from a provider, I should only export this route to customers, because I need at least one side to pay me, and the provider isn't paying.



Route advertised by...	Export route to...
Customer	Everyone (providers, peers, customers)
Peer	Customers only
Provider	Customers only

The Gao-Rexford rules allow us to provably show that this statement is true: Assuming that the AS graph is hierarchical and acyclic, and all ASes follow the Gao-Rexford rules, then we can guarantee reachability and convergence in steady state.

Breaking down the specific terms in the statement: Reachability means that any two ASes in the graph can communicate. Convergence means that all ASes will eventually stop updating their paths, and the network will reach a steady state with valid paths between any two ASes. “In steady state” means that if the network topology changes, the paths might take some time to change and reach steady state again.

Recall that hierarchical means that starting from any AS, moving up the hierarchy (from customers to providers) will lead to a Tier 1 AS. Acyclic means there is no cycle of customer-provider relationships (directed edges).

The proof of this statement requires that everybody follows the Gao-Rexford rules. If ASes were running their own arbitrary policies, the guarantees would no longer hold.

Modification: BGP Aggregates Destinations

There are two more modifications we need to make to the distance-vector protocol.

In distance-vector protocols, we showed that each destination had a unique address, and the forwarding table mapped each destination to a next hop and distance.

In BGP, each AS is addressed by a prefix, which indicates that all machines inside that AS share the same prefix.

These forwarding tables could get very large (imagine if a provider had hundreds of customers), and every single destination would need to be described in a separate announcement. Is there any way we can express this forwarding table more concisely?

To improve scalability, BGP allows ASes to **aggregate** multiple destinations into a single forwarding table entry, and announce a more general prefix that includes all of the destinations combined.

Destination Prefix	Next hop
12.1.0.0/16	Physical port #1
12.2.0.0/16	Physical port #1
12.3.0.0/16	Physical port #1

→

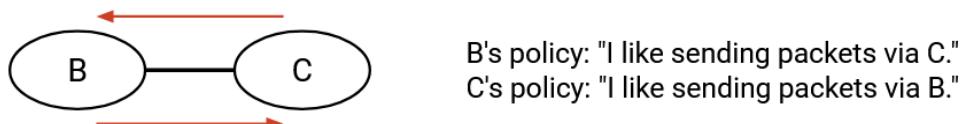
Destination Prefix	Next hop
12.0.0.0/8	Physical port #1

Note that in practice, BGP has conventions on the size of the prefixes being announced. For example, ASes will not make an announcement for an individual IP address. 24-bit prefixes (blocks of 256 addresses) are usually the smallest unit of addresses that are announced.

Modification: Path-Vector Protocol

In least-cost protocols such as distance vector, we didn't have to worry about loops. Every router was trying to find least-cost routes, and by definition, the least-cost route will not contain a loop.

Now that each AS is choosing routes based on its own preferences, we've lost the guarantee of no loops. For example, suppose B likes paths through C, and C likes paths through B. We've created a routing loop!



To fix this problem, instead of the distance to destination, BGP announcements will include the full AS path to the destination. This changes the protocol from a distance-vector to a **path-vector** protocol.

For example, in a distance-vector protocol, A would announce: "I can reach the destination with cost 1." Then, B would announce: "I can reach the destination with cost 2."

In a path-vector protocol, A would announce: "I can reach the destination with the path [A]." Then, B would announce: "I can reach the destination with the path [B, A]."

With this modification, ASes can determine whether an advertised path contains a loop by tracing through the path in the advertisement. Specifically, if I receive an advertisement, I just need to check if the path includes myself. That would cause the packet to be sent back to me, creating a loop, so I would ignore that advertisement and not accept or advertise the route with the loop.

Note: If everybody agrees to discard routes with loops, this guarantees that advertisements won't contain loops. The only way that an advertised route would create a loop is if I see a route that already includes myself, and the addition of myself is what creates the loop.

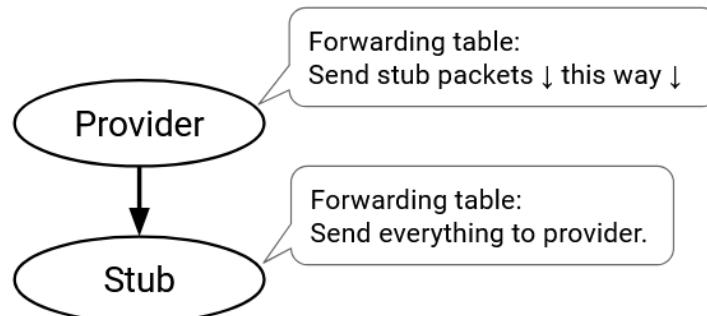
The change from distance-vector to path-vector also allows ASes to implement arbitrary policies. In a distance-vector protocol, I might have a policy like "avoid AS#2063 when possible." If I receive an advertisement "I can reach the destination with cost 12," I have no idea if the path being advertised goes through AS#2063. If instead, the advertisement contained the entire path, I can check if the path goes through AS#2063 before deciding to accept or reject it.

Note: The conventional BGP import policy we saw earlier (prefer selecting routes that go to customers, over peers, over providers) only depends on the next hop, not the entire path. Still, the change to path-vector is useful for loop detection, and lets us generalize the protocol to arbitrary policies.

Stub ASes Use Default Routes

Some ASes don't need to run BGP to determine how to forward packets through the network. In particular, if a stub AS is only connected to a single provider, then every packet bound for other ASes should be sent to that one provider. The stub AS can install a single hard-coded **default route** for all destinations in other ASes.

What about other ASes trying to send packets to the stub AS? The stub can ask the provider to install a **static route**, which tells the provider how to send packets to the stub AS. Now, the provider can run BGP and advertise this static route to the rest of the Internet. The stub can ask the provider to hard-code the static route, and the stub never has to run BGP, since the provider is advertising routes to the stub on behalf of the stub.



Most small ASes in the Internet are stub ASes that use default and static routes.

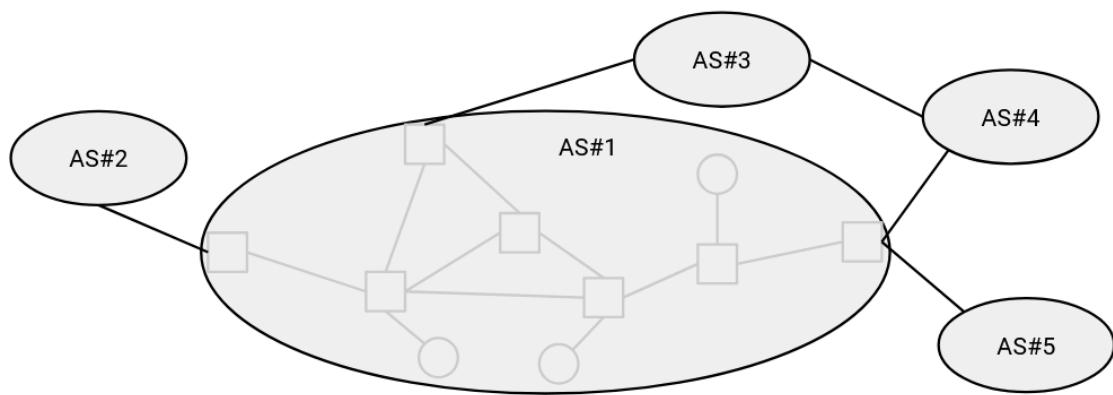
Stub ASes are similar to end hosts in intra-domain routing. They send and receive packets for their own AS, but do not forward packets of their own and do not participate in the routing process. Just like in intra-domain routing, we will usually ignore stub ASes and only consider transit ASes that actually participate in BGP.

BGP Implementation and Issues

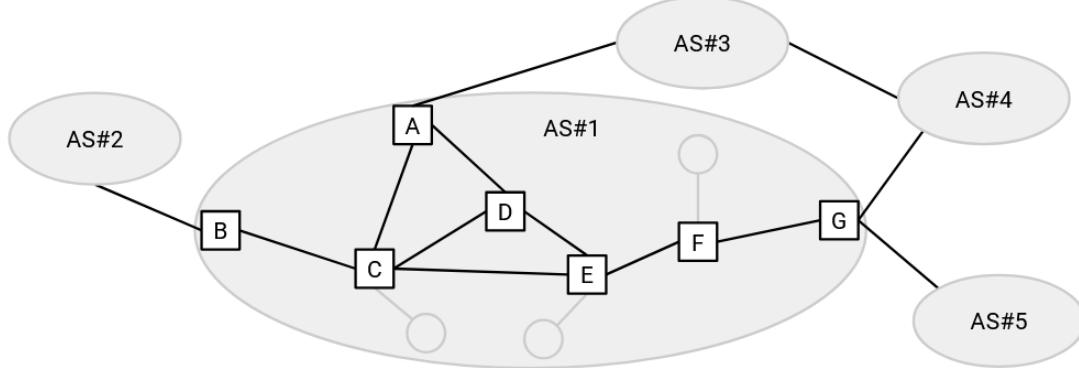
Border and Interior Routers

At this point, we have an intuitive picture of how BGP works between ASes. In this section, we'll show how BGP is actually implemented at the router level. In doing so, we will also show how BGP interacts with the intra-domain routing protocols from earlier.

So far, our model of inter-domain routing has treated an entire AS as a single entity, importing and exporting paths.

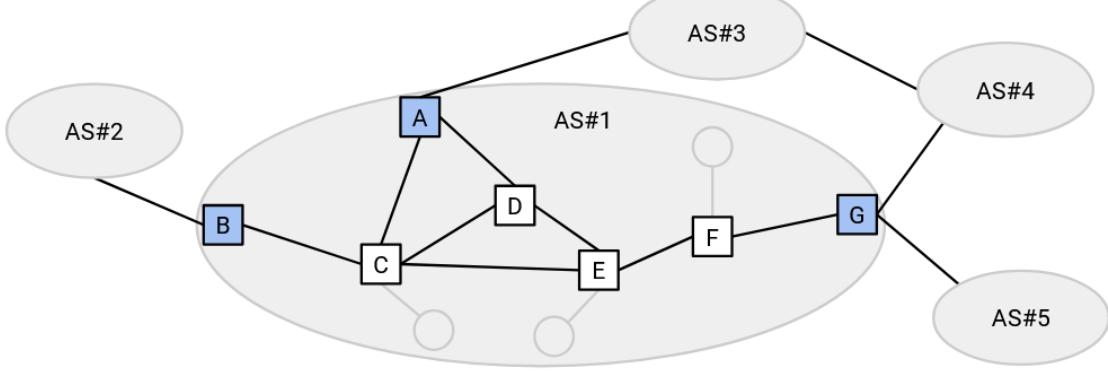


However, in reality, the AS contains many routers (and hosts) connected by links.



In order to actually implement BGP, we need all the routers inside the AS to work cooperatively to act as a single node.

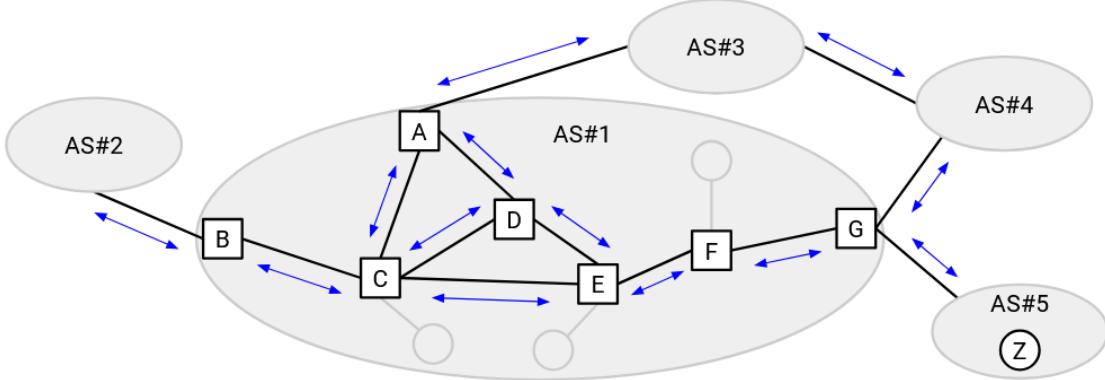
Within an AS, we will classify all routers into two types. **Border routers** have at least one link to a router in a different AS. **Interior routers** only have links to other routers within the same AS.



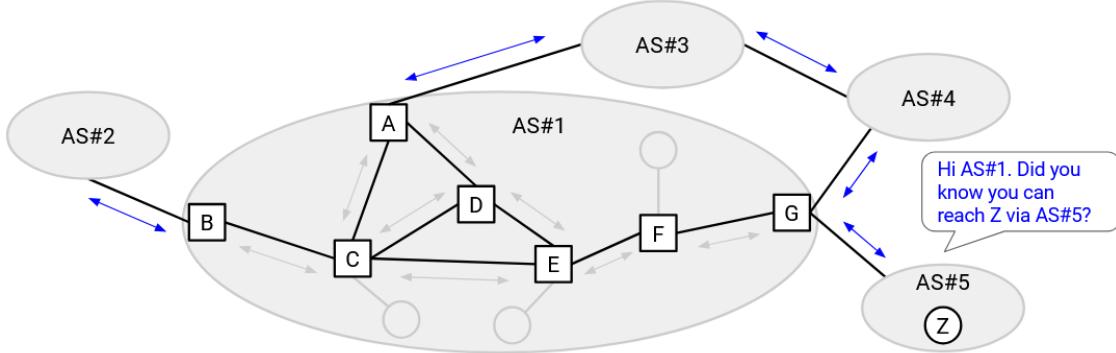
Only the border routers need to advertise routes to other ASes. Sometimes, we call the routers advertising BGP routes **BGP speakers**. The BGP speakers need to understand the semantics and syntax of the BGP protocol (how to read and create a BGP announcement, what to do when receiving an announcement, and so on).

External and Internal BGP Sessions

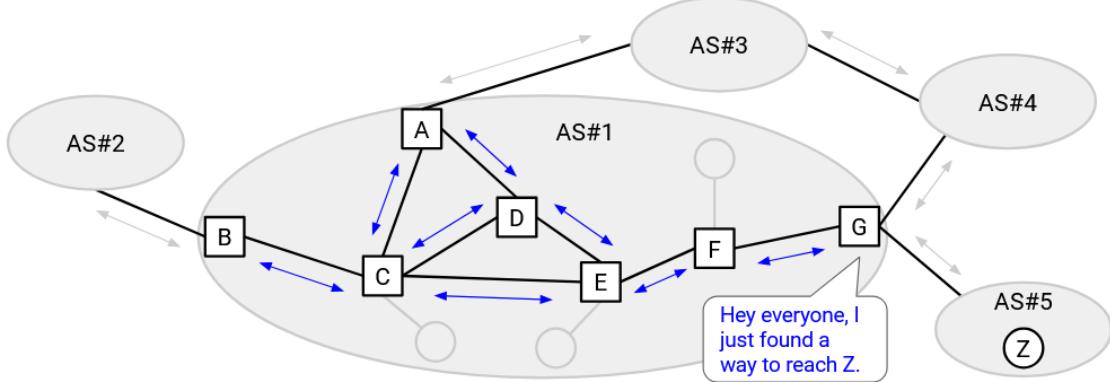
A **BGP session** consists of two routers exchanging information between each other.



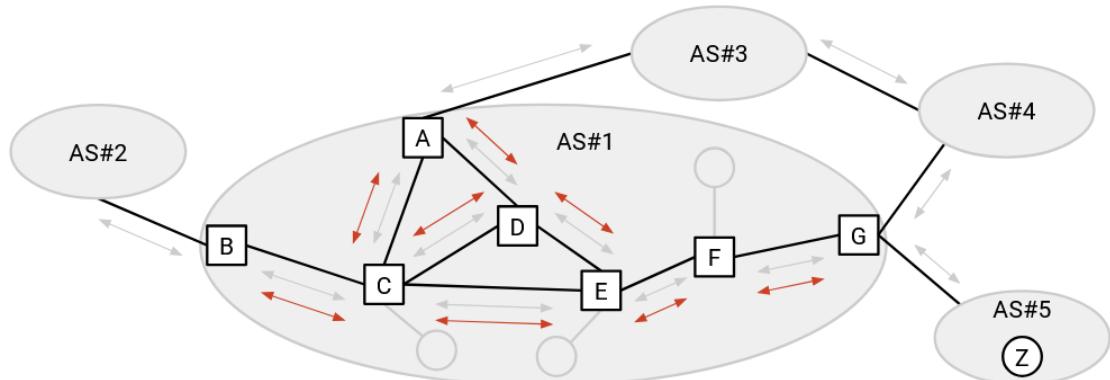
An **external BGP (eBGP) session** is between two routers from different ASes. eBGP sessions can be used to exchange announcements between different ASes and learn about routes to other ASes. Only border routers participate in eBGP sessions (since eBGP requires talking to a different AS).



By contrast, an **internal BGP (iBGP) session** is between two routers in the same AS (not necessarily directly connected by a link). More specifically, if a border router learns about a new route, it can use iBGP to distribute that new route to the other routers in the AS. This allows all the routers in the AS to coordinate and act together as one entity. Both border and internal routers participate in iBGP sessions.



eBGP and iBGP sessions are different from **interior gateway protocols (IGP)**. These are the intra-domain routing protocols (e.g. distance-vector, link-state) that are deployed within an AS to route packets inside the AS.

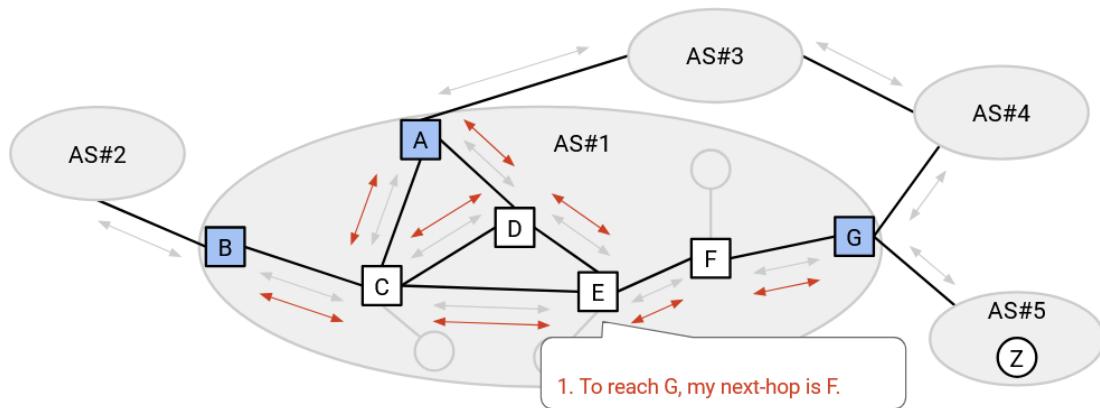


It's easy to confuse iBGP and IGP. Both exchange messages within the same AS. However, iBGP is part of an inter-domain protocol, helping routers learn about paths to other ASes. IGP is an intra-domain protocol, helping routers learn about paths to destinations in the same AS.

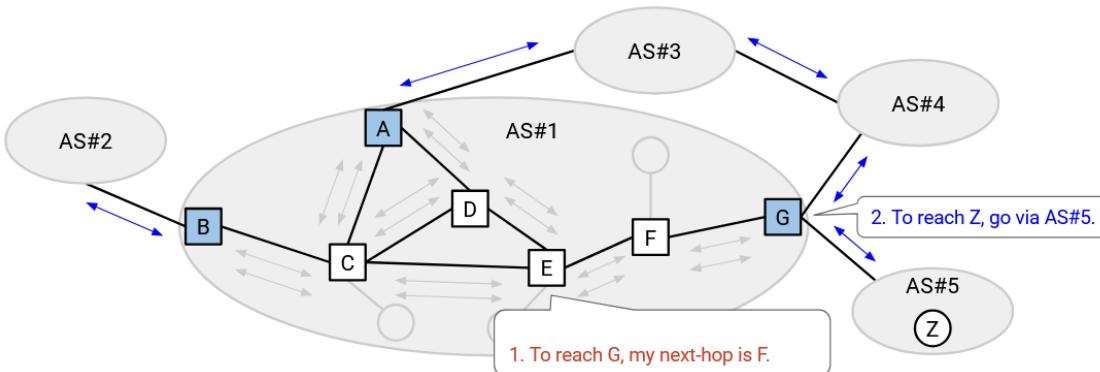


eBGP, iBGP, and IGP work together to establish routes from any one router to any other router in the Internet (even if the routers are in different ASes).

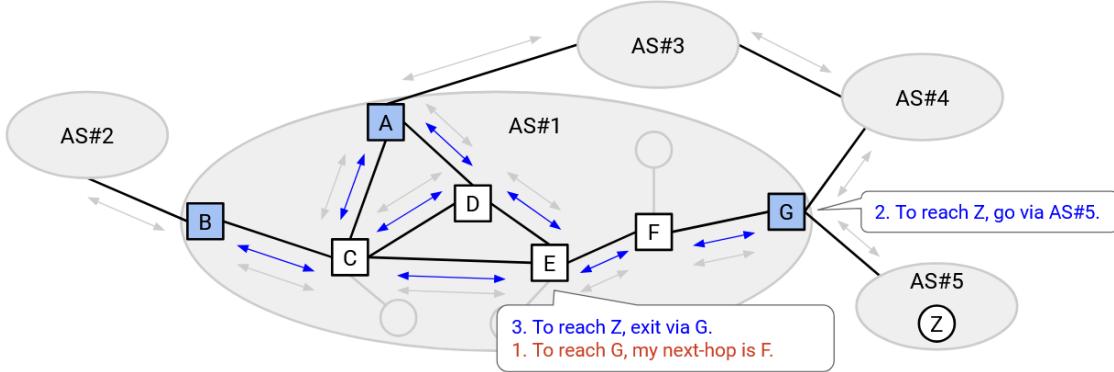
First, each AS runs IGP to learn least-cost paths between any two routers inside the same AS.



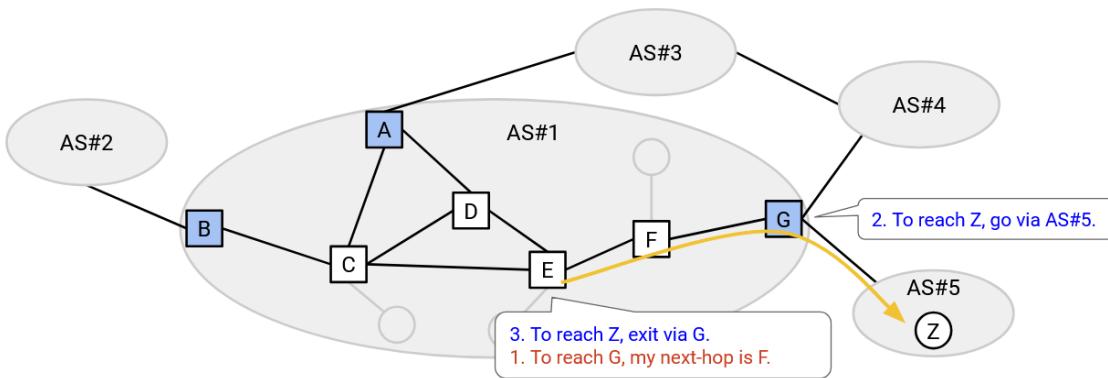
Next, the ASes run eBGP, advertising routes to each other to learn about routes to other ASes.



Finally, the ASes run iBGP, so that a router that has learned about an external route can distribute that route to all the other routers in the same AS.



The routes learned from eBGP, iBGP, and IGP can be used to send packets anywhere in the Internet. If the destination is within the same AS (same IP prefix), we can use the routes learned from IGP to forward the packet. If the destination is in a different AS (different IP prefix), we can think back to iBGP, which told us about any external routes discovered by anybody in my AS. Using the iBGP results, we can figure out which border router is on that external route. Then, we can use IGP to forward the packet to the correct border router (who will then forward the packet to the next AS).

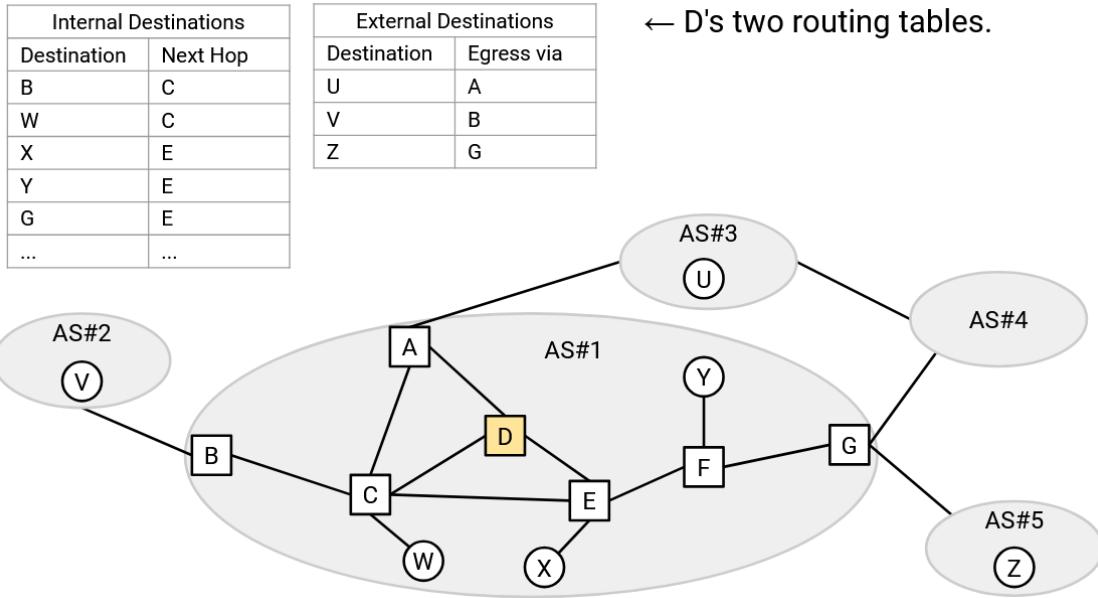


As a concrete example, let's say E wants to send packets to Z. First, every router in E's AS runs IGP, learning all the internal routes. Next, some router in AS#5 advertises a route to Z using eBGP. At this point, only G knows that it can reach Z. Finally, G tells all routers in its own AS that it can reach Z, using iBGP.

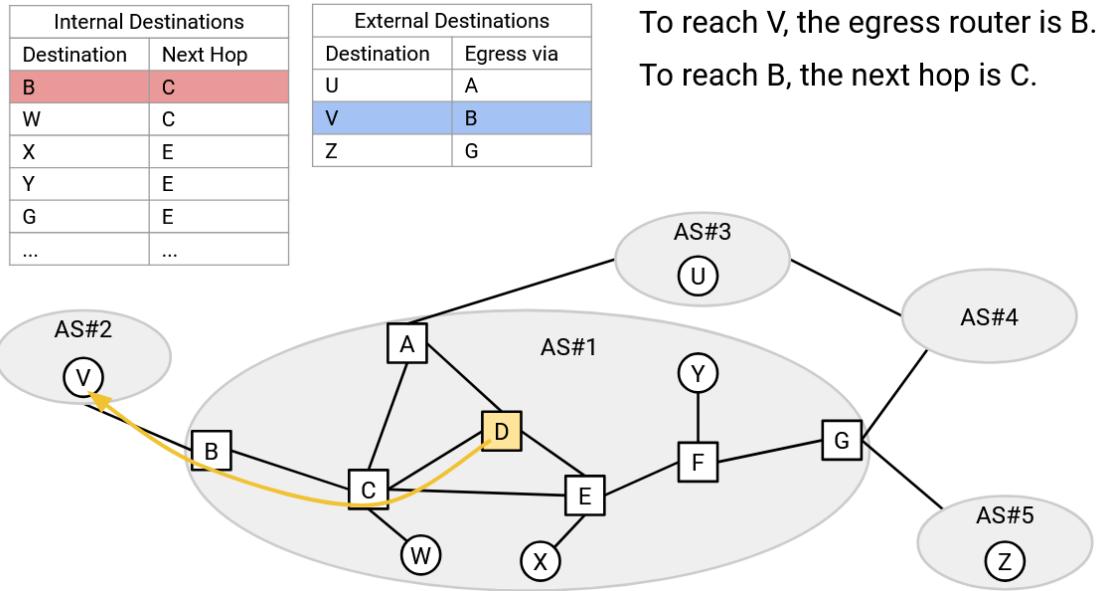
E has heard from iBGP that G, a router in the same AS, can reach Z. Using the IGP routes, E can send the packet to G (forwarding to F first). Then, G can use the route learned in eBGP to send the packet to Z.

The border router who advertises a route to an external destination is sometimes called the **egress router** for that destination. This is the router who can help your packet exit the local network and move to other networks closer to the destination. In the example above, G is the egress router for destination Z.

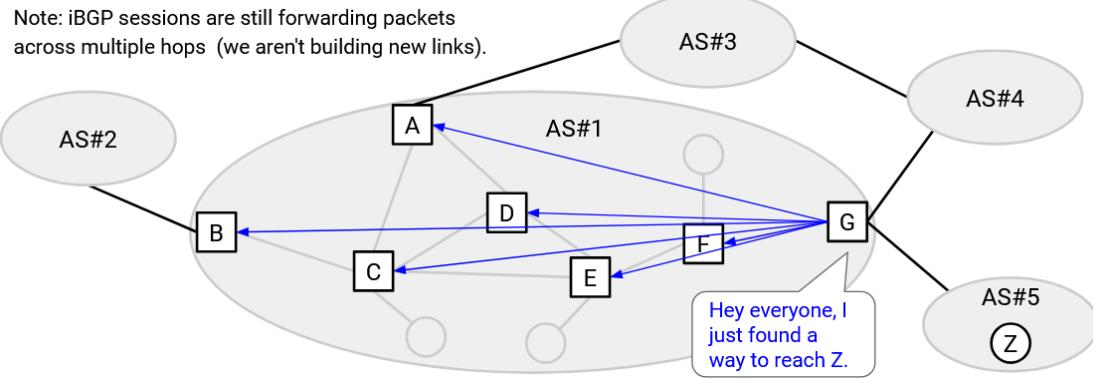
A consequence of these protocols is that every router has two forwarding tables. One is a table mapping all internal destinations (same AS) to a next hop, populated with information from IGP. The other is a table mapping all external destinations to an egress router (who knows a route to the external destination), populated with information from eBGP.



Note that in the eBGP table, the egress router is not necessarily a next hop. The egress router might be several local hops away, but we use IGP to reach that egress router.



We've seen how eBGP (path-vector, advertising routes) and IGP (distance-vector or link-state) are implemented as algorithms. How is iBGP implemented? When a border router installs a new route to a destination, it has to inform the other routers in the AS. One simple solution is to have the border router directly tell every other router in the AS.



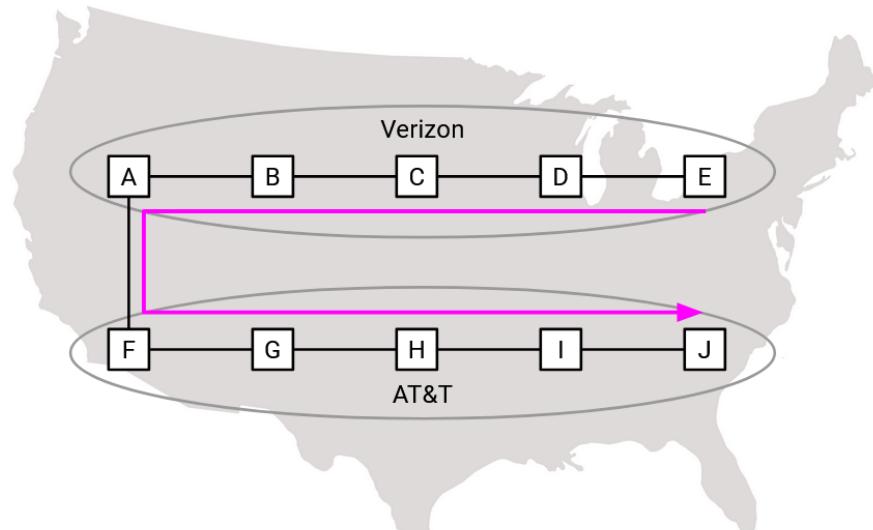
This solution is relatively simple, though it requires every border router to have an iBGP session with every other router. In a network with B border routers and N routers total, this protocol would require BN iBGP connections, and might scale poorly as local networks get larger.

Note: In reality, there are other ways to combine inter-domain and intra-domain routers. You can look up “route reflectors” if you’re interested, though they won’t be covered in this class.

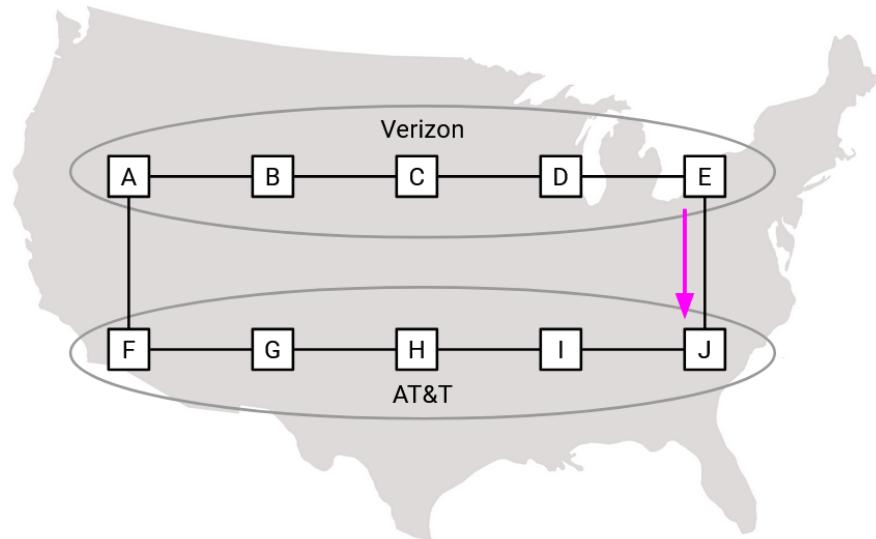
Multiple Links Between ASes: Hot Potato Routing

So far, in our AS graph, we’ve shown two ASes having a single link (edge) between them if they are connected. In practice, because an AS actually consists of many routers, it’s possible for two ASes to be connected by multiple links.

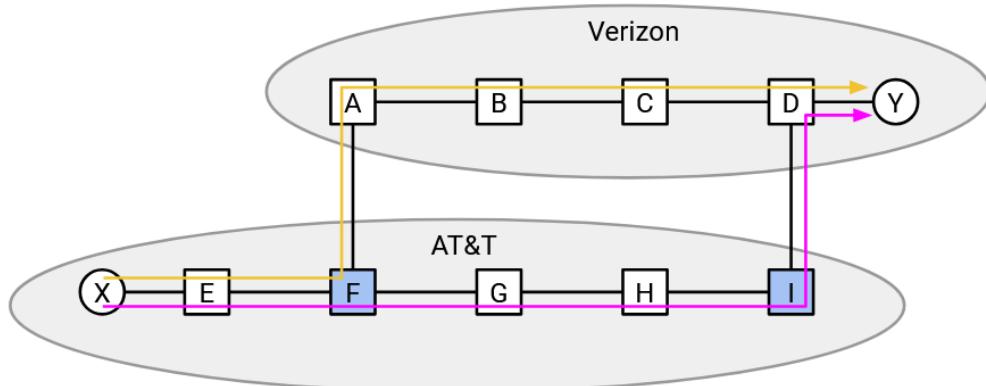
In practice, it can be useful to have multiple links between large ASes. For example, Verizon and AT&T are very large ASes with infrastructure across the entire United States. Suppose there was only one link between the two ASes on the west coast. If a Verizon router in the east coast and an AT&T router in the east coast wanted to communicate, the packet would have to travel across the country on Verizon’s network, traverse the link into AT&T’s network, and then travel back across the country to the destination.



Multiple links between two ASes also means that there can be multiple paths between two routers that pass through the same ASes. At the AS level, both of these paths go through the same ASes, and our earlier model made no distinction between them. However, in our more detailed model, both paths need to be exported, and a preferred route has to be imported.

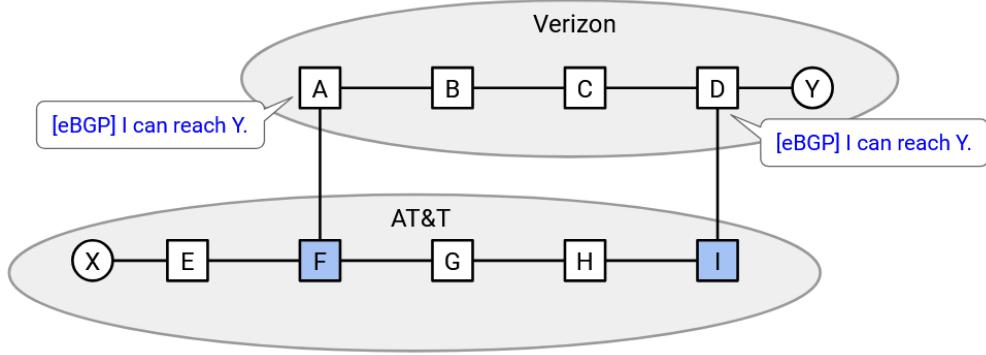


If there are two routes, which route does the importing AS prefer?

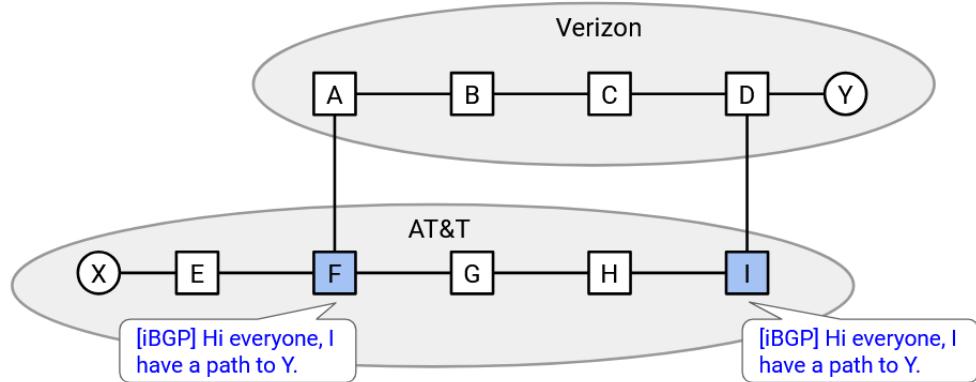


Bandwidth costs money, so I would prefer if this traffic traveled as far as possible on infrastructure owned and paid for by other people, and traveled as little as possible on my own infrastructure. Therefore, the orange path is preferred.

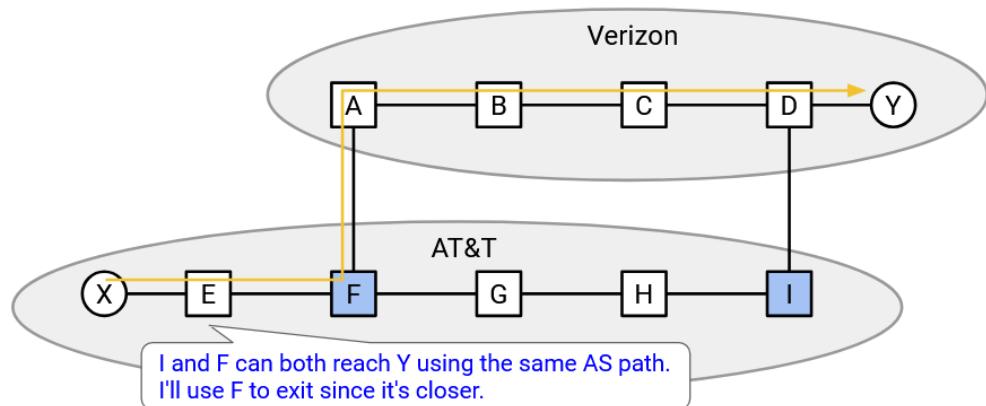
More formally, the importing AS receives two announcements: one from the west router, and one from the east router.



Using iBGP, every router inside the AS sees both announcements. One says, the egress router is the west router, and the other says, the egress router is the east router. Every router has to decide which announcement to import.



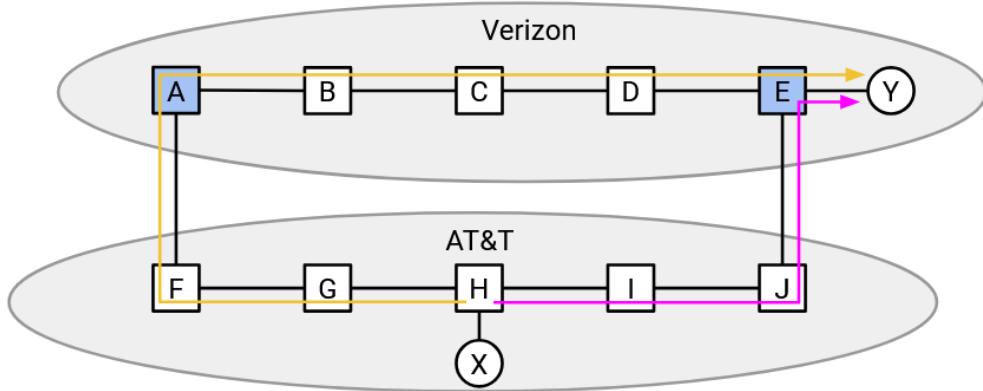
Let's focus on router E. Using IGP, this router can figure out the distance to the west egress router, and the distance to the east egress router. Since the east egress router is closer, routing packets via the east egress router will use up less of this AS's bandwidth. Therefore, this router will import the path via the east egress router. Another router, like one closer to the west egress router, might decide to import a different path.



This strategy of selecting the nearest egress router is sometimes called **hot potato routing**. We want the packet to leave our AS as soon as possible, and start traveling over somebody else's links as soon as possible.

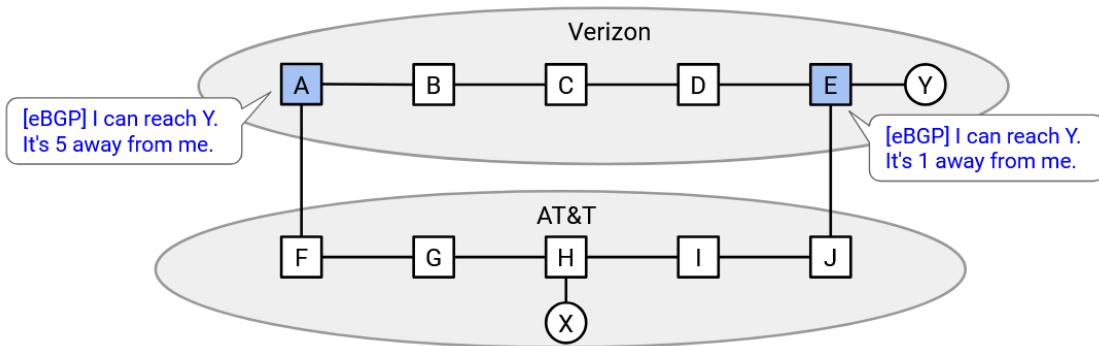
Multiple Links Between Routers: MED

What if a router is equally close to both possible egress routers?

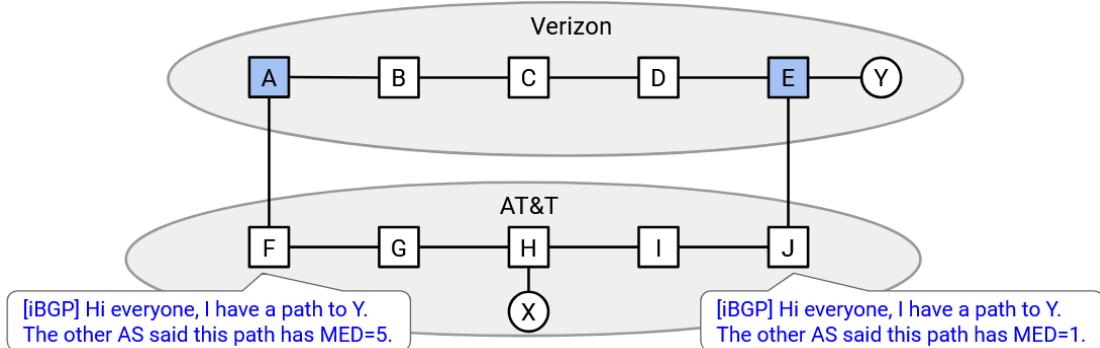


In order to tiebreak, the exporting AS can announce a preference for one route over the other.

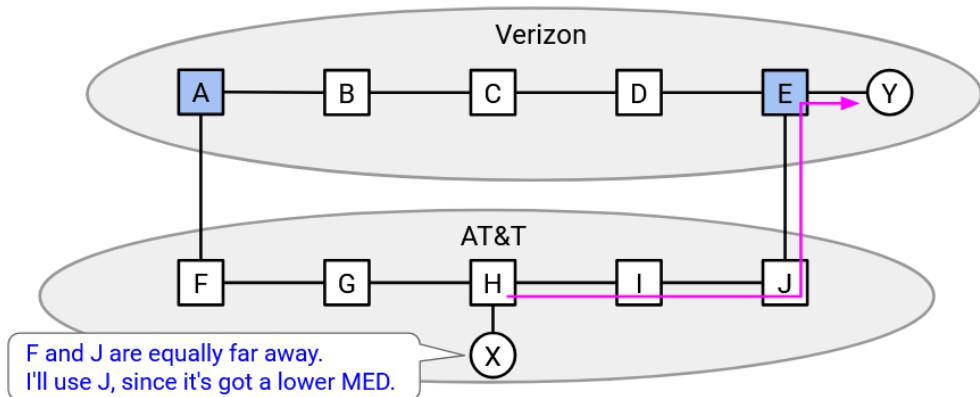
Which route does the exporting AS prefer? Again, since bandwidth costs money, the exporting AS prefers the pink path, which uses less of its bandwidth. In the announcement of the pink path, the exporting AS can additionally say "I prefer if you used this path," and in the announcement of the orange path, the exporting AS can additionally say "I prefer if you avoided this path."



Now, the router that is equally close to both egress routers can see this extra information in the iBGP announcement.



Using this extra information, the router can select the egress router on the pink path, since the exporting AS preferred this path.



This additional information in the exporting announcement is called the **Multi-Exit Discriminator (MED)**. From the perspective of the exporter, it indicates my preferred router for entering my network. From the perspective of the importer, it indicates the other AS's preferred router for exiting my network and entering the other AS's network.

Another way to interpret the MED is, the distance to the destination, via this router. The exporter can say, “the west coast router is 3 hops away from the destination,” and “the east coast router is 12 hops away from the destination.” Lower MED numbers are preferred, since the exporter wants to use as little of its own bandwidth as possible. The exporter would rather use 3 of its own links, instead of 12 of its own links.

Import Policy Priority

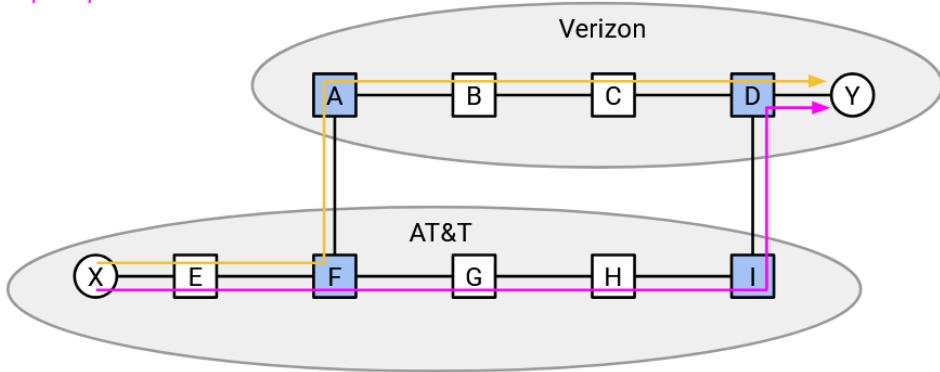
Our more detailed model, where two ASes can be connected with multiple links, means that we now have additional import policy rules, in addition to the Gao-Rexford rules. When you receive multiple announcements for the same destination, select a path based on these tiebreaking rules, in this order:

1. Use the **Gao-Rexford rules**. Select the path advertised by a customer, over the path advertised by a peer, over the path advertised by a provider.

2. If multiple paths have the same Gao-Rexford priority (e.g. two paths from customers), select the **shorter path** (the path passing through fewer ASes).
3. If multiple paths have the same length, select the path with the **closer egress router** (using IGP to find distance to each egress router).
4. If multiple paths have the same distance to egress router, select the path with the **lower MED** (where MED is included in the advertisement).
5. If multiple paths have the same MED, **tiebreak arbitrarily** (e.g. pick the router with the lower IP address).

AT&T prefers gold path.

Verizon prefers pink path.

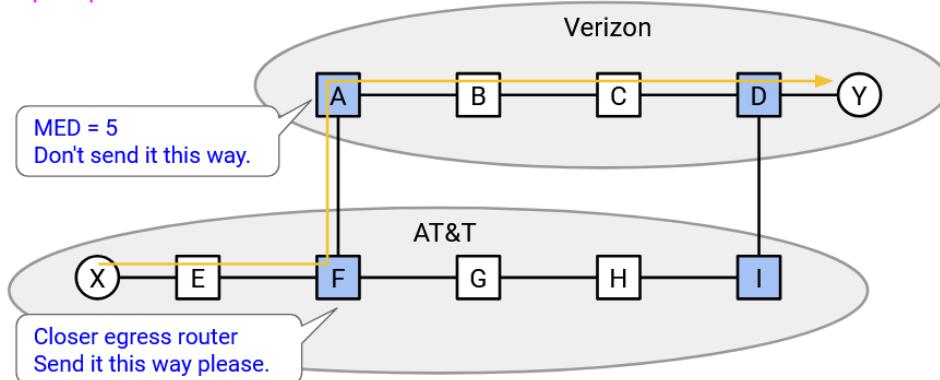


Notice that closest egress router (hot potato routing) and MED are often contradictory. Every AS prefers to minimize their own bandwidth usage, and wants the packet to be carried on other ASes' bandwidth.

As the exporting AS, I want the packet to enter my AS as close to the destination as possible. This means I want the importing AS to carry the packet really far (long path to egress).

AT&T prefers gold path.

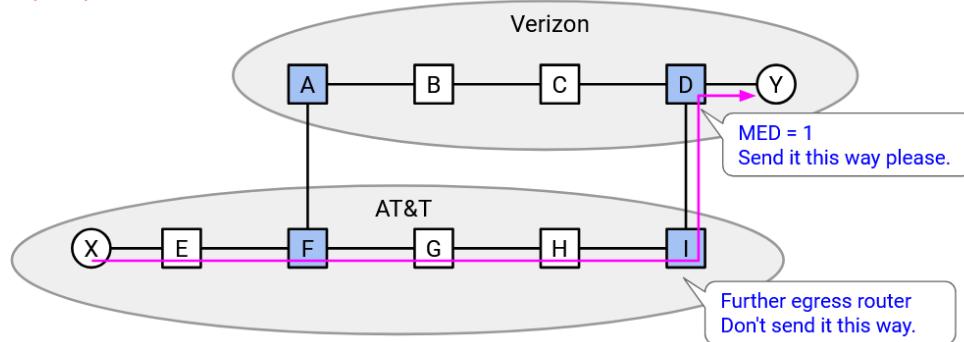
Verizon prefers pink path.



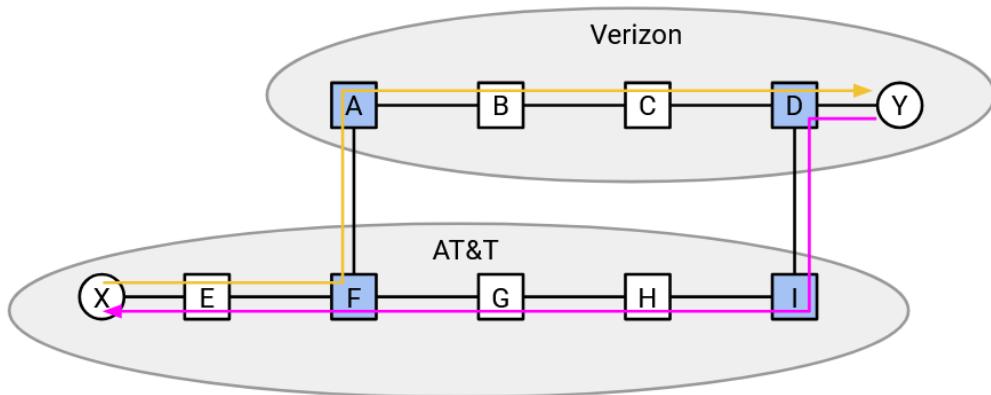
By contrast, as the importing AS, I want to carry the packet as little as possible (short path to egress). This means I want the packet to enter the other AS as far from the destination as possible (force the other AS to do all the work).

AT&T prefers gold path.

Verizon prefers pink path.



One consequence of this contradiction is that paths through the Internet are often asymmetric. If two hosts are sending packets back and forth, the path in one direction might be different from the path in the other direction.



In this example, for eastbound packets, A picks the west egress router and forces B to carry the traffic most of the way. In the other (westbound) direction, B picks the east egress router, and forces A to carry the traffic most of the way.

Fundamentally, BGP allows this behavior because every AS is granted the autonomy to set their own policy (here, that policy is hot potato routing).

In practice, sometimes ASes will try and implement more clever strategies to trick other ASes into carrying the packet further. Or, an AS with better bandwidth might agree to carry your traffic further for you, if you pay a premium fee.

BGP Message Types and Route Attributes

Recall that a protocol must specify syntax and semantics. Specifically, BGP must specify the structure of messages being sent and received. BGP must also specify what a router should do when it receives a message.

There are four different BGP message types. Open messages can be used to start a session between two routers to communicate with each other. KeepAlive messages can be used to confirm that a session is still open, even if messages haven't been sent recently. Notification messages can be used to process errors. We won't describe these first three message types in any further detail.

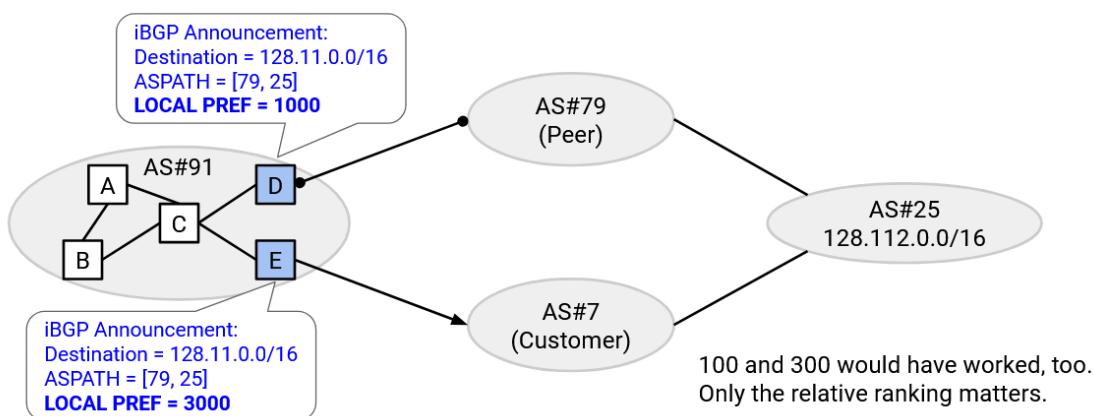
We'll focus on the fourth and most interesting message type, Update. These messages are used to announce new routes, change existing routes, or delete routes that are no longer active.

The Update message contains a destination, represented as an IP prefix. The message also contains **route attributes**, which can be used to encode any useful information corresponding to that IP prefix. The route attributes are a set of name-value pairs, where the name indicates the type of attribute, and the value indicates the value of that attribute. A non-networking example of attributes might be: color=red, shape=triangle. The attribute names are color and shape, and they correspond to values of red and triangle, respectively.

Some attributes are local to an AS, and are only exchanged in iBGP messages. Other attributes are global, and can be sent in eBGP advertisements.

There are many BGP attributes, but we'll focus on three important ones, which are used to encode the different tiebreakers for importing paths.

The **LOCAL PREFERENCE** attribute encodes the Gao-Rexford import rules (top priority tiebreaker) inside a specific AS. An AS can assign a higher value to more preferred routes (e.g. from customers), and a lower value to less preferred routes (e.g. from providers). This attribute is local, and only carried in iBGP messages. This attribute is not sent to other ASes in eBGP announcements, because other ASes don't need to know about this AS's preferences.



As an example, suppose router E receives an eBGP announcement from AS#7, and router A knows that AS#7 is a customer. Then, in the iBGP message, router E can set a local preference value of 3000 (high number). Now, every other router in the same AS knows that router E can reach the destination it's announcing, via

the path in the **ASPATH** attribute, with a local preference of 3000.

By contrast, if router D receives an eBGP announcement from AS#79, and this AS is a peer, then in the iBGP message, router D can set a lower local preference value of 1000 and then distribute this path (with lower local preference) to the other routers in the AS.

The local preference numbers are arbitrary, and only their relative ranking is important. In the example above, the numbers could have been 300 and 100 instead of 3000 and 1000, and the behavior would be the same. The local preference numbers are often set manually by operators.

The **ASPATH** attribute contains a list of ASes along the route being advertised (in reverse order). This attribute is global, and can be sent in eBGP announcements.

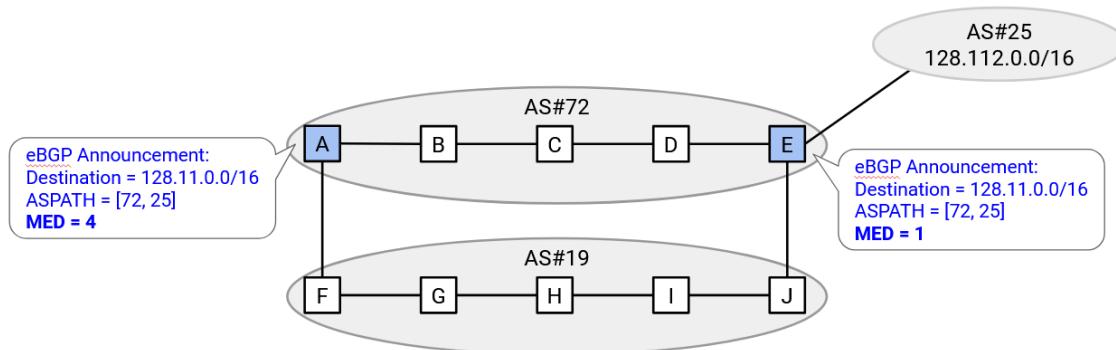


As an example, an announcement would have IP prefix of the destination (128.112.0.0/16), and an **ASPATH** attribute of [3, 72, 25].

The **ASPATH** is the second priority tiebreaker when importing paths. If two announcements have the same local preference (e.g. both are from customers), then we'll select the shorter path. **ASPATH** tells us the length of each path, measured by the number of ASes the path goes through.

If the local preference and path length are tied, the third priority tiebreaker is the IGP cost to the egress router. This cost is stored in the router's local forwarding table (e.g. a local distance-vector protocol would store the cost to every other router in the same AS).

The **MED** attribute encodes the preferences of the exporting AS. Equivalently, this attribute represents the distance from the exporting router to the destination (lower numbers are preferred).



For example, if there are two links between these two ASes, both border routers from the exporting AS will announce a path. The **ASPATH** and destination are the same, since the path of ASes to the destination is

the same in both cases. However, the west router will include a lower MED attribute number, than the east router. This says: when possible, please route packets for the destination through my west router (lower number), because this router is closer to the destination.

If the local preference, path length, and distance to egress router are all tied, the fourth priority tiebreaker is the MED number inside each announcement.

Issues with BGP

BGP has no built-in security guarantees. A malicious AS could lie and advertise a route to a destination, even if the AS cannot reach that destination. A malicious AS could also advertise a very cheap route to a destination, even if that cheap route doesn't actually exist. This could encourage other ASes to route packets through the malicious AS, where the attacker could delete or modify packets passing through the malicious AS. These attacks are called **prefix hijacking**. There is active research on using cryptography to secure BGP, though such protocols are not widely deployed.

BGP prioritizes policy over least-cost when selecting paths. Also, because BGP measures path length in terms of the number of ASes, the path length can be misleading (e.g. one AS could contain 2 routers or 200 routers along the path being advertised). This can lead to issues where packets don't always take least-cost paths, and it's difficult to reason about performance on the Internet. Some might classify these as issues, though they may be more of an intentional design trade-off. The designers of BGP made a conscious design choice to prioritize policy and hide the internal topology of an AS, at the expense of performance.

BGP is complicated to implement. There are many subtle implementation details that we didn't cover. Even in the topics we covered, certain configurations like local preference or MED numbers have to be manually set by the operator, and incorrect configurations could lead to incorrect paths spreading through the network. BGP misconfigurations can often lead to Internet outages, and there is active research on tools to verify that BGP is properly configured.

BGP requires certain assumptions (everybody is following the Gao-Rexford rules, AS graph forms a hierarchy, no provider-customer cycles) in order to guarantee reachability and convergence. If these assumptions don't hold (e.g. an AS chooses its own policy that violates Gao-Rexford), BGP can produce unstable behavior, where routes never converge, or cycles and dead-ends appear.

IP Header

IP Header Design Goals

Recall that a protocol like IP consists of syntax and semantics. The syntax determines what fields are in the IP header, and the semantics determine how those fields are processed.

Also, recall that the IP packet consists of a header and payload. The header contains relevant metadata that the IP protocol can process. The payload contains any data that will be passed up to higher-layer protocols, and is not parsed by the IP protocol.

Finally, recall that headers are added as we move down the stack, and stripped away as we pass packets up the stack. IP headers are processed at both end hosts, and at every intermediate router.

The IP header should be as small as possible. Every packet sent across the Internet need the IP header attached, so increasing the IP header size, even by one byte, would significantly increase the total amount of bandwidth across the Internet.

The IP header should be as simple as possible. Every router and end host has to process IP packets as they're sent and received, so a header that's complicated to process would slow the entire Internet down. Ideally, we'd like the header to be processed purely in hardware, so we can't assume we have access to general-purpose CPU operations when processing this header.

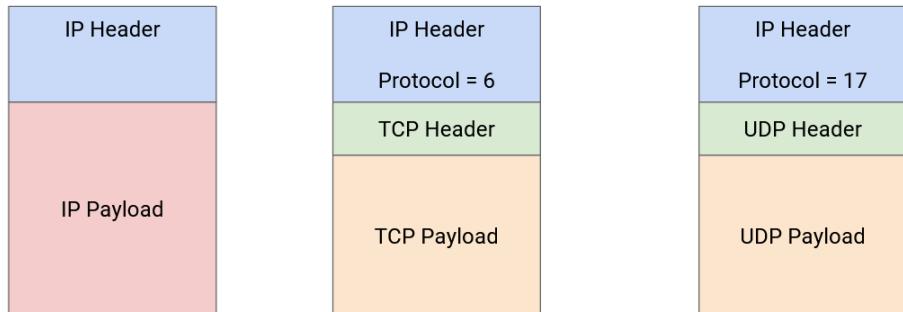
IP Header Fields

An IP protocol needs to do four things:

Everybody (end hosts, routers) need to be able to **parse** the packet and understand what the bits mean. To support this, the header will include the **IP version** (4-bit value), the **header length** (4-bit value, measured in 4-byte words, required because the IP header length is not fixed), and the **packet length** (16-bit value, measured in bytes).

Routers (not end hosts) need to **forward** the packet to the next router. To support this, the header will include the **destination IP address** (32-bit value).

End hosts (not routers) need to **pass the packet up** to higher layers. To support this, the header will include a **protocol number** (8-bit value), which tells us which Layer 4 protocol (TCP or UDP) should be used to process the payload. For example, a protocol number of 6 says to use the TCP protocol to read the remaining payload (reading the first bits of the payload as the TCP header, and so on). A protocol number of 17 corresponds to the UDP protocol.



End hosts and routers need to be able to **send replies** back to the source. To support this, the header will include the **source IP address** (32-bit value).

IP Error Handling

End hosts and routers also need to be able to **specify problems or special cases** in case a packet needs additional handling.

IP packets can be stuck in loops (e.g. if the routing protocol hasn't converged yet). One possible option is to let the packet loop indefinitely until routes converge, but packet forwarding happens on nanosecond scale, and routing convergence happens on the millisecond or second scale. Letting the packet loop until routes converge can take a long time and waste a lot of bandwidth. To prevent indefinite looping, the IP header has a **time to live (TTL)** (8-bit value), which is decremented at each hop. If the TTL reaches 0, the packet is discarded, and an error message is sent back to the source. (The error message is required by the IP specification, though is not always sent in practice.)

IP packets can be corrupted (e.g. bits on the wire can be corrupted from electrical processes). To detect corruption, the IP header contains a **checksum** (16-bit value), and discards packets if the checksum is incorrect.

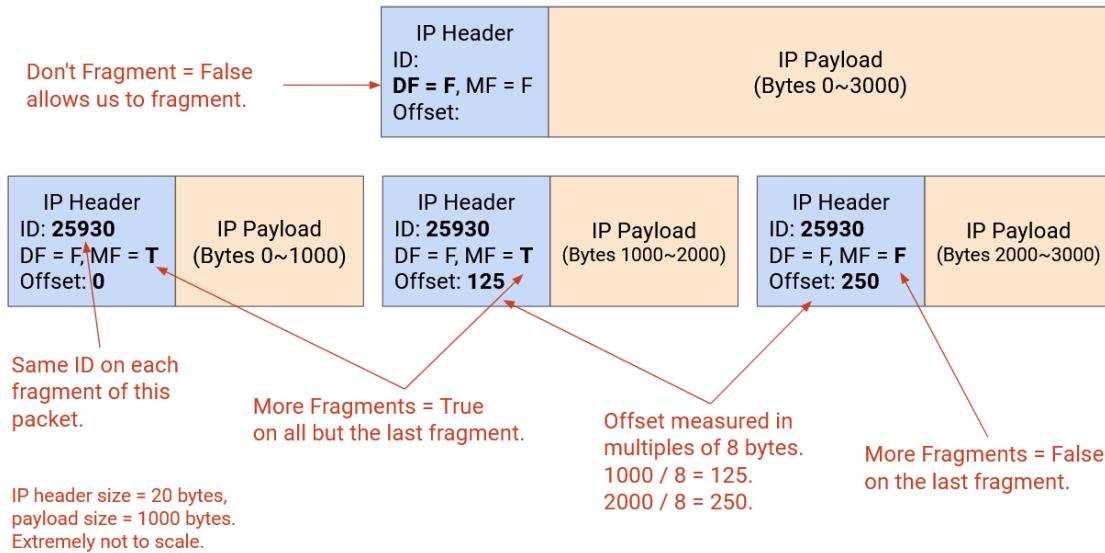
Note that the IP checksum is only computed over the IP header. The checksum can only detect errors in the IP header, not errors in the IP payload. This reflects the end-to-end principle, where we enforce that the payload is checked by the end host, not the intermediate routers.

The IP checksum is updated at every router, because the TTL changes, and the checksum has to be recomputed. One possible alternative design is to exclude the TTL in the checksum, to save routers the extra work.

IP packets could be too large for a specific link. Each link has a **maximum transmission unit (MTU)**, indicating the largest packet size (in bytes) that link can carry as one unit. For example, the link might have limited memory for remembering a packet while it sends the bits along the wire.

The end host doesn't know which links will be carrying the packet, so the end host might send a packet that's too large for one of the links. To solve this, a router can perform **fragmentation**, splitting the packet into multiple fragments, which the router on the other end of the link must reassemble to recover the

original packet. The identification (16-bit), flags (3-bit), and offset (13-bit) fields in the header are used to implement fragmentation.



Fragmentation is achievable in hardware (e.g. a router can quickly fragment packets without punting the packet for special handling), but it introduces extra overhead. The modern Internet avoids fragmentation whenever possible. For example, we try to standardize the MTU as much as possible (a modern standard is 1500 bytes).

The early designers of IP did not fully embrace best-effort design, and thought it might be useful to allow applications to send packets of different types based on the application's needs. To implement this, the IP header has **Type of Service (ToS)** bits (8-bit value), which can be used to request different forms of packet delivery. For example, some packets can be marked as delay-sensitive or high-priority. Over the years, these bits have been redefined to represent different protocols, and ToS no longer exists in its original form. Instead, these bits now represent some notion of priority. Examples of protocols using these bits are Differentiated Services Code Point (DSCP), which defines certain classes of traffic, and Explicit Congestion Notification (ECN), which will help with traffic congestion (discussed later).

In the original IP design, additional **option bits** can be added to the IP header to request more advanced processing on the packet. For example, the sender can request routers to record the route that the packet is taking (e.g. for diagnostics). The sender could include a source route in the packet header and force the packet to travel a certain route. The packet header could also include a timestamp. In modern implementations, these options are almost always disabled, because they lead to unnecessarily complicated implementations that increase the packet processing overhead. For example, these options force the IP header to be variable-length, which is harder to process than a fixed-length header.

1. Parse the packet. (both router and destination)
2. Forward packet to the next hop. (router only)
3. Tell the destination what to do next. (destination only)
4. Send responses back to the source. (both router and destination)
5. Handle errors. (both router and destination)
6. Specify any special packet handling. (both router and destination)

Version (4)	Hdr len (4)	Type of Service (8)	Total Length in Bytes (16)	
		Identification (16)	Flags (3)	Fragment Offset (13)
TTL (8)	Protocol (8)		Header Checksum (16)	
		Source IP Address (32)		
		Destination IP Address (32)		
		Options (if any)		
		Payload		

IPv6 Header Changes

IPv6 was motivated by the concern that we would eventually run out of 32-bit IPv4 addresses. IPv6 **expanded addresses** so that addresses are 128 bits long. The number of possible IPv6 addresses is astronomically large (think: number of atoms in the universe), so we will almost certainly never run out of IPv6 addresses.

The designers of IPv6 took the opportunity to clean up and modernize the IP header, removing and updating fields that are outdated. Originally, IPv6 was intended to be a more ambitious protocol with many new addressing features, but most of these features were never realized. In practice, besides this “spring cleaning” removal of outdated features, there weren’t many significant changes to the protocol from IPv4, so the result is a more elegant IP protocol, without many ambitious changes.

Note: In case you’re curious, IPv5 was published in 1990 (before IPv6 in 1998). It was an experimental protocol that was never widely implemented.

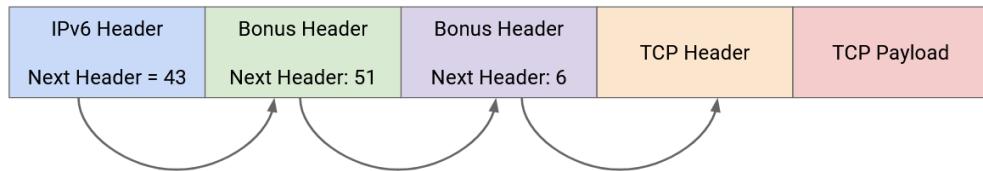
IPv6 **eliminates checksums** in the IP packet header. The argument in favor of including a checksum is: if a packet is corrupted and is not detected, the corrupt packet continues being sent, wasting bandwidth. Including the checksum ensures that the packet is dropped and bandwidth is not wasted on a corrupt packet. In modern times, bandwidth is less of a bottleneck, so the checksum is no longer necessary, and it’s not a huge performance impact if some corrupt packets are sent all the way through the network.

IPv6 **eliminates fragmentation**. If an IPv6 packet is too large for a specific link, the router will drop the packet and send an error message back to the source with the maximum allowable packet size (MTU). The original sender is responsible for splitting up the data into smaller packets and re-sending those smaller packets. End hosts (e.g. your personal computer) process fewer packets than routers (e.g. in data centers), so transferring the workload of fragmentation from routers to end hosts improves the overall scalability of the Internet.

IPv6 replaces the variable-length options section with a modified implementation of the protocol field. In

IPv4, options were problematic because they created variable-length headers, which are harder to parse. In IPv6, the header is fixed in length. This also means that the **header length** field can be eliminated.

In order to continue supporting options, IPv6 generalizes the protocol field to allow the IP packet to be passed up for special processing before reaching Layer 4. (Recall, the protocol header in IPv4 is set to either 7 or 19, to indicate which Layer 4 protocol processes the packet next.) The field is renamed from protocol to **next header** in IPv6.



If you want an additional protocol to process the IP packet, you can put that protocol's corresponding number in the next header field. The designers and users of these extra protocols need to agree on which numbers correspond to which protocols, and a standards body organization needs to manage these numbers. Then, the payload can be passed to the additional protocol, which can read an additional header (after the Layer 3 IPv6 header, but before the Layer 4 header) and perform additional processing, before passing the remaining payload to Layer 4.

If the packet has no additional options, then the next header field is the same as the old protocol field, allowing the IP packet to be directly passed up to a Layer 4 protocol with no further processing.

The idea of next headers can be generalized to allow multiple protocols to process the packet after IPv6, but before Layer 4. For example, IPv6 could have a next header for special processing. Then, the special processing protocol's header can also contain a next header field, which either specifies a Layer 4 protocol, or yet another special processing protocol. This approach is future-proof, because it supports future protocols that haven't been invented yet. Those future protocols can be added in this next-header approach, without breaking IPv6 or requiring an update to IPv6.

IPv6 adds a **flow label** field to the header. At layer 3, packets are sent independently (how one packet is sent doesn't affect other packets), but in practice, it's common for many packets to be related in some way. For example, in a video stream between two hosts, there can be many packets being sent between the same two applications. Layer 3 is supposed to treat these packets separately, but in practice, routers have added more advanced systems called **middleboxes** (e.g. firewalls, intrusion detection systems) that might care about the fact that these packets are part of the same flow, or connection. For example, a firewall might need to read multiple packets from a connection to decide whether that connection should be allowed or blocked. When all packets are sent independently, these middleboxes have to guess whether two packets are related or not (e.g. it notices packets with the same source/destination IP address). IPv6 adds an explicit way to denote that multiple packets are related.

Renamed and moved:

- Type of Service → Traffic Class
- Total Length → Payload Length
- TTL → Hop Limit

1. Eliminate checksums.
2. Eliminate fragmentation.
3. Eliminate options, add next header.
4. Add flow label.

Version	Hdr Len	Type of Service	Total Length in Bytes						
Identification		Flags	Fragment Offset						
TTL	Protocol	Header Checksum							
Source IP Address (32 bits)									
Destination IP Address (32 bits)									
Options (if any)									
Payload									

Vers	Traffic Class	Flow Label		
Payload Length	Next Header	Hop Limit		
Source IP Address (128 bits)				
Destination Address (128 bits)				

The version number is unchanged between IPv4 and IPv6. The packet length is unchanged (though renamed from Total Length to Payload Length). TTL is renamed to Hop Limit, though the functionality is unchanged.

The Type of Service bits are renamed to Traffic Class, and can still be used to implement some notion of packet priority.

In general, IPv6 embraces the end-to-end principle and asks the end host to do the work (fragmentation, verifying checksum and re-sending corrupt packets) when possible. Some fields, like the hop limit or TTL, are fundamentally an IP-level problem, and can't be implemented by end hosts. (How would the end host help with a packet looping through the network?)

IPv6 also tries to simplify the header (removing variable-length options), while still allowing extensibility for future improvements (next-header approach, flow label).

IP Header Security

IP does not have any built-in security against attackers. An attacker could send a packet with an incorrect source IP address, allowing the attacker to impersonate somebody else. This might cause the impersonated host to be wrongly blamed for a packet. Or, if the attacker sends a spoofed packet, the reply may be sent to the impersonated host. Lying about the source address is known as **IP spoofing**.

IP spoofing can be used for denial-of-service (DoS) attacks. A DoS attack can be used to overwhelm a server and cause it to crash by flooding the server with packets. If all the packets came from the same sender, the server could stop the attack by ignoring packets from the attacker's IP address. However, if the attacker lies about the source IP address, the server has a harder time distinguishing attacker traffic from legitimate traffic.

More sophisticated attacks involving spoofing exist, though we won't cover them in detail in this class (see the UC Berkeley CS 161 notes for more details).

The ToS field in the IP header allows the sender to set a priority on their packets. If we allow everybody

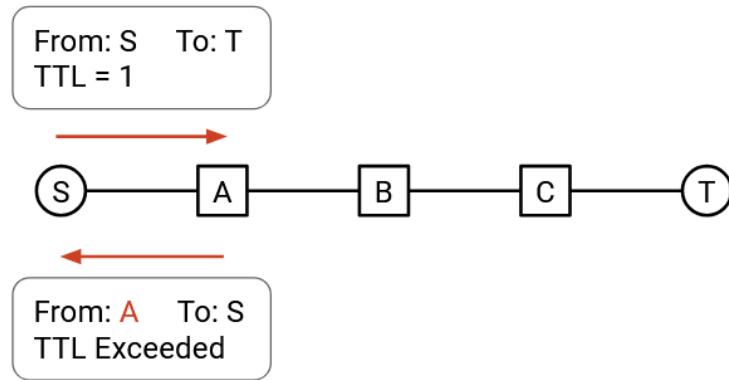
to set their own priority, malicious users can set higher priorities and trick the network into prioritizing attacker traffic.

If the network charges an extra fee for high-priority traffic, the attacker could send a spoofed high-priority packet, and the impersonated host would have to pay for the attacker's traffic.

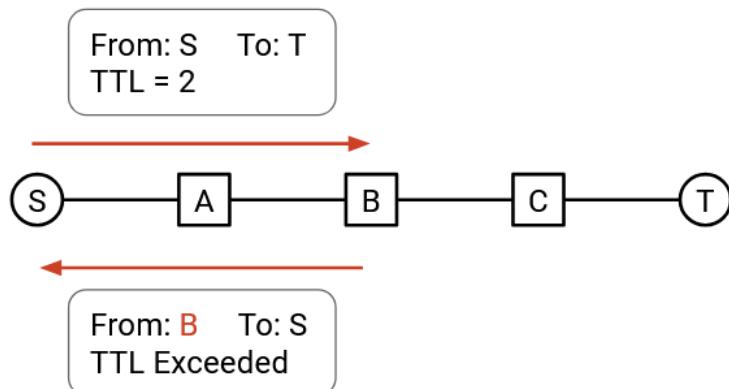
The original Internet design did not stop these attacks, though modern ISPs (Internet service providers) have implemented additional security measures to mitigate IP layer attacks. In the modern Internet, ISPs don't allow end hosts to set the ToS field, and many ISPs have tools to detect and block spoofed packets.

In IPv4, attackers could intentionally send large packets, forcing routers to perform extra work fragmenting those packets. Or, attackers could intentionally add extra options, forcing routers to process those extra options. This could be used to perform DoS attacks and overwhelm a router's processing capacity.

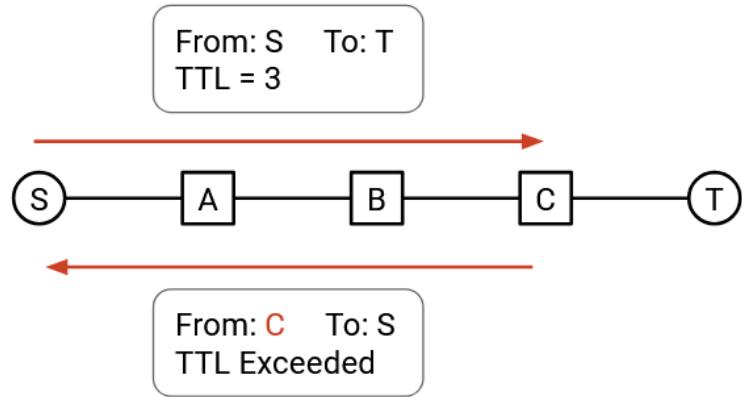
The TTL field can be exploited to learn about the network topology. You could send a packet with TTL 1. The packet will expire at the first hop, and the first router will send you an error message, allowing you to learn the identity of the first router.



Then, you can send a packet with TTL 2, which will expire at the second hop. The second router will send you an error message, allowing you to also discover the second router.



By repeating this with TTL 3, TTL 4, and so on, you can discover all the routers on your path. This attack is known as **traceroute**, though others argue that it's not an attack and is useful for diagnostics.



Repeating this attack on different sources and destinations allows you to learn more of the network topology. Some routers do not send an error message when the TTL is exceeded, which might limit this exploit.

An attacker could theoretically tamper with the protocol or checksum field, but this would likely cause the packet to be dropped because of an invalid protocol or checksum, so practical attacks with these two fields don't really exist.

Source IP address: Spoofing.

Type of service: Prioritize attacker traffic.

Fragmentation, Options: Denial-of-service.

TTL: Traceroute.

Protocol, Checksum: No apparent problems.

Version	Hdr len	Type of Service	Total Length in Bytes	
Identification			Flags	Fragment Offset
TTL	Protocol	Header Checksum		
Source IP Address (32 bits)				
Destination IP Address (32 bits)				
Options (if any)				
Payload				

Transport Layer Principles

Reliability Abstraction and Goals

Many applications require reliability. For example, when sending a file over the Internet, we want the recipient to receive the same bytes in the same order as what the sender sent.

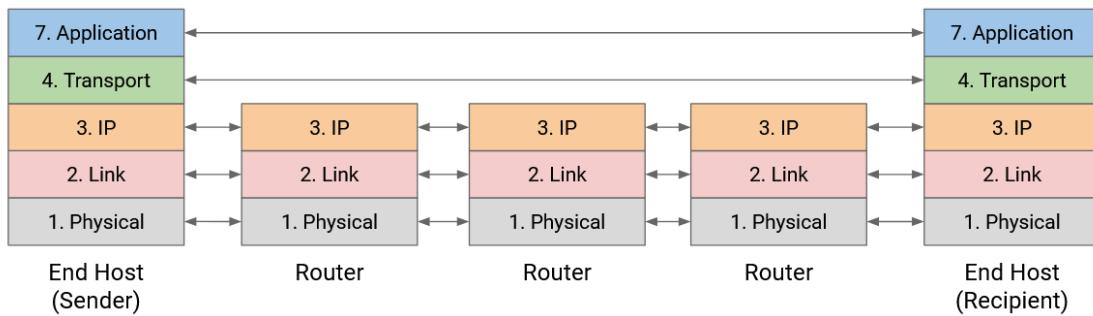
However, Layer 3 only provided unreliable, best-effort packet delivery. Packets can be lost (dropped), corrupted, and reordered (order of packets sent doesn't match order of packets received). Packets can be delayed (e.g. a packet could stuck in a queue waiting to cross a link).

In rare cases, packets can even be duplicated, where the sender sends one packet but the recipient receives multiple copies of that packet. This usually happens if a router along the path encounters an error of some sort. In practice, this error is very rare.

Fun fact: Vern Paxson, UC Berkeley faculty, was one of the first people to discover and report packets being duplicated at the link layer.

We will use Layer 4 (the transport layer) to bridge this gap by developing protocols that rely on the best-effort packet abstraction supported by the network, and provide a reliable abstraction that application developers can use.

For practical reasons (discussed elsewhere), reliability is implemented at the end hosts, not at intermediate routers. Also, reliability is implemented in the operating system for convenience, so that applications don't need to all re-implement their own reliability.



We will formalize reliability by defining **at-least-once delivery**. In this model, the destination must receive every packet, without corruption, at least once, but may receive multiple duplicate copies of a packet. The transport layer will use the best-effort delivery to provide at-least-once delivery. Then, using at-least-once delivery, our protocol can remove duplicates and provide exactly-once delivery to the applications.

Note that reliable delivery does not guarantee that packets will be sent. A computer not connected to the network cannot send data to the destination, no matter what reliability protocol we use. Reliability protocols are allowed to give up and fail to send a packet, but the failure must be reported to the application. The protocol cannot falsely claim to have successfully delivered a packet.

Our protocol should also be efficient. More specifically, our protocol should deliver data as quickly as possible, and our protocol should minimize bandwidth use and avoid sending packets unnecessarily. For

example, we could guarantee that packets arrive by re-sending every packet hundreds of times, but this would violate our requirement of using bandwidth efficiently.

Transport Layer Goals

At the transport layer, our goal is to provide applications with a convenient abstraction that makes developers' lives easier. The transport layer allows application developers to think in terms of connections, instead of individual packets being sent across the network. Ideally, the developers shouldn't need to think about the low-level network details like splitting long data into packets, re-sending dropped packets, timeouts, etc.

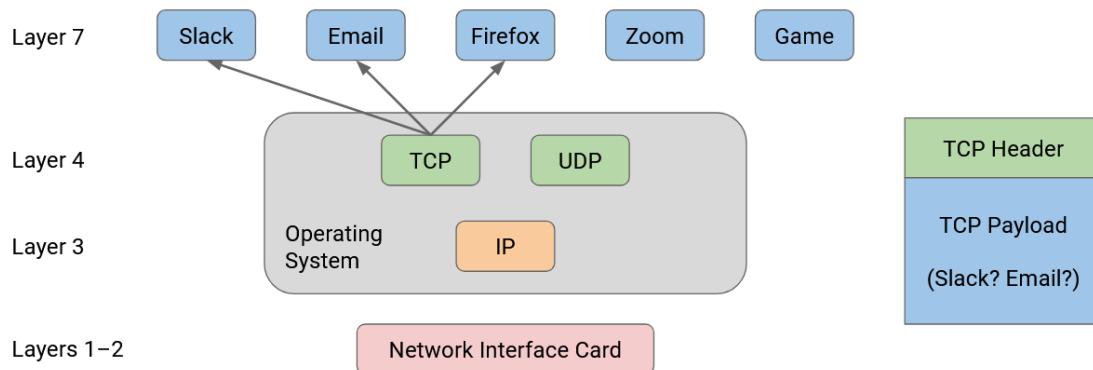
Reliability is just one of several goals we might want to achieve at the transport layer.

The transport layer implements **demultiplexing** between different processes at the end host, by introducing port numbers that can be used to associate each flow (connection) with a different process on the end host.

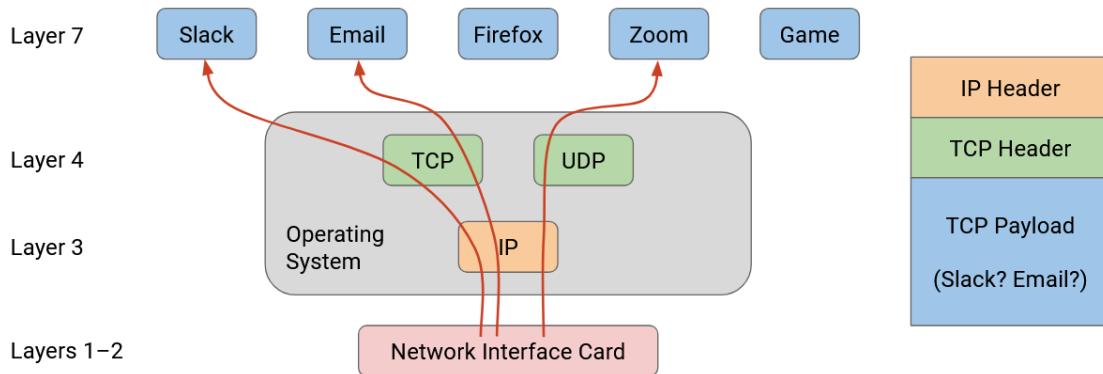
The transport layer also implements flow control and congestion control, which will help limit the rate of packets being sent in order to avoid overloading the receiver and the network, respectively.

Demultiplexing with Ports

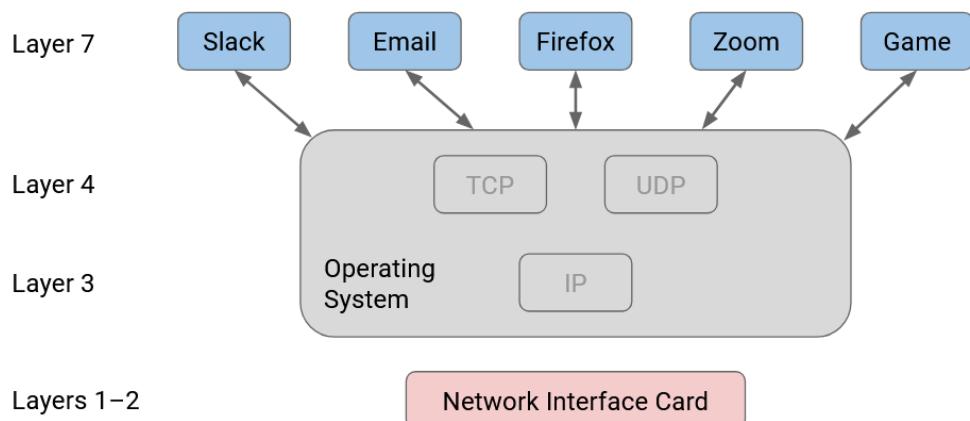
Suppose that my personal computer has two applications that are both talking to the same server. When packets arrive at my personal computer, they have the same source IP address (server), and the same destination IP address (my computer). How can I tell which packets are meant for which application?



In order to distinguish, or **demultiplex**, which packets are meant for which application, the transport layer header includes an additional **port number**, which can be used to identify a specific application on an end host.



When the transport layer receives a packet, it can use the port number to decide which higher-layer application the payload should be sent to. Because the transport layer is implemented in the operating system, these ports (sometimes called **logical ports**) are the attachment point where the application connects to the operating system's network stack. The application knows its own port number, and the operating system knows the port numbers for all the applications, and the matching number is how data is unambiguously transferred between the application and operating system (without getting mixed up with data from other applications).

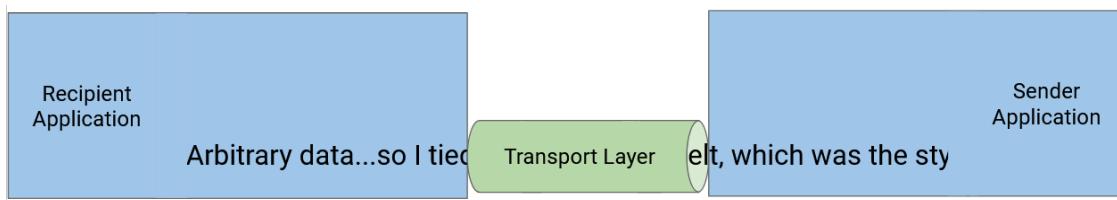


Port numbers are 16 bits long. The modern Internet commonly uses the client-server design, where clients access services, and servers provide those services. Servers usually listen for requests on well-known ports (port numbers 0-1023). Clients know these ports and can access them to request services. For example, application-level protocols with well-known port numbers include HTTP (port 80) and SSH (port 22).

By contrast, clients can select their own random port numbers (usually port numbers 1024-65535). These port numbers can be randomly-chosen, since the client is the one initiating the connection, and nobody is relying on the client having a fixed port number (the client isn't providing services). Client port numbers are **ephemeral** (temporary), because the port number can be abandoned after the connection is over, and does not need to be permanent.

Bytestream Abstraction

Implementing reliability at the transport layer means that the application developer no longer needs to think in terms of individual limited-size packets being sent across the network. Instead, the developer can think in terms of a **reliable in-order bytestream**. The sender has a stream of bytes with no length limit, and provides this stream to the transport layer. Then, the recipient receives the exact same stream of bytes, in the same order, with no bytes lost. You can think of a bytestream as a pipe, where the sender inserts bytes, one by one, into the pipe, and those same bytes appear, one by one, on the recipient's end of the pipe. The sender and recipient don't need to think about re-sending lost packets or packets arriving out of order, because the transport layer protocol will implement that for the developer.



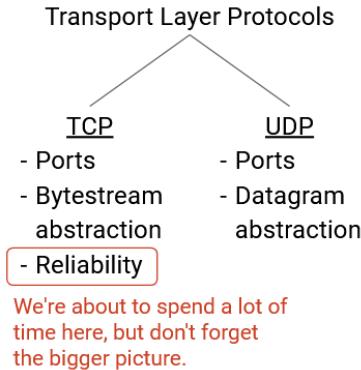
UDP and Datagrams

Sometimes, applications don't need reliability. For example, consider a sensor that reads the water pressure in your home. The sensor sends a reading (small, fixed-size message with the time and water pressure) to the utility company every minute. This system might not need packets to arrive in order (e.g. if the readings already include timestamps), and might not need the ability to split long messages into packets (every reading is small). The system might not even need reliability, as long as most of the readings arrive at the utility company.

Applications that don't need reliability can use **UDP** (User Datagram Protocol) instead of TCP at the transport layer. UDP does not provide reliability guarantees. If the application needs a packet to arrive, the application must handle re-sending packets on its own (the transport layer will not re-send packets). Messages in UDP are limited to a single packet. If the application wants to send larger messages, the application is responsible for breaking up and reassembling those messages. Note that UDP still implements the notion of ports for demultiplexing, though.



At the transport layer, you can choose to use either UDP and TCP depending on your needs, but you can't choose to use both. UDP and TCP are the standard transport layer protocols in the modern Internet.



Other Reliability Designs

TCP was initially implemented by Vint Cerf and Bob Kahn, while they were students at UCLA. They have since been given the Turing Award, Presidential Medal of Freedom, etc. for their work. It's pretty remarkable that the initial TCP design is pretty similar to what is used in practice today, and has stood the test of time. The core ideas of TCP are quite simple, and the design is quite elegant (though not perfect). However, the implementation can be tricky to get right, and the stakes are high, since almost the entire modern Internet runs on TCP.

Since its initial creation, many individual pieces of TCP have evolved (e.g. better algorithms for estimating timers, smarter acknowledgements, smarter ISN selection, congestion control), but the core architectural decisions and abstractions (connection-oriented bytestreams, windows) have remained the same.

TCP is the standard reliability protocol on the Internet, but other fundamentally different approaches exist.

For example, the sender could exploit the idea of redundancy (as seen in error-correcting codes or RAID) to send data more reliably. Instead of sending the user data as-is, the sender encodes the data into more packets with redundancy intentionally built into each packet. For example, the user might have 10 packets, and an algorithm might encode that data into 20 packets. The algorithm might guarantee that as long as any 15 of the 20 packets are received, then the original 10 packets of data can be reconstructed.

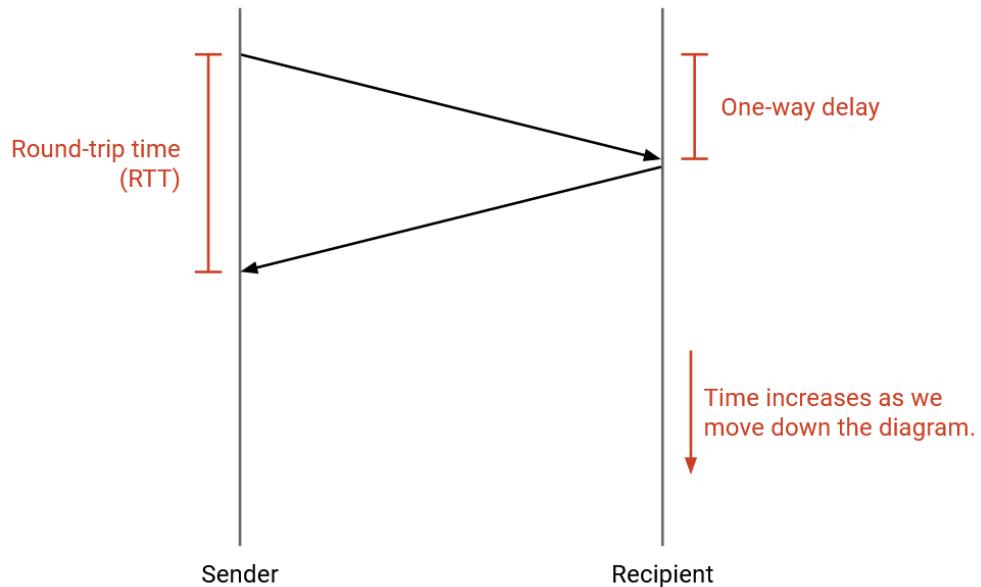
More formally, an encoding algorithm might take k packets, encode them as n packets (where n is greater than k), such that the original k packets can be recovered as long as any k' of the packets are received (where k' is greater than k but less than n).

Coding schemes are a deep topic with many algorithms (e.g. fountain codes, raptor codes), though we will not discuss them any further. They can be seen in practice in video streaming platforms.

TCP Design

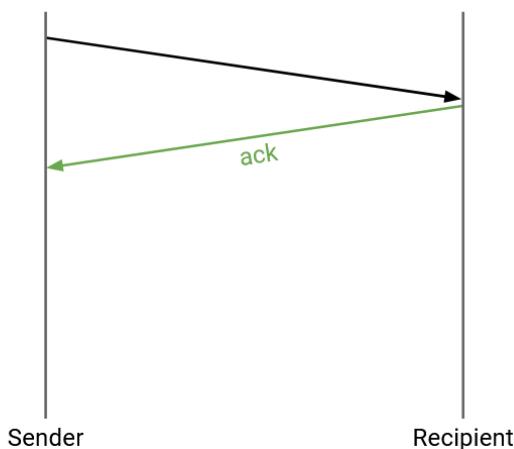
Reliably Delivering a Single Packet

The time it takes for a packet to travel from sender to receiver is the **one-way delay**. The time it takes for a packet to travel from sender to receiver, plus the time for a reply packet to travel from receiver to sender, is the **round-trip time (RTT)**.



Let's build intuition by designing a simplified protocol for reliably sending a single packet.

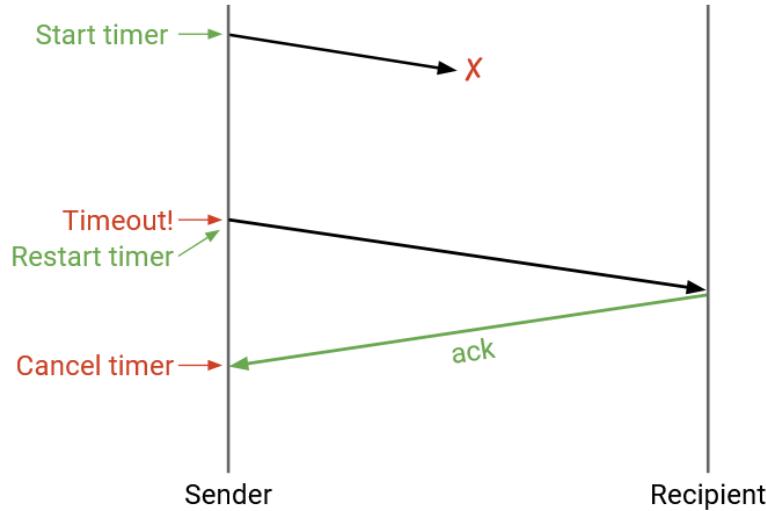
The sender tries to send a packet. How does the sender know if the packet was successfully received?



The receiver can send an **acknowledgment (ack)** message, confirming that the packet was received.

What happens if the packet gets dropped?

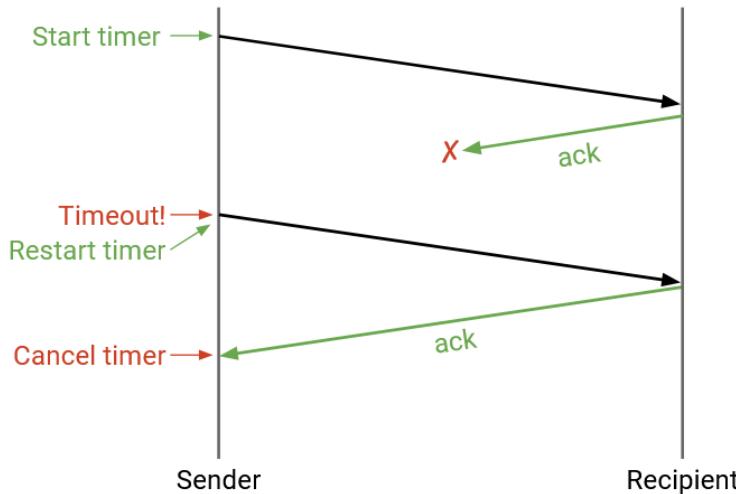
We can re-send the packet if it's dropped. How do we know when to re-send the packet?



The sender can maintain a timer. When the timer expires, we can re-send the packet.

When the sender receives an ack, the sender can cancel the timer and does not need to re-send the packet.

What happens if the ack is dropped?



The protocol still works without modification. The sender will time out (no ack received) and re-send the packet until the ack is successfully sent. In this case, the destination received two copies of the same packet, but that's okay. The destination can notice the duplicate and discard it.

How should the timer be set? If the timer is too long, the packet might take longer than needed to be sent. If the timer is too short, the packet might be re-sent when it didn't need to be. Getting the timer wrong can affect our efficiency goals.

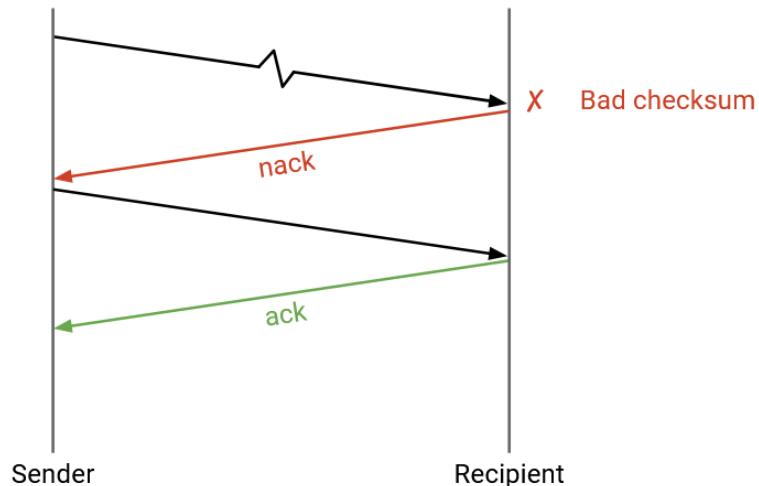
A good timer length would be the round-trip time. This is when the sender expects to receive the ack, so if an ack hasn't arrived by then, the sender should re-send the packet at that time.

In practice, estimating RTT can be difficult. RTTs can vary depending on what path the packet takes through the network, and even along a specific path, the RTT can be affected by the load and congestion along that path.

One way to estimate RTT is to measure the time between sending a packet and receiving an ack for that packet. We can get an estimated RTT measurement from every packet sent, and apply some algorithm (e.g. exponential moving average) to combine these measurements into one RTT estimate. Our algorithm would also have to account for messages being re-sent (variance in the measurements).

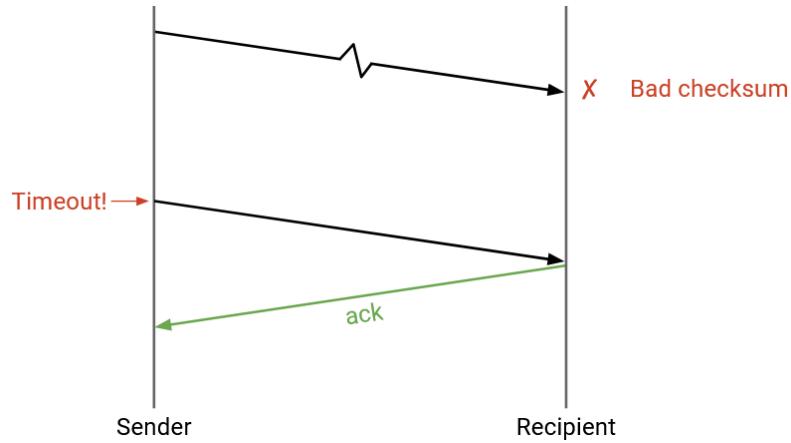
In practice, operators usually err on the side of setting the timer to be longer. If the timer is too short, and timeouts are constantly happening, your connection is probably behaving poorly (constantly re-sending packets).

What if the bits are corrupted?



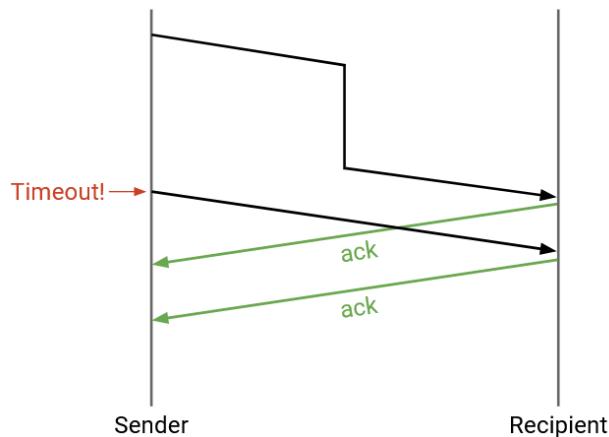
We can add a checksum in the transport layer header (different from the IP layer checksum). When the receiver sees a corrupt packet, it can do two things: Either the receiver can explicitly re-send a **negative acknowledgement (nack)**, telling the sender to re-send the packet.

Or, the receiver can drop the corrupt packet and do nothing (don't send an ack or nack). Then, the sender will time out and re-send the packet.



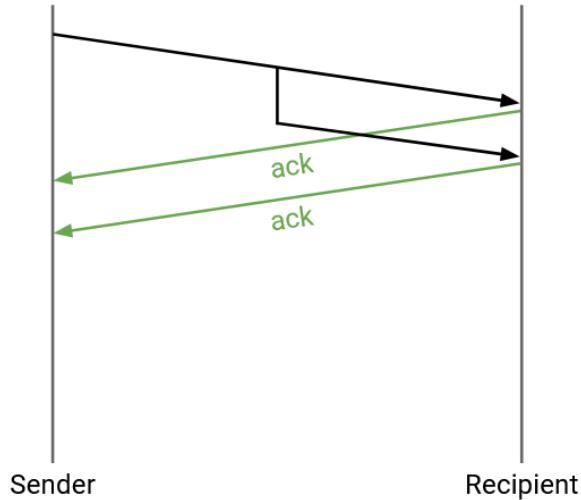
Both approaches (nack or wait for timeout) work, though TCP uses the latter (wait for timeout) and does not implement nacks.

What if the packets are delayed?



No modifications are needed. If the delay is very long, the sender might time out before the ack arrives. The sender will re-send the packet (so the recipient might get two duplicates), and the sender might get two acks, but that's okay.

What if the sender sends one packet, but it's duplicated in the network, and the recipient receives two copies?



No modifications are needed. The recipient would send two acks, but both the sender and the recipient can safely handle duplicates.

Note: From this simplified protocol, we can see that sometimes the recipient receives two copies of the packet. If a specific link was implementing a reliability protocol, the recipient side of the link might receive two copies. Normally, the duplicate would get dropped and only one packet would be forwarded to the destination. But, if the router crashes and restarts in between the two copies arriving, the router might forward both copies to the destination.

In summary, the single-packet reliability protocol is:

If you are the sender: Send the packet, and set a timer. If no ack arrives before the timer goes off, re-send the packet and reset the timer. Stop and cancel the timer when the ack arrives.

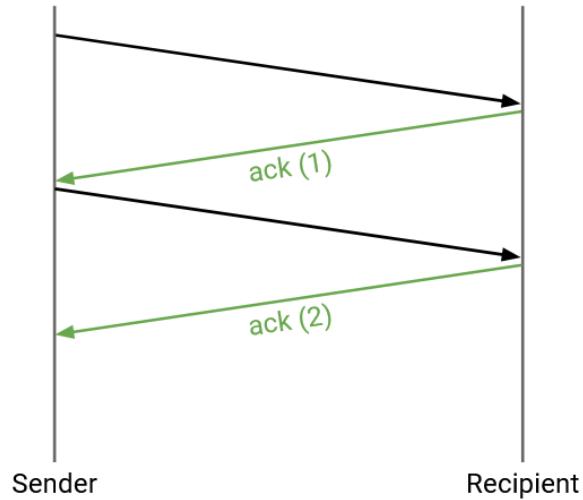
If you are the recipient: If you receive the uncorrupted packet, send an ack. (You might send multiple acks if you receive the packet multiple times.)

The core ideas in this example will apply to later protocols as well: checksums (for corruption), acknowledgements, re-sending packets, and timeouts.

Note that this protocol guarantees at-least-once delivery, since duplicates may exist.

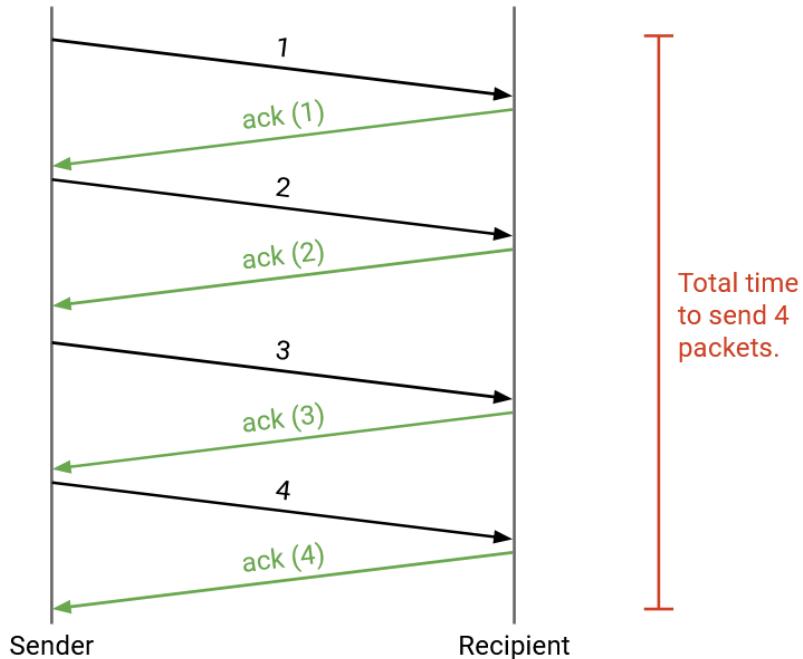
Reliably Delivering Multiple Packets

How would this protocol be extended to multiple packets?

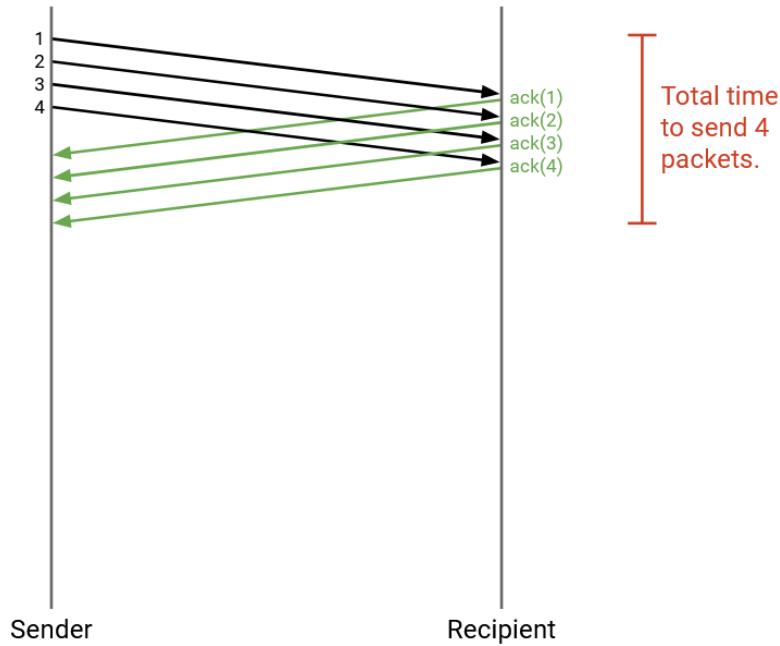


We could follow the same transmission rules (re-send when timer expires) for every single packet. To distinguish packets, we can attach a unique **sequence number** to every packet. Each ack will be related to a specific packet. Sequence numbers can also help us reorder packets if they arrive out of order.

When does the sender send each packet? The simplest approach is the **stop and wait** protocol, where the sender waits for packet i to be acknowledged before sending packet $i+1$. This will correctly provide reliability, but it is very slow. Each packet takes at least one RTT to be sent (more if a packet is dropped or corrupted).



This protocol might work in smaller settings where efficiency is less of a concern, but this is too slow for the Internet. How can we make this faster?



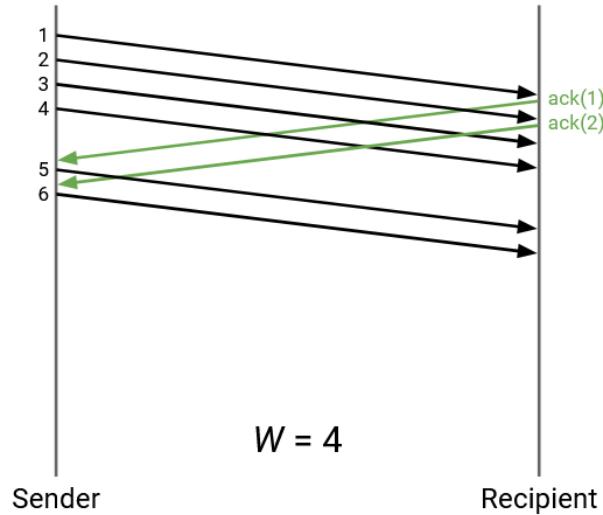
We can send packets in parallel. More specifically, we can send more packets while waiting for acks to arrive. When a packet is sent, but its corresponding ack has not been received, we call that packet **in flight**.

The simplest approach would be to send all packets immediately, but this could overwhelm the network (e.g. the link to your computer might have a limited bandwidth).

Window-Based Algorithms

Sending packets one at a time is too slow, but sending all packets at once overwhelms the network. To account for this, we will set a limit W and say that only W packets can be in flight at any given time. This is the key idea behind **window-based protocols**, where W is the size of the window.

If W is the maximum number of in-flight packets, then the sender can start by sending W packets. When an ack arrives, we send the next packet in line.



How should W be selected?

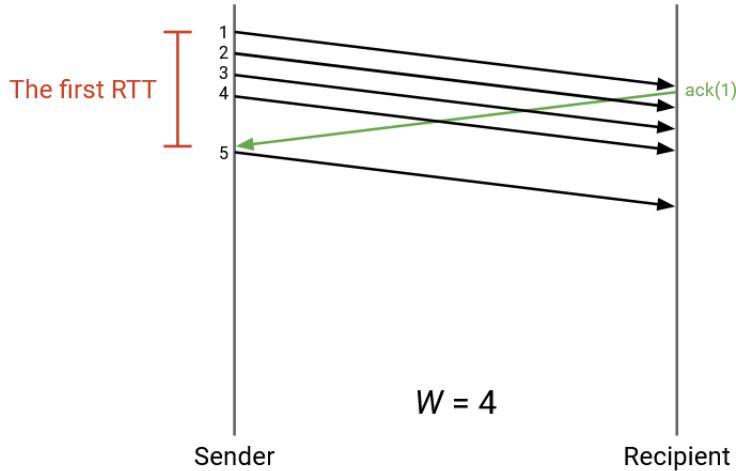
We want to fully use our available network capacity (“fill the pipe”). If W is too low, we are not using all of the bandwidth available to us.

However, we don’t want to overload links, since other people may be using that link (congestion control). We also don’t want to overload the receiver, who needs to receive and process all the packets from the sender (flow control).

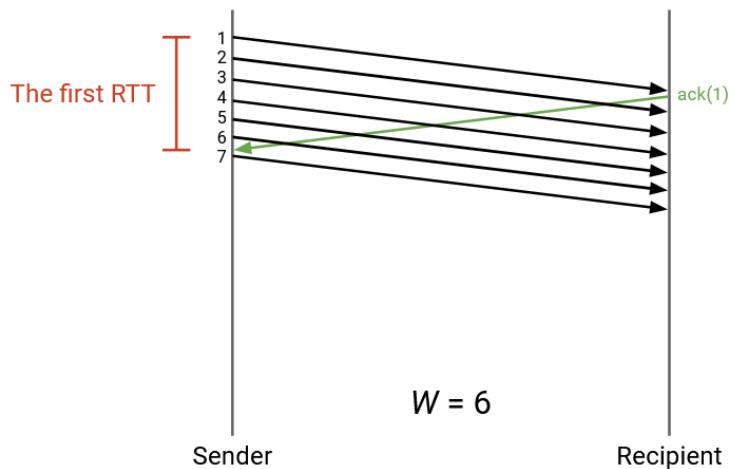
Window Size: Filling the Pipe

Let’s focus on just the first RTT, from the time the first packet is sent, to the time the first ack arrives. Suppose this time is 5 seconds (not a realistic number, just for example). Also, suppose that the outgoing link allows the sender to send 10 packets per second (also not a realistic number). In total, during this first RTT time, the sender should be able to send 50 packets in total. Therefore, 50 would be a reasonable window size, so that the sender is always sending packets and never sitting idle.

If we set W to be lower than 50, then the sender would finish sending all the initial packets before the first ack arrives. Then, the sender would be forced to sit idling while waiting for acks to arrive, and some network bandwidth would be wasted. More generally, we want the sender to be sending packets during the entire RTT.



In this example, W is 4. But, after sending 4 packets, the sender is idling and wasting bandwidth while waiting for the first ack to arrive.



In this example, W is increased so that the sender is constantly sending packets. As the first ack arrives, the sender is just about to reach the limit of W packets in flight, and is able to immediately continue sending packets as more acks arrive.

The route to the destination might have multiple links, with different capacities. Let B be the minimum (bottleneck) link bandwidth along the path. We shouldn't send packets faster than B , to avoid overloading the link. We also don't want to send packets any slower than B (i.e. we want to be using the rate B at all times).

Also, suppose that R is the round-trip time between sender and recipient. We can multiply R times B to get the total number of packets that can be sent during the RTT. (We can send B packets per second, for R seconds.) This tells us the window size, in packets.

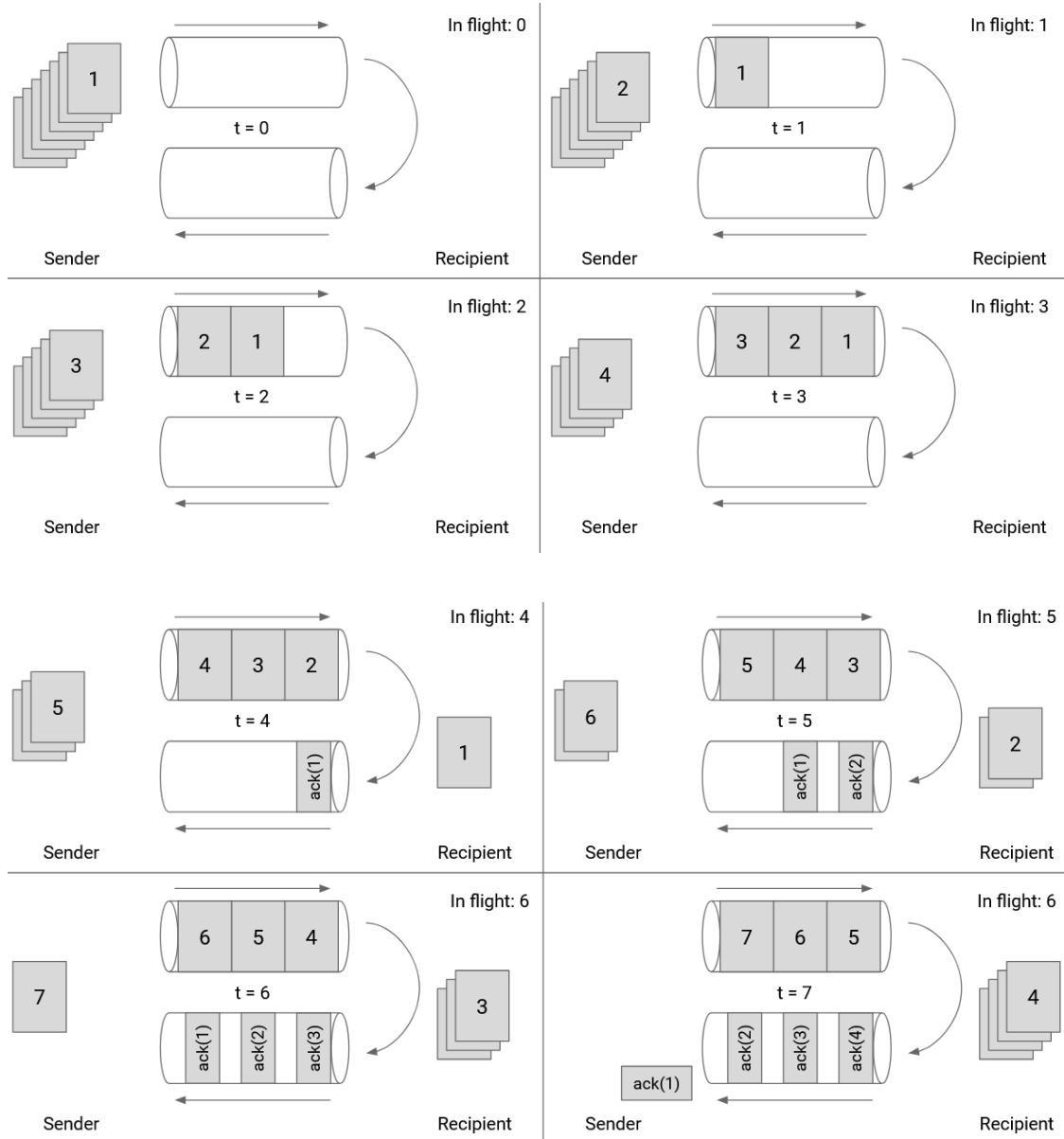
In reality, B is measured in bits per second, not in packets per second. When we multiply R times B , we get the number of bits that can be sent during the RTT. (B bits per second, for R seconds.) This tells us the window size, in bytes. In total, we can write:

$$W \text{ times packet size} = R \text{ times B}$$

The left-hand-side tells us the number of bytes sent during the window (W packets, times number of bytes per packet), and the right-hand-side tells us the number of bytes that can be sent during the RTT.

For a concrete example, we can set RTT = 1 second, and $B = 8$ Mbits/second. Then, R times B is 8 Mbits, or 1 megabyte, or 1,000,000 bytes.

If our packet size is 100 bytes, then we want $W = 10,000$ packets, so that we are fully using the bandwidth and sending 1,000,000 bytes during the RTT.

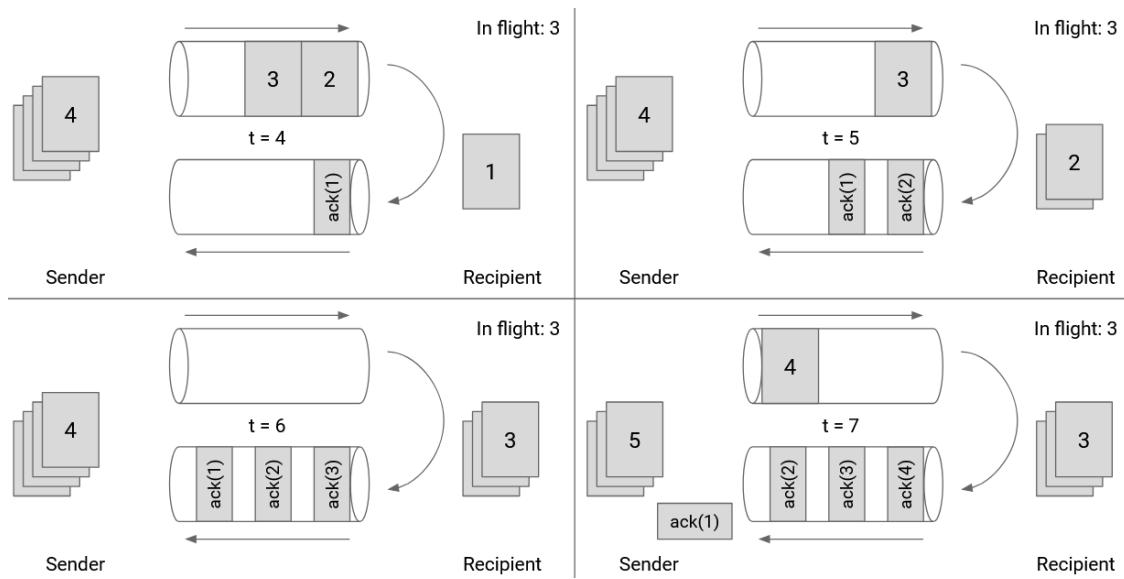


We can also draw the window size in terms of the link itself. In this picture, we are showing the outgoing and incoming directions of a specific link. As the sender pushes packets through the link at maximum

capacity, the first ack will arrive immediately after the 6th packet is sent. Therefore, our window size should be 6.

Note that the window size is not 3. When packet 6 is sent, 3 packets are being sent, but there are 3 more packets whose acks have not arrived, so there are a total of 6 packets in flight.

If we set the window size to 3, the outgoing pipe would have been unused while the acks for 1, 2, 3 are in flight.

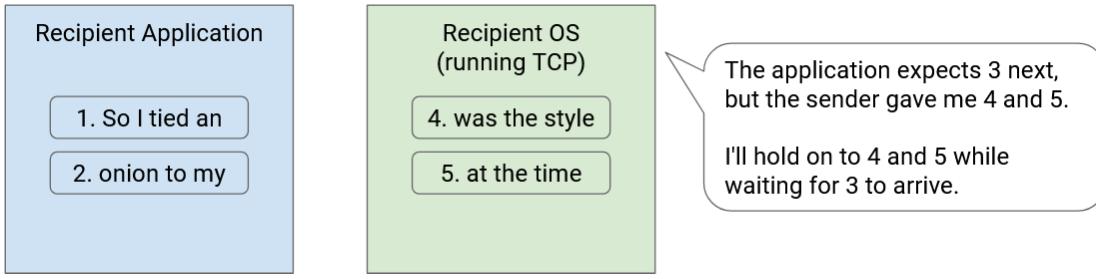


Note that the acks don't fill up the entire incoming pipe because the packets don't contain any actual data besides acknowledging receipt of a packet.

Window Size: Flow Control

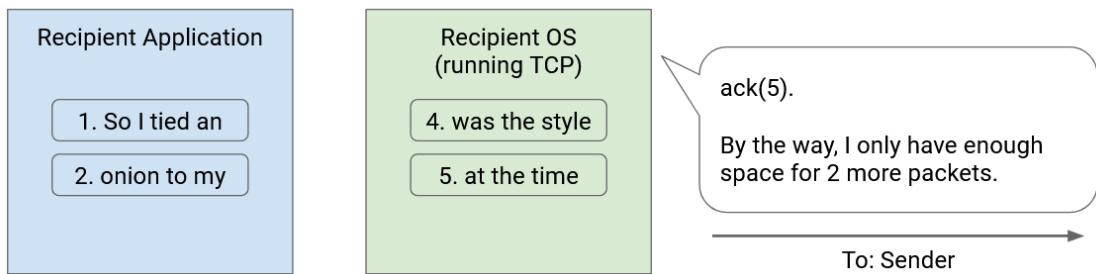
Consider the transport layer protocol in the recipient's operating system. The recipient might receive packets out-of-order, but the bytestream abstraction requires that packets are delivered in-order. This means that the transport layer implementation must hold on to the out-of-order packets by **buffering** them (keeping them in memory) until it's their turn to be delivered.

For example, suppose the recipient has received and processed packets 1 and 2. Then, the recipient sees packets 4 and 5. The transport layer implementation cannot deliver 4 and 5 to the application yet. Instead, we have to wait for packet 3 to arrive, and in the meantime, we have to keep packets 4 and 5 stored in the transport layer implementation's memory.



However, memory is not unlimited, and the recipient's buffer size for storing out-of-order packets is finite. The recipient has to store every out-of-order packet in memory until the missing packets in between arrive. If the connection has a lot of packet loss and reordering, the recipient might run out of memory.

Flow control ensures that the recipient's buffer does not run out of memory. To achieve this, we have the recipient tell the sender how much space is left in the buffer. The amount of space left in the recipient buffer is called the **advertised window**. In the acknowledgment, the recipient says "I have received these packets, and I have X bytes of space left to hold packets."

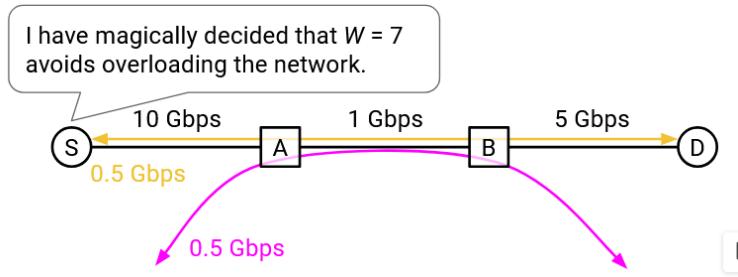


When the sender learns about the advertised window, the sender adjusts its window accordingly. Specifically, the number of packets in flight cannot exceed the recipient's advertised window. If the recipient says "my buffer has enough space for 5 packets," the sender must set the window to be at most 5 packets (even if the bandwidth might allow for more packets to be in flight).

Window Size: Congestion Control

Recall that in order to make the most use of bandwidth, the sender sets the window size to fully consume the bottleneck link bandwidth. For example, if the bottleneck link has bandwidth of 1Gbps, we will set the window size such that the sender is constantly sending data at 1Gbps for the entirety of the RTT (no idling).

In practice, it's unlikely that the 1Gbps link is only being used by a single connection. Other connections could also be using the capacity along that link. Instead of consuming the entire bandwidth on that link, the sender should only consume its own share of that bandwidth capacity.



But, what share of the bandwidth goes to each connection?

Suppose we had two connections using 400Mbps and 250Mbps, respectively. If another connection then tries to use that same link, maybe the sender's share is the remaining 350Mbps. But another argument is that the bandwidth is not being shared fairly, so perhaps everybody should adjust to use 333Mbps.

Determining and computing the exact amount of bandwidth that each connection gets to use is the goal of congestion control. Algorithms for congestion control are its own entire topic (covered in the next section). For now, we'll abstract away congestion control and say that as part of the transport layer, the sender is implementing a congestion control algorithm, whose job is to dynamically compute the sender's share of the bottleneck link on the connection.

The result of running the algorithm is the sender's congestion window (cwnd). For now, all you need to know is that the algorithm outputs this number, which represents a bandwidth that maximizes performance, without overloading a link, while fairly sharing bandwidth with other connections.

We now know how to set the window to achieve our three goals from earlier. To fully utilize network capacity, we will set the window size according to the RTT and the bottleneck link bandwidth.

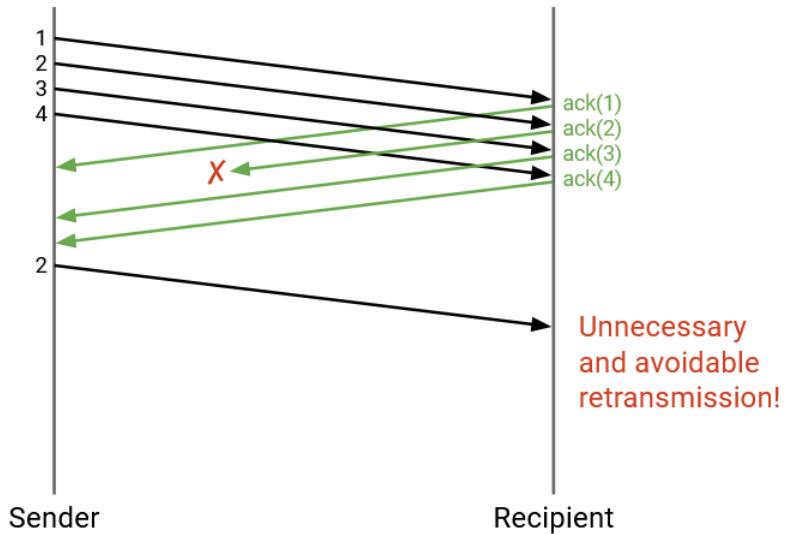
To avoid overloading the receiver, we will limit the window size according to the recipient's advertised window. To avoid overloading links, we will limit the window size according to the sender's congestion window (some number outputted by the sender running a congestion control algorithm).

In order to meet all three goals, we'll set the window size to the minimum of all three values. In practice, note that the congestion window (third goal) is always less than or equal to the window size from fully using bandwidth (first goal). If there's no congestion, we'd be fully using all of the bottleneck bandwidth, so the two numbers would be equal. In most cases, congestion will force us to use less than all of the bottleneck bandwidth, so the third number would be less than the first number. There is no case where the congestion window bandwidth would be greater than the bottleneck bandwidth.

Also, in practice, it's difficult to discover the bottleneck bandwidth. The sender would have to somehow traverse the network topology and learn about each link's bandwidth. Because the first number is hard to learn, and is always greater than or equal to the third number, we can set our window size to the minimum of the latter two numbers (ignoring the first number). The window size is the minimum of the sender's congestion window, and the receiver's advertised window.

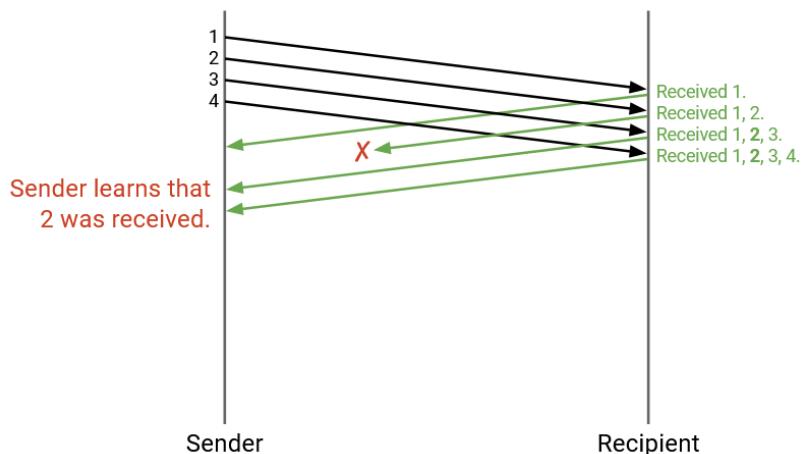
Smarter Acknowledgments

So far, every ack packet corresponds to a single packet. Can we do better than acknowledging one packet at a time? What are some issues with acknowledging one packet at a time?



In this example, one of the acks is dropped, even though the recipient successfully received all 4 packets. This would force the sender to re-send packet 2, even though this re-sending was unnecessary.

Instead of sending an acknowledgement for a specific packet, each time we send an acknowledgement, we can actually list every packet we have received. This is called a **full information ack**.

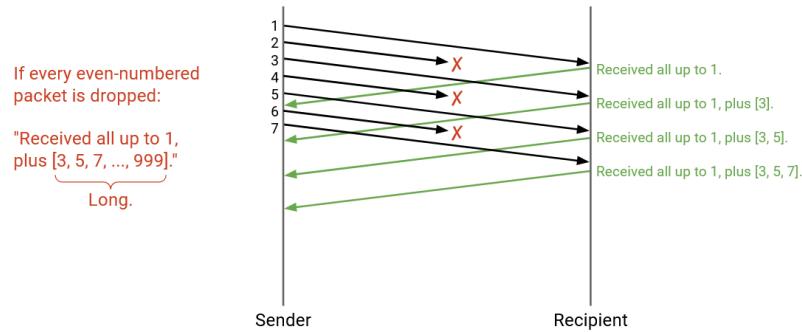


In this example, the acks now say: “I received 1,” and “I received 1 and 2”, and “I received 1, 2, 3”, and “I received 1, 2, 3, 4.”

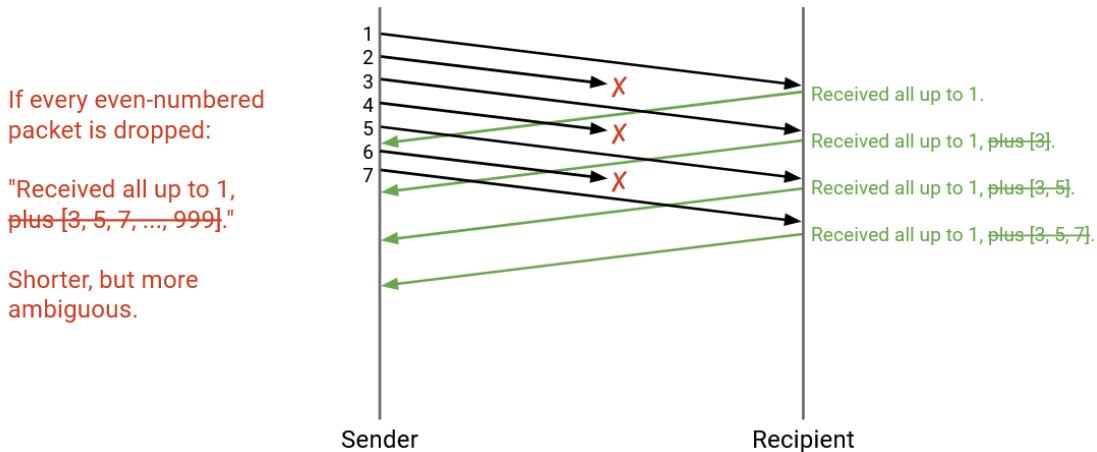
Even though the second ack was dropped, the third and fourth acks help the sender confirm that packet 2 was received, and packet 2 no longer needs to be re-sent.

As more packets are sent, the list of all packets received is going to get very long. Full information acks can abbreviate this information by saying: “I have received all packets up to #12. Also, I received #14 and #15.” Formally, we give the highest cumulative ack (all packets less than or equal to this number have been received), plus a list of any additional packets received.

Even with this abbreviation, full information acks can get long. For example, if all even-numbered packets are dropped, then the highest cumulative ack will always be 1 (we can only say all packets up to 1 have been received, since 2 is dropped). The rest of the received packets will have to be in a list like $[1, 3, 5, 7, 9, \dots]$ which can get very long.



A compromise between individual acks (every ack drop forces re-sending) and full information acks (acks can get long) is **cumulative acks**, where we provide only the highest cumulative ack, and discard the additional list. Formally, the ack encodes the highest sequence number for which all previous packets have been received.



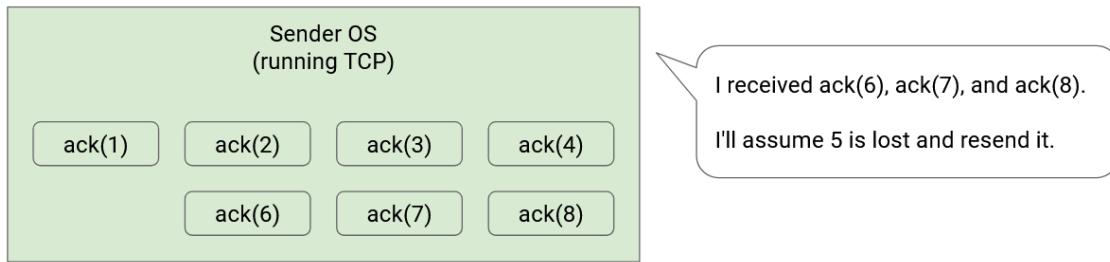
In this example, where even-numbered packets are dropped, every cumulative ack would say: “I received all packets up to and including 1.” Even though 3 and 5 were received, the cumulative ack will not encode this information, because it only confirms receipts of consecutive packets starting from 1.

Cumulative acks no longer have scaling issues (we’re always sending one number, not a list of numbers). However, they can be more ambiguous, as in the case above. The sender sees three acknowledgements all saying “I received everything up to and including 1,” and can deduce that 3 packets were received (packet 1, and two other packets), but cannot deduce what those other two packets are.

Detecting Loss Early

Can we do better than waiting for timeouts, and use other information that we receive to detect loss earlier and re-send packets sooner? For example, in our individual ack model, if we receive acks for packets 1, 3, 4, 5, 6, we might deduce that packet 2 is lost and re-send it, even before packet 2's timer expires.

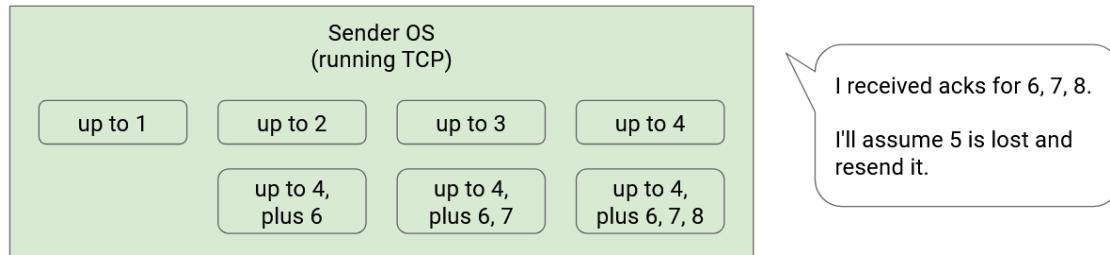
More formally, we can set a value K (not related to the window), and say that if K subsequent packets are acked after the missing packet, we'll consider the packet lost (even if the timer hasn't expired). For example, if K=3, we're waiting on packet 5's ack, and we get acks for 6, 7, and 8, then we can consider packet 5 lost.



In practice, detecting loss from subsequent acks is much faster than waiting for a timeout. If our timeout is calculated from the RTT, it could be on the order of seconds. On the other hand, modern bandwidths can allow for acks to arrive once every few microseconds.

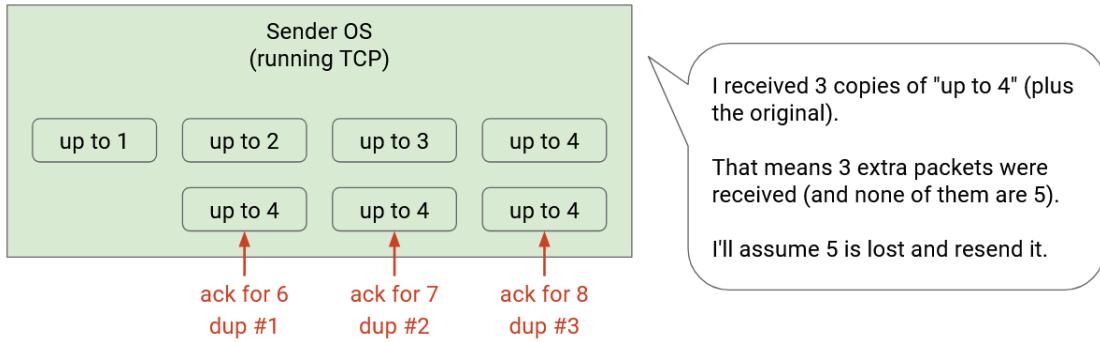
This strategy for detecting loss looks different depending on our strategy for sending acks. The above examples assume that we're sending individual acks, but what about the other two ack models?

If we use full-information acks, the strategy is pretty similar, and the acks will actually show the missing packet more clearly.



If packet 5 is lost, the acks might say "up to 4", then "up to 4, plus 6", then "up to 4, plus 6, 7", then "up to 4, plus 6, 7, 8." At this point, if K=3, then K packets after 5 have been acked, so we can declare that packet 5 is lost.

If we use cumulative acks, this strategy can be more ambiguous. If packet 5 is lost, then the acks might say "up to 4" (acking 4), "up to 4" (acking 6), "up to 4" (acking 7), "up to 4" (acking 8). The sender is seeing **duplicate acks** because of the gap in consecutive packets. If K=3, then we can declare packet 5 lost after receiving 3 duplicate packets (corresponding to 3 more packets acked after the gap), for a total of 4 duplicates.



When we had individual and full-information acks, we could clearly see which packet needed to be re-sent. There was one packet missing the ack (and K subsequent acks arriving). However, the decision for which packet to re-send is more ambiguous with cumulative acks, especially when multiple packets are lost.

As an example, consider a sender with window size $W=6$, and $K=3$. So far, packets 1 and 2 have been acked, and packets 3-8 are in flight. Suppose packets 3 and 5 have been dropped. Let's first walk through this example with individual ACKs.

4 arrives, and the recipient sends an ack for 4. The sender can now send 9.

6 arrives, and the recipient sends an ack for 6. The sender can now send 10.

7 arrives, and the recipient sends an ack for 7. The sender can now send 11.

At this point, the sender notices that $K=3$ packets after packet 3 (namely 4, 6, and 7) have been acked. The sender can declare 3 lost, and re-send 3 as well.

Note that even though the sender re-sent 3 and sent 11 as a response to the ack for 7, there are still a total of 6 packets in flight with this re-sending, so the window is not violated. This is because 3 was already one of the in-flight packets when we re-sent it.

8 arrives, and the recipient sends an ack for 8. The sender can now send 12.

Also, the sender notices that $K=3$ packets after packet 5 (namely 6, 7, and 8) have been acked, so the sender can re-send 5 as well.

9 arrives, and the recipient sends an ack for 9. The sender can now send 13.

Now, let's redo this example with cumulative ACKs.

4 arrives, and the recipient sends an ack for 4, which says "ack everything up to 2." At this point, the sender knows a packet must have arrived, but does not know that it's 4. Still, the sender can send 9 next. Note that the window is not violated, because even though the sender seemingly has 7 un-acked packets, one of them did get acked by the duplicate "ack everything up to 2," so there are only 6 packets in flight.

6 arrives, and the recipient sends an ack for 6, which still says "ack everything up to 2." Again, the sender deduces that another packet arrived, and can send 10 next.

7 arrives, and the recipient sends an ack for 7, which still says "ack everything up to 2." The sender deduces that another packet arrived, and can send 10.

At this point, the sender notices that "ack everything up to 2" has arrived 3 duplicate times (in addition to

the initial ack for 2). The next un-acked packet is 3, so the sender will re-send 3.

This is when things get ambiguous. When 8, 9, and 10 arrive at the recipient, the sender will receive three more copies of “ack everything up to 2.” (We’re assuming the recipient hasn’t received 3 yet, since it was re-sent after 9 and 10).

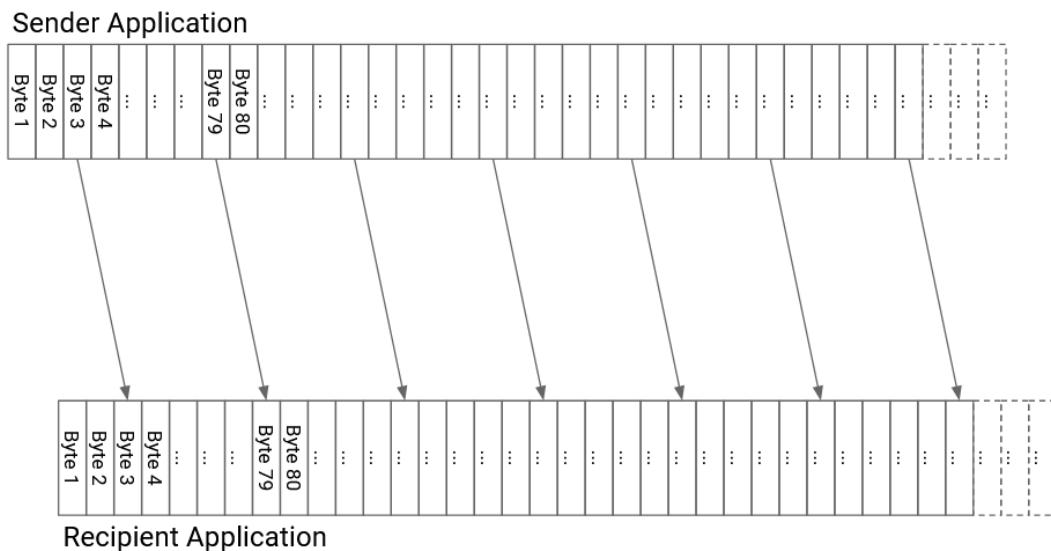
The sender can now send 12, 13, and 14, since three more acks have arrived, but which packet should be re-sent next? Should the sender re-send 3, 4, 5, or something else?

This example shows that cumulative acks don’t always indicate exactly which packets were received. However, the number of acks (possibly including duplicates) can be used to determine how many packets were received (without knowing exactly which packets were received), which allows us to keep sending according to the window size. However, ambiguity arises when we receive too many duplicate acks and cannot tell which packet to re-send.

TCP Implementation

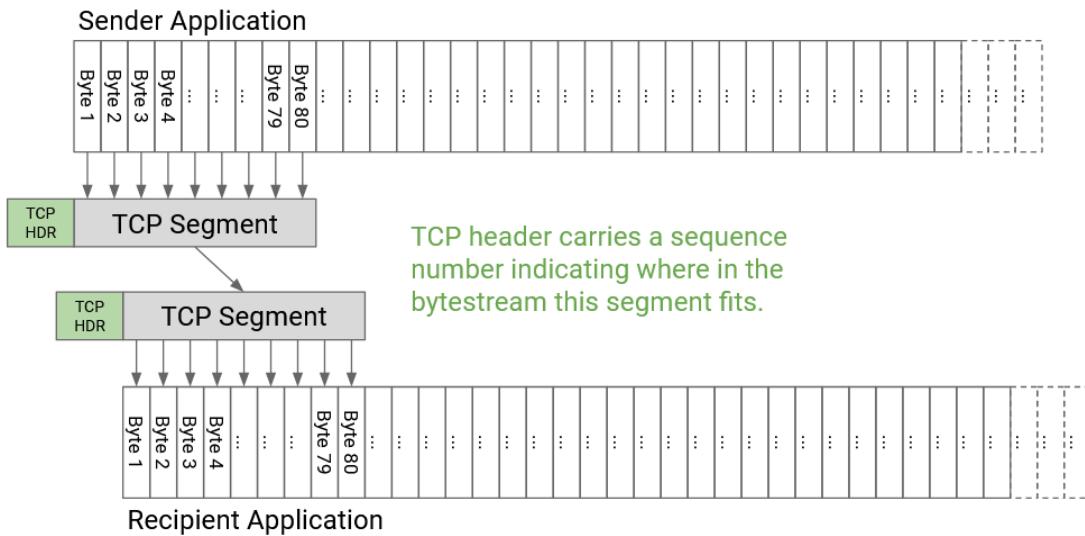
TCP Segments

So far, we've been talking about TCP conceptually, in terms of individual packets being sent. But the application doesn't provide us with pre-made packets that we can directly send into the Layer 3 network. The application is relying on a bytestream abstraction, and is instead sending us a continuous stream of bytes. In order to fully implement TCP, we'll need to rethink all of our previous ideas (e.g. sequence numbers, window size, etc.) in terms of bytes, not packets. (You should still be able to reason about the design choices in terms of both bytes or packets, though.)



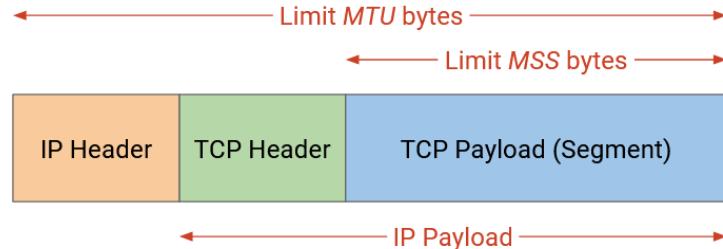
In order to form packets out of bytes in the bytestream, we'll introduce a unit of data called a **TCP segment**. The TCP implementation at the sender will collect bytes from the bytestream, one by one, and place those bytes into a TCP segment. When the TCP segment is full (reaches a fixed maximum segment size), we send that TCP segment, and then start a new TCP segment.

Sometimes, the sender wants to send less data than the maximum segment size. In that case, we wouldn't want the TCP segment to be waiting forever for more bytes that never come. To fix this, we'll start a timer every time we start filling a new empty segment. If the timer expires, we'll send the TCP segment, even if it is not full yet.



Before sending the data in a TCP segment, the sender's TCP implementation will add a TCP header with relevant metadata (e.g. sequence number, port numbers). Then, the segment and header are passed down to the IP layer, which will attach an IP header and send the packet through the network.

The TCP segment, with a TCP header and IP header on top, is sometimes called a **TCP/IP packet**. Equivalently, this is an IP packet whose payload consists of a TCP header and data.



How should the **maximum segment size (MSS)** be set? Recall that the size of an IP packet is limited by the maximum transmission unit (MTU) along each link. However, the IP packet must also contain the IP and TCP header, so the TCP maximum segment size is going to be slightly smaller than the IP maximum transmission unit. Specifically:

$$\text{MSS (TCP segment limit)} = \text{MTU (IP packet limit)} - \text{IP header size} - \text{TCP header size}$$

Sequence Numbers

So far, we've been labeling each packet with a number, so that the recipient can receive packets in the correct order.

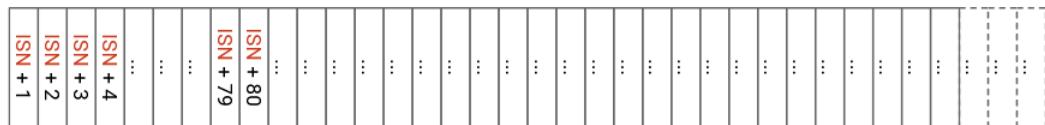
In practice, instead of numbering individual segments, we assign a number to every byte in the bytestream. Each segment's header will contain a **sequence number** corresponding to the number of the first byte in

that segment. The recipient can still use sequence numbers to figure out where each segment fits in the bytestream, and reassemble the segments in the correct order.

Each bytestream starts with an **initial sequence number (ISN)**. The sender chooses an ISN and labels the first byte with number ISN+1, the next byte with number ISN+2, the next byte with ISN+3, and so on.



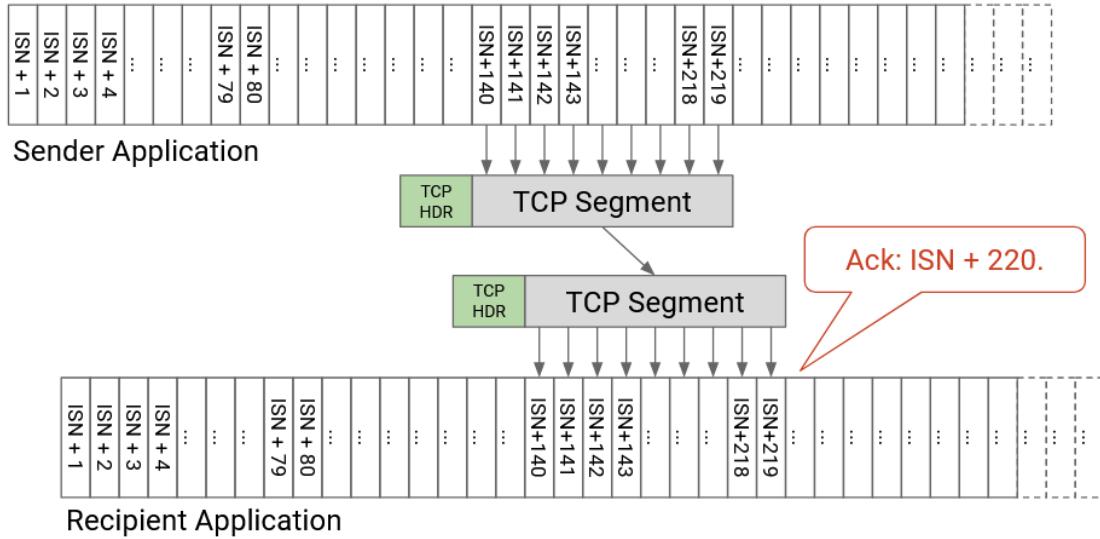
Sender Application



Recipient Application

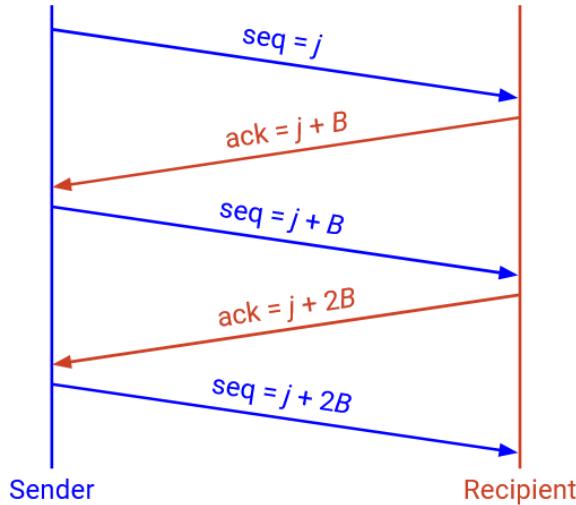
Since we're now numbering bytes instead of packets, acknowledgement numbers will also now be in terms of bytes, not packets. Specifically, the acknowledgement number says, I have received all bytes up to, but not including, this number. Equivalently, the acknowledgement number represents the next byte it expects to receive (but has not received yet). Note that TCP is using the cumulative ack model (as opposed to full-information acks or individual byte acks).

As an example, suppose the ISN has randomly been chosen to be 50. Then the first few bytes have numbers 51, 52, 53, etc. A specific TCP segment might contain the bytes 140 to 219, inclusive. The sequence number of this segment is 140 (representing the first byte in the segment). If the recipient has received everything so far, the recipient can acknowledge this segment by sending an ack number of 220, which is the next byte that has not been received yet.



More generally, suppose we have a packet where the first byte has sequence number X, and the packet has B bytes. This packet has the bytes X, X+1, X+2, ..., X+B-1. If this packet (and all prior data) is received, the ack will acknowledge X+B (the next expected byte). If this packet is not received, or this packet is received but some prior packet was not received, then the ack will acknowledge some smaller number (because TCP uses cumulative acks).

More generally, suppose we had many packets, all B bytes long. The ISN is X, and the window size is 1 (stop-and-wait protocol, only one packet or ack being sent at once). Assume that no packets are dropped. Then, the sequence and ack numbers would proceed as follows: The first packet has sequence number X. The first ack has ack number X+B. The second packet has sequence number X+B. The second ack has ack number X+2B. The third packet has sequence number X+2B, and so on. In particular, note that when there's no loss, the ack number corresponds to the next packet's sequence number.



Historically, the ISN was chosen to be random because the designers were concerned about ambiguous sequence numbers if all bytestreams started numbering at 0. Specifically, suppose a TCP connection sends

some data starting at ISN 0, and then the sender crashes. If the sender restarts a new connection, and the ISN starts at 0 again, the recipient might get confused if it sees a packet with sequence number 0. Is this packet from the first connection before the crash, or the second connection after the crash?

In practice, the ISN is chosen to be random for security reasons. If the ISN is chosen in a predictable way, attackers can deduce the ISN and send spoofed packets that look like they're coming from the sender. When the ISN is chosen randomly, it's harder for the attacker to deduce the ISN and send spoofed packets.

TCP State

In TCP, both the sender and recipient need to maintain state. The state is maintained at the end hosts implementing TCP, not in the network.

The sender has to remember which bytes have been sent but not acknowledged yet. The sender also has to keep track of various timers, e.g. a timer for when to send a less-than-full segment, and a timer for when to resend bytes.

The recipient has to remember the out-of-order bytes that can't be delivered to the application yet.

Because TCP requires storing state, each bytestream is called a **connection** or **session**, and TCP is a connection-oriented protocol. Unlike Layer 3, where every packet could be considered separately, TCP requires both parties to establish a connection and initialize state before data can be sent. TCP also needs a mechanism to tear down connections to free up the memory allocated for state on both end hosts.

TCP is Full Duplex

So far, we've seen TCP as a bytestream from one end host (the sender) to the other end host (recipient). In practice, the two end hosts often want to send messages in both directions.

To support sending messages in both directions, TCP connections are **full duplex**. Instead of designating one sender and one recipient, both end hosts in the connection can send and receive data simultaneously, in the same connection.

15	16	17	18	19	20	21
H	e	I	I	o	,	B

A to B bytestream

77	78	79	80	81	82	83
H	e	I	I	o	,	A

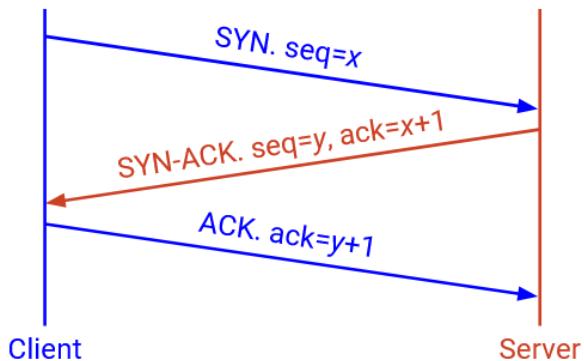
B to A bytestream

To support sending data in both directions, each TCP connection has two bytestreams: one containing data from A to B, and the other containing data from B to A. Each packet can contain both data and acknowledgement information. The sequence number would correspond to the sender's bytestream (the bytes I am sending), and the acknowledgement number would correspond to the recipient's bytestream (the bytes I received from you).

TCP Handshake

Recall that TCP is connection-oriented, so connections must be explicitly created and destroyed. Also, recall that bytestreams start at a randomly-selected initial sequence number (ISN), and that each TCP connection is full-duplex (two bytestreams, one in each direction). When we create a new connection, we need both sides to agree on two starting ISNs (one per direction).

To establish a TCP connection, the two hosts perform a **three-way handshake** to agree on the ISNs in each direction.



The first packet (from A to B) is the **SYN** message. This message contains A's ISN (data from A to B will start counting at this ISN), in the sequence number.

The second packet (from B to A) is the **SYN-ACK** message. This message contains B's ISN (data from B to A will start counting at this ISN), in the sequence number. This message also acknowledges that B has received of A's ISN, in the ack number.

The third packet (from A to B again) is the **ACK** message. This message acknowledges that A has received B's ISN, in the ack number.

This handshake is why bytestreams start counting at ISN+1. When I send an ISN, the ack is ISN+1, indicating that the ISN was received, and the next (first) byte expected is ISN+1.

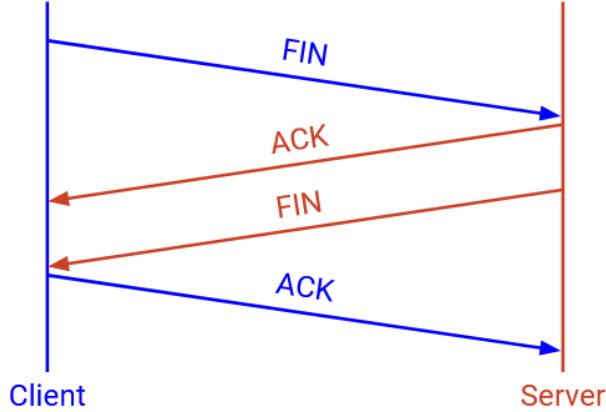
After the three-way handshake concludes, B can start sending data.

Ending Connections

There are two ways to end a connection.

In normal cases, when I am done sending messages, I can send a special FIN packet, which says: I will not send any more data, but I will continue to receive data if you have any more to send. At this point, the connection is half-closed. This packet will be acked, just like any other packet.

Eventually, the other side will also finish sending data and send a FIN packet. When this FIN packet is acked, the connection is closed.

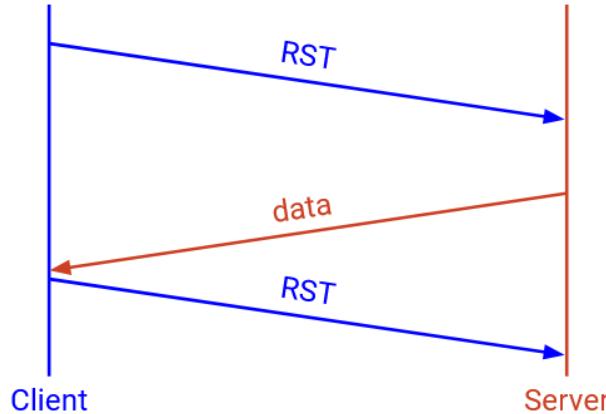


Sometimes, we have to terminate a connection abruptly, without the agreement of the other side. To unilaterally end a connection, I can send a special RST packet, which says: I will not send or receive any more data. This packet does not have to be acked, and I can tear down my connection as soon as I send this data.

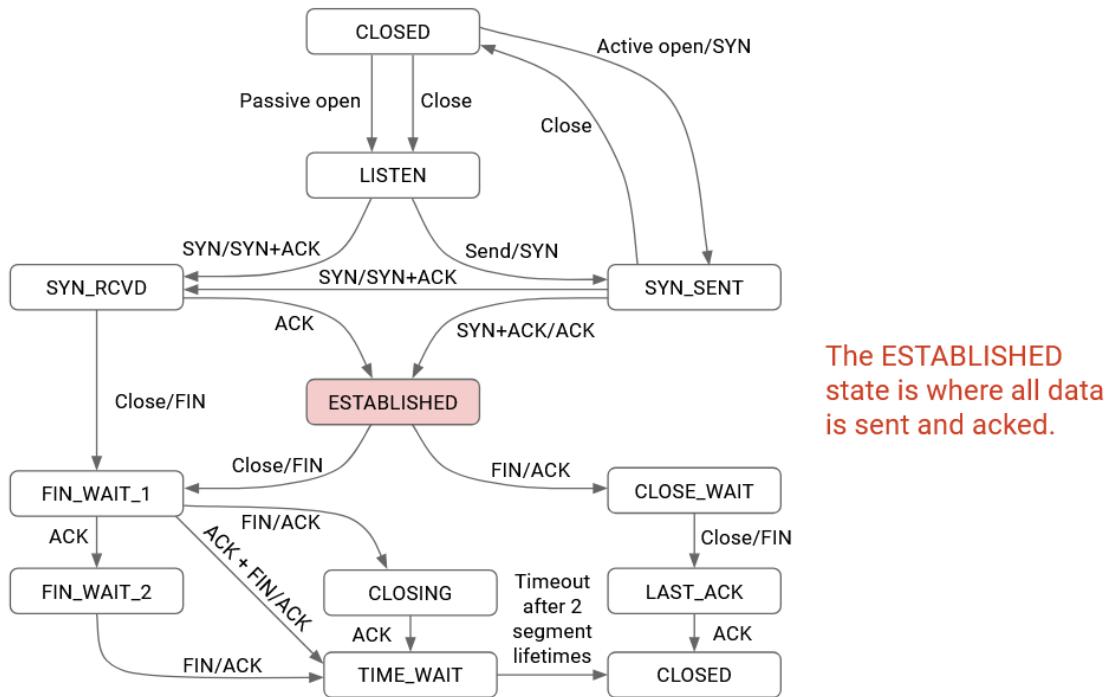
RST packets are often used when a host encounters an error and is unable to continue sending or receiving packets. Note that any in-flight data is lost if a RST occurs and the end host crashes and loses its state.

If I sent a RST, and someone continues sending me data, if I am able, I will continue to send copies of the RST packet to repeatedly try and terminate the connection.

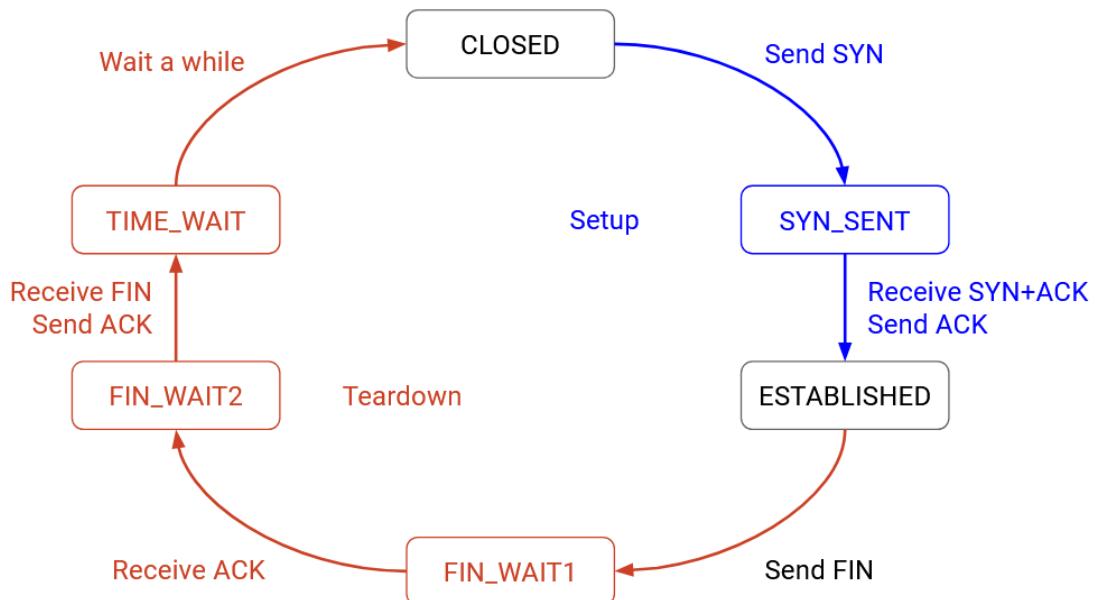
RST packets can also be used by attackers to censor connections. An attacker can spoof and inject a RST packet, which causes the entire connection to terminate.



The full TCP state diagram is quite complicated, with many intermediate states in the process of opening or closing a connection. Examples of intermediate states include: I have sent a SYN, and am waiting for a SYN-ACK. Or, I have received a FIN, sent my FIN, but am waiting for my FIN to be acked. Most TCP connections spend most of their time in the Established state, where the connection has started (but not ended), and data is being exchanged back-and-forth. You don't need to understand this full state diagram for these notes.



In the simplified state diagram, we start in the closed state (no connection in progress). To start a connection, we send a SYN. Eventually, we receive a SYN-ACK and reply with an ACK, moving to an established connection. When we're done sending data, we send a FIN, and receive an ACK. Eventually, we receive a FIN, and the connection is closed again.



Piggybacking

Because TCP is full duplex, it's possible for a packet to both acknowledge some data and send new data.

When the recipient gets a packet, if it has no data to send, the recipient has two choices. The recipient could either immediately send the ack, with no data to send. Or, the recipient could wait until it has some data to send, and then send the ack with the new data. This latter approach is called **piggybacking**.

In practice, one reason we might not piggyback is because TCP is implemented in the operating system, separate from the application.

Consider the operating system, which has no idea what the application code is doing. When the operating system receives a packet, it doesn't know when the sender will have more data to send (or if the sender will ever have more data to send), so it might be stuck waiting a long time before it's able to piggyback the ack with some new data.

On the other side, consider the application, which has no idea what the operating system is doing. The application is running on the bytestream abstraction, and isn't thinking about packets at all, so it has no way to think about piggybacking at all.

Piggybacking is further complicated by the fact that the operating system isn't running every program simultaneously. Thinking back to a computer architecture course (like CS 61C at UC Berkeley), the CPU is constantly switching between different processes on your computer, depending on what needs attention. It would be pretty silly if, every time a TCP packet arrived, the CPU interrupted what it was doing to pass that packet to the application, and gave the application some time to respond. Instead, when a TCP packet arrives, the operating system might send out the ack, before the application gets a chance to piggyback new data on the ack.

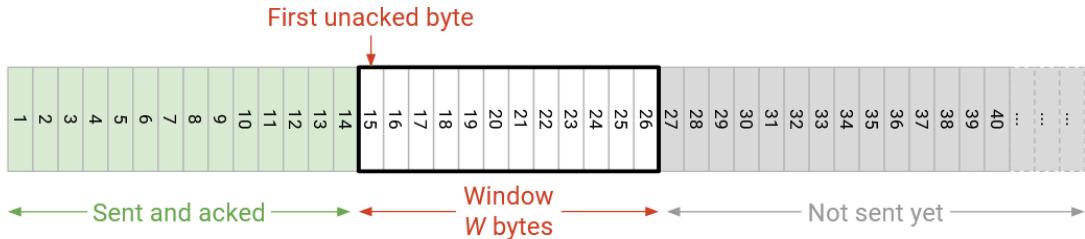
One case where data is always piggybacked is the SYN-ACK packet in the handshake. In addition to the ack, we're piggybacking our own initial sequence number. This doesn't have the problem discussed above, since the TCP handshake is entirely performed by the operating system. (The application isn't thinking about SYN or SYN-ACK packets at all.)

Sliding Window

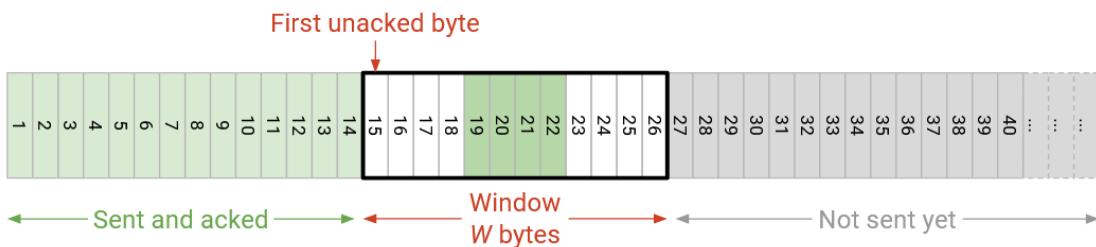
When we discussed packets, we defined the window as the number of packets that could be in flight at any given time. Now that we're implementing TCP in terms of bytes, we'll define the **sliding window** as the maximum number of contiguous bytes that can be in flight at any given time.

The restriction of the in-flight bytes being contiguous is different from before. Our packet-based window definition allowed for non-contiguous packets (e.g. 5, 7, 8) to be in-flight. However, the bytes in flight are required to be consecutive, with no gaps. This requirement creates a window (range of bytes) in the byte stream.

The left side of the window is the first unacknowledged byte (as determined by the ack number from the recipient). Starting at this byte, the next W bytes, up to the right side of the window, can be in-flight.



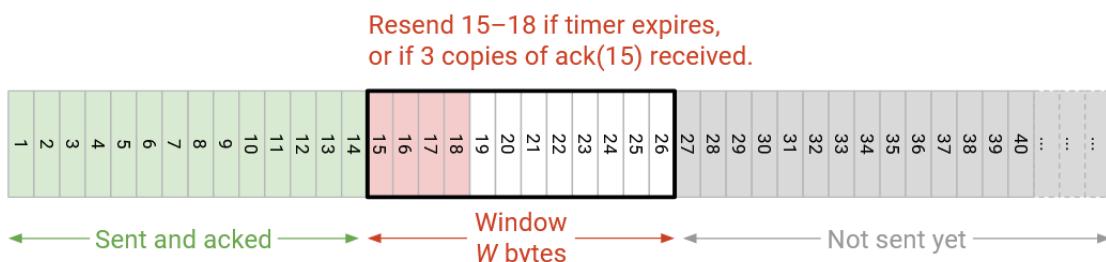
Note that even if some of the intermediate bytes in this window were acknowledged, we still cannot send more bytes beyond the window. The only way we can send more bytes is if the window slides to the right, i.e. when the ack number increases (bytes on the left side of the window are acknowledged).



Recall that the window size (which determines the right edge of the window) is limited by flow control and congestion control. In the case of flow control, the window size is decided by the window advertised by the recipient. The recipient decides the advertised window based on the amount of buffer space available on the receiver end.

Detecting Loss and Re-Sending Data

There are two conditions for data to be re-sent. Only one condition (not both) needs to be true to trigger a re-send.



The first trigger for retransmission is a timer (data not acknowledged after some time). In packet-based TCP, every packet had a timer, and when the timer expired without that packet being acked, we would re-send that packet.

In byte-based TCP, instead of one timer per byte or per packet, we will only have a single timer, corresponding to the first unacknowledged byte (left side of the window). If the timer expires, we will re-send the left-most

unacknowledged segment. Recall that the timer length is based on the RTT, and the RTT is estimated using measurements of the time between sending data and receiving an ack. Also, recall that the timer is reset every time a new ack arrives (and the window changes).

The second trigger for retransmission is assuming that data is lost when we receive acks for subsequent packets. In packet-based TCP with cumulative acks (which is what TCP uses), we would re-send a packet if we received K duplicate acks ($K=3$ is common), which indicated that three subsequent packets were acknowledged.

In byte-based TCP, if we receive K duplicate acks, we will re-send the left-most unacknowledged segment.

TCP Header

What functions does TCP implement?

1. Demultiplexing ([ports](#))
2. Reliability ([checksum, sequence and ack numbers](#))
3. Connection setup and teardown ([flags](#))
4. Flow control ([advertised window](#))

Source Port (16)		Destination Port (16)	
Sequence Number (32)			
Acknowledgment Number (32)			
Hdr Len (4)	0000	Flags (8)	Advertised Window (16)
		Checksum (8)	Urgent Pointer (8)
Options (variable-length)			
Payload			

The TCP header has 16-bit source and destination ports.

The TCP header has a 32-bit sequence number (byte offset of the first byte in this packet), and a 32-bit acknowledgement number (highest contiguous sequence number received, plus one).

The TCP header has a checksum over the entire data (not just the header), to detect corrupt data.

The TCP header has the advertised window, which is used to support flow control and congestion control.

The header length specifies the number of 4-byte words in the TCP header. Assuming there are no additional options, this length is 5.

The flags are a sequence of bits that can be set to 1 or 0. When a bit is set to 1, the corresponding flag is enabled. Everybody understands the semantics of the header, so they know which bits correspond to which flags. There are four relevant flags for these notes.

The SYN (synchronize) flag is turned on when the host is sending its ISN. This flag is usually only enabled in the first two messages of the three-way handshake.

The ACK (acknowledge) flag is turned on when the acknowledgment number is relevant and being used to

ack data. If I want to send data, but didn't receive any data that needs to be acked, I can turn this flag off, which tells the other host to ignore the ack number.

There are 6 reserved bits after the header length that are always set to 0. You can safely ignore these.

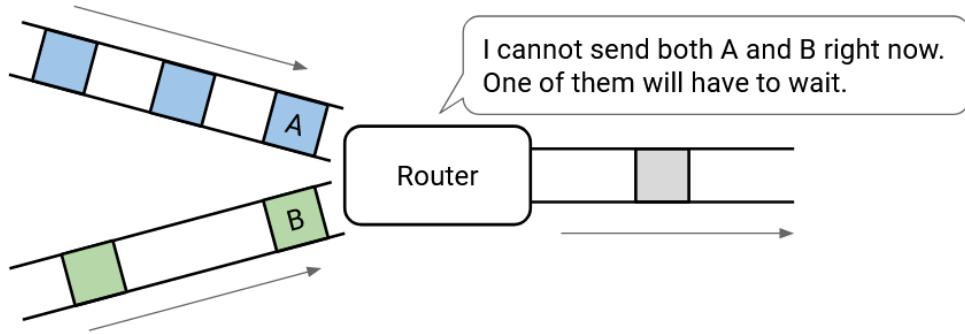
The urgent pointer can be used to mark certain bytes as urgent, which tells the recipient to send this data to the application as soon as possible. This is a historical field that we won't cover any further.

The TCP header can have additional options appended to the end (which would make the header longer), but we'll ignore options for this class. For example, if you wanted to implement full-information acks, there is an option called selective acknowledgements (SACK) that can be added to the header.

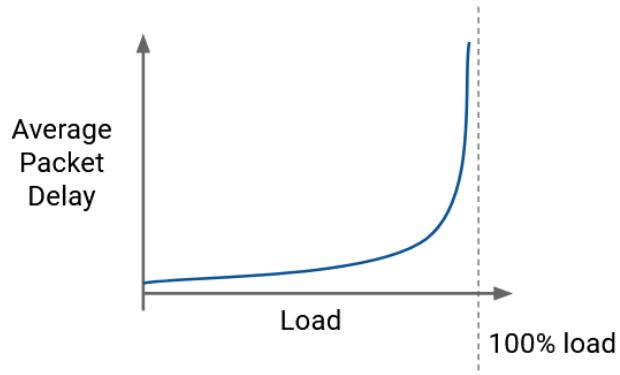
Congestion Control Principles

Congestion is Harmful

Recall that if many packets arrive at a router at the same time (e.g. bursty traffic), and the router needs to send both packets over the same link, then the router will send one packet and put the other packets in a queue (to be sent later).



More generally, if the input rate of packets exceeds the output rate that the link can sustain, the router will be unable to keep up with the pace of incoming packets. This router is **congested**, and needs to keep packets in a queue while they wait their turn to be sent. The queue can cause packets to be delayed. If the queue itself gets too full and packets are still incoming, then packets can get dropped.



This graph shows the performance of a queueing system with bursty arrivals. The dotted line represents the link's capacity (maximum load). As we increase the load, packets get more delayed.

When arrivals are bursty, we can't realistically use the maximum capacity of the link. We have to find an appropriate performance trade-off between load and packet delay.

Notice that the graph starts sloping upwards even before we reach the dotted line. This means that the queueing is already delaying packets, even if nothing is dropped. By the time we reach maximum utilization and start losing packets, we are already incurring very large packet delays from the queue.

Brief History of Congestion

In the 1980s, TCP did not implement any congestion control. The sending rate was only limited by flow control (recipient buffer capacity).

If packets were dropped, the sender would re-send copies of the packet repeatedly, at the same fast rate, until the packet arrived. A smarter approach would be to slow down to avoid packets being dropped and reduce the copies clogging up the network, but early TCP implementations did not do this.

In October 1986, the Internet started to suffer from a series of congestion collapses, where the capacity of the Internet significantly decreased. One link between UC Berkeley to Lawrence Berkeley Lab (two sites roughly 400 yards away) had its throughput drop from 32 Kbps = 32,000 bps to 40 bps.

Michael Karels (UC Berkeley undergraduate) and Van Jacobson (Lawrence Berkeley Lab researcher) were working on the networking stack in the Berkeley Unix system (influential early operating system), and they realized that the network had thousands of copies of the same packet, because everybody was trying to re-send packets that were being dropped.

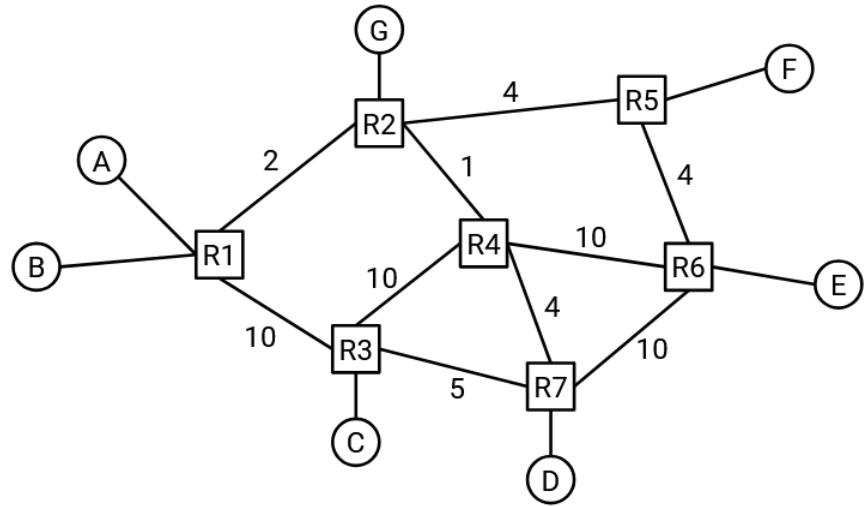
Karels and Jacobson developed an algorithm for fixing the problem, which evolved into the modern TCP congestion control algorithm. Their fix was a modification to TCP itself, where the window size (which dictates the rate of sending packets) is dynamically adjusted in response to packet loss.

Because their solution was a modification to the logic of TCP (recall, TCP is implemented in the operating system), no upgrades to routers or applications were needed.

TCP congestion control is one of many examples of Internet design being ad-hoc. Karels and Jacobson's patch was only several lines of extra code in the BSD operating system's implementation of TCP. The patch worked, so it was quickly adopted. Since then, the topic of congestion control has been extensively researched and several improvements have been made, but ultimately, the core ideas in the original patch persist to this day. The Internet has not had a congestion collapse since then, so the original fix has withstood the test of time.

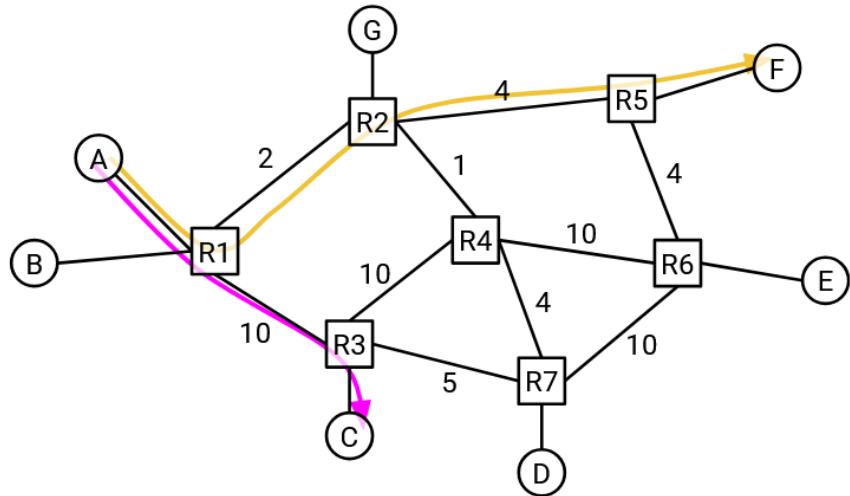
Why is Congestion Control Hard?

To get a sense of why congestion control is a difficult problem, consider the following network graph. At what rate should host A send traffic?



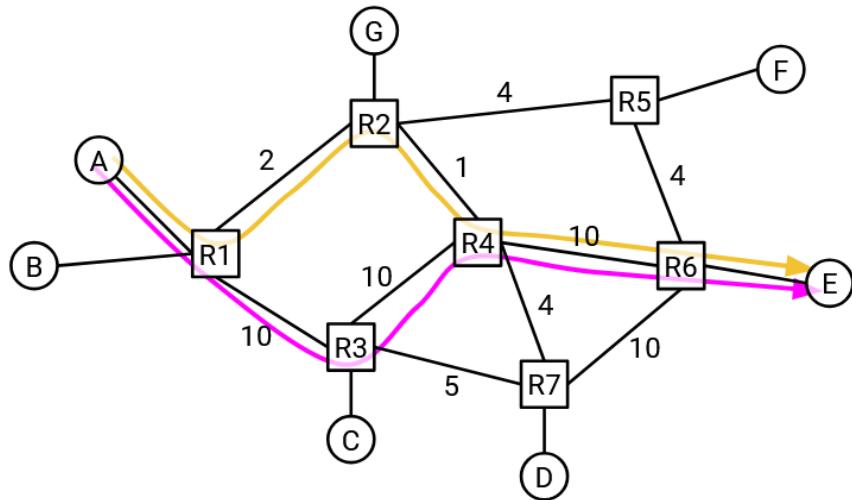
It depends on the destination, so A can't just come up with one fixed rate for all destinations. For example, if A is communicating with C, then A could send packets at 10 Gbps.

What if A is communicating with F instead? The bottleneck link (least capacity) along this path is 2Gbps, so A should probably send packets at 2 Gbps.



What if A is communicating with E?

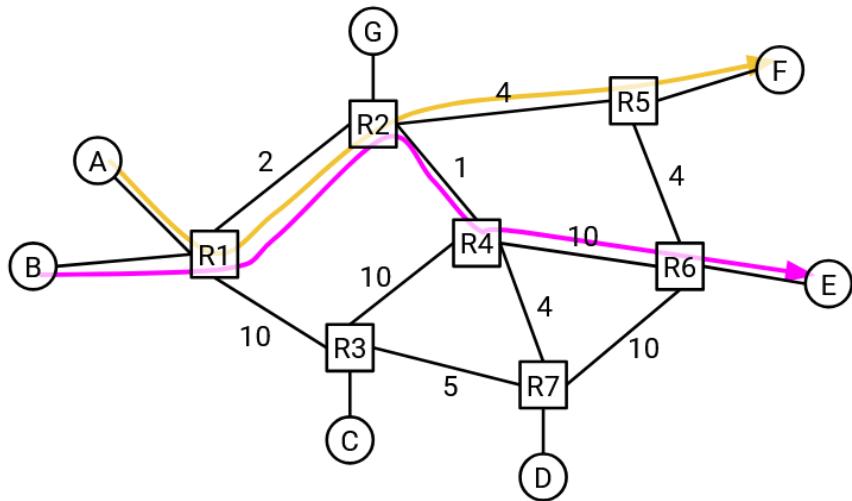
It depends on what path the traffic is taking between A and E. If the traffic is taking the bottom path through R3, then A could send packets at 10 Gbps. But if the traffic is taking the top path through R2, then A can now only send packets at 1 Gbps.



One takeaway so far is that our congestion control algorithm will need to somehow learn about the bandwidths and bottlenecks along the path that the packet is taking.

Also, recall that the network graph changes over time as new links are added or links go down. This means that it's not enough to learn about paths a single time. Our algorithm will need to be adaptive to changes in network topology.

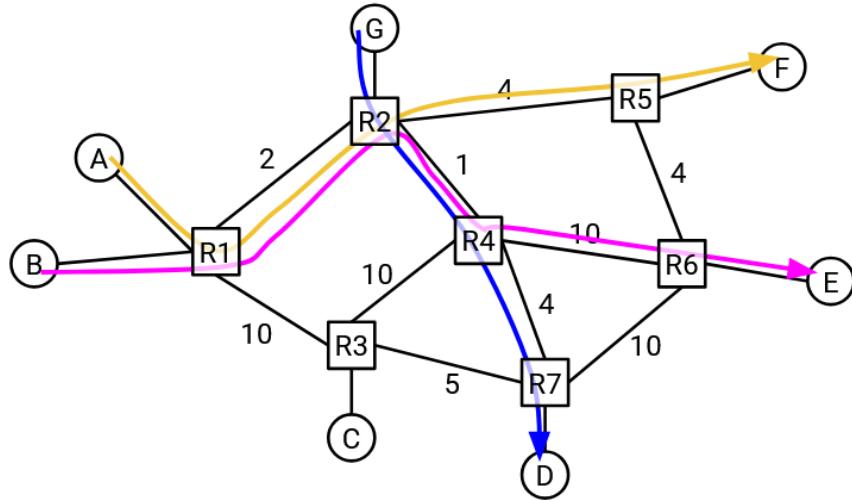
So far, we've assumed that A is the only host sending traffic on the network, and A can use the full capacity of every link. But what if other connections are also using bandwidth?



In this example, A and F have a connection, and B and E have a connection. The two connections seem like they should be totally separate (different senders, different recipients), but in fact, their paths share a link in the network.

If we want the two connections to share the capacity on this link fairly, maybe A and B should each send at 1 Gbps.

What if a new connection starts between G and D? Should A change its rate of 1 Gbps? (No formal algorithm yet, just think about using bandwidth in a way that seems reasonable.)



First, notice that the G-D and B-E connections are sharing a link. This means that these two connections have to slow their rate down to 0.5 Gbps.

Now, if we look back at the 2 Gbps link that A-F and B-E had in common, B-E is only using 0.5 Gbps on this link. This means that A could increase its rate to 1.5 Gbps.

What happened here? The G-D connection was created, and its path has no links in common with the A-F connection. And yet, this seemingly unrelated connection caused the A-F connection's rate to increase. Connections can indirectly affect other connections, even if those two connections don't share any links in common!

In summary: When the sender is trying to determine a rate for sending packets, it has to consider: The destination, the path to that destination, the connections sharing links along that path, and the connections sharing links with those connections (indirect competition), and so on. Congestion control is a hard problem because all the connections in the network are dependent on each other to determine their optimal sending rate.

More fundamentally, congestion control is a resource allocation problem. Bandwidth is a limited resource, each connection wants a certain amount of that resource, and we need to decide how much bandwidth to allocate to each connection.

Resource allocation is a classic problem in computer science. (Examples include CPU scheduling and memory allocation algorithms.) However, unlike some resource allocation problems, a change in one connection's allocation can have a global impact across all other connections. Also, allocations have to change every time a connection is created or destroyed. As a result, congestion control is more complex than the traditional resource allocation problem, and in fact, we don't even have a formal model to define the problem.

Unlike a traditional resource allocation problem, where the algorithm knows about the resource (e.g. CPU time) and the jobs (e.g. processes) ahead of time, there is no global mastermind that can see the entire

network to allocate resources. Our solution has to be decentralized, where every sender decides its own allocation (even though everyone's decisions are highly inter-dependent).

Goals for a Good Congestion Control Algorithm

From a resource allocation perspective, there are three goals we want out of a good congestion control algorithm.

We'd like the resource allocation to be efficient. Links should not be overloaded, and there should be minimal packet delay and loss. Also, links should be utilized as much as possible.

We'd also like the resource allocation to be fair between connections. We'll formalize the definition of fair later, but roughly speaking, every connection should share an equal portion of the available capacity.

We want a solution that achieves a good trade-off between these goals. It would be possible to optimize one goal at the expense of the others, but that leads to bad solutions. For example, we could ensure maximal link utilization by having everyone send packets extremely quickly (bad solution, causes congestion). Or, we could ensure minimal packet loss by making everybody send packets extremely slowly (bad solution, not utilizing capacity).

From a more practical systems perspective, the solution we come up with needs to be scalable and decentralized. Our solution should also be able to adapt to changes in the network (e.g. changing topology, connections being created and destroyed).

Design Space of Solutions

As we saw earlier, Karels and Jacobson fixed TCP congestion control by patching the TCP implementation in the operating system. But, if we could go back and re-design the Internet from scratch, what other possible designs for congestion control exist?

One possible alternate design is based on reservations. The sender could request bandwidth ahead of time, and then free up that bandwidth after the connection is over. As discussed earlier, maintaining a reservation across the entire network comes with many technical difficulties. This approach is also problematic because it assumes that the sender knows what bandwidth it needs ahead of time, which isn't necessarily true.

Another alternate design is based on pricing. As an analogy, consider express toll lanes on the highway (dedicated lanes only available to drivers who pay). The price to use the express toll lane depends on how congested the highway is. When the highway has very few cars, using the toll lane is very cheap, and when there is heavy traffic, using the toll lane is more expensive. Another form of congestion pricing occurs in airplane tickets, which cost more during busier times (e.g. holidays).

To apply congestion pricing to the Internet, your ISP could add a button in your web browser that enables higher Internet speeds for an extra fee, and the fee could change depending on how congested the Internet is. Then, routers can prioritize sending packets from users who are paying more, and drop packets from users who are not paying. Research exists on congestion pricing on the Internet, and economists sometimes claim that if bandwidth is a scarce commodity, then a market structure will lead to an optimal solution. Congestion pricing has not been widely deployed, because it requires some form of business model connecting payments to congestion.

All modern congestion control algorithms (including the ones we'll study) are based on dynamic adjustment. Hosts dynamically learn the current level of congestion, and adjust their sending rate accordingly. In practice, dynamic adjustment is a practical solution because it can be easily generalized. This approach doesn't assume any business model (needed for pricing), and doesn't assume anything about users knowing the bandwidth they need ahead of time (needed for reservations).

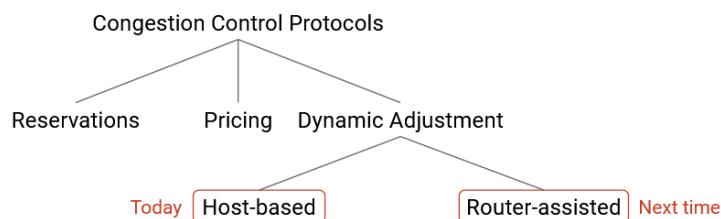
Dynamic adjustment does require good citizenship. TCP needs everybody on the network to work together to share the resources fairly. For example, when a new connection starts using links, other connections need to slow down and share the bandwidth.

Within the dynamic adjustment approach, there are two broad classes of solutions. In **host-based** congestion control algorithms, the sender is monitoring the performance and adjusting its rate accordingly. These algorithms are implemented entirely at the sender, and there is no special support from routers. The modification to TCP is a host-based algorithm, and is widely deployed today.

In **router-assisted** congested control algorithms, routers will explicitly send information about congestion back to the sender, to help the sender adjust its rate. Congestion happens at routers, so routers are in a good position to offer information about congestion. Router-assisted algorithms have been deployed in recent years, especially in datacenters.

Some router-assisted algorithms send very little information, e.g. a single bit indicating congestion, while other algorithms send more detailed information, e.g. the exact rate the sender should use.

Note that in both cases, routers are signaling congestion back to the sender. In router-assisted algorithms, the router is explicitly sending a message about its level of congestion. By contrast, in host-based algorithms, the sender does not receive explicit feedback from the routers. Instead, the sender uses implicit clues from the router (e.g. packets getting dropped or delayed) to deduce that the router is congested.



In this taxonomy of congestion control approaches, we'll focus on the dynamic adjustment approach, and within the space of dynamic adjustment solutions, we'll focus on host-based solutions.

Congestion Control Design

Host-Based Algorithm Sketch

Most host-based algorithms follow the same general approach, and the differences arise in three key choice points.

Each source independently runs the following logic repeatedly, in a loop: Try sending at a rate R for some period of time. Then, ask: Did I experience congestion in this time period? If yes, then reduce R . If no, then increase R .

One missing piece is: what rate do we initially start sending at? We'll need some way to pick the initial rate R .

The three key choice points are: How do we pick the initial rate? How do we detect congestion? By how much should we increase and decrease each time?

Detecting Congestion

How does the sender detect if the network is congested? There are two common approaches.

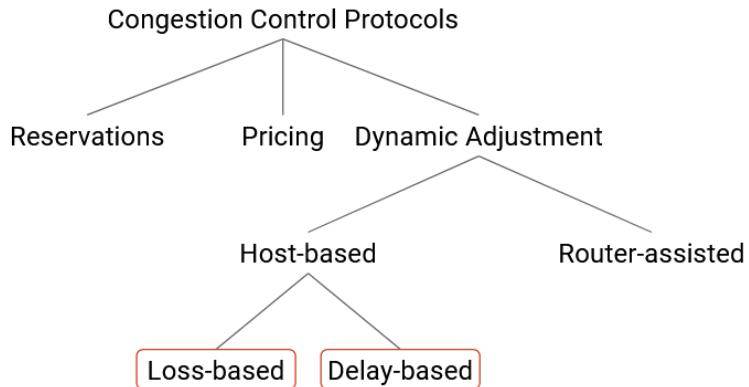
The sender could check for packet loss. This is the approach commonly used by TCP. This approach is good because the signal is unambiguous. Every packet is either marked as lost (timeout or duplicate ack), or not lost. Also, TCP already detects lost packets in order to re-send them, so we don't need to re-implement this from scratch.

This approach can be bad because sometimes, packet loss is due to corruption (bad checksum), not congestion. In fact, TCP gets confused and behaves poorly when a link is not congested, but frequently corrupts packets. Also, TCP could be confused by packets being reordered. A packet arriving late could be mistakenly considered lost.

Another key downside to this approach is, we detect congestion late. By the time packets are being dropped, router queues are already full and packets are being delayed.

Instead of checking for packet loss, the sender could instead detect congestion by checking packet delay. The sender can measure the time between sending a packet and receiving an ack for that packet. If the sender notices that the delay is increasing, that could be a sign of congestion.

Historically, accurately measuring delay has been considered difficult. Packet delay can vary depending on queue size and other traffic. For many years, packet delay was not widely deployed, though in recent years, Google's BBR protocol (2016) has shown that delay-based algorithms are possible, and some services (e.g. Google services) have adopted delay-based algorithms.



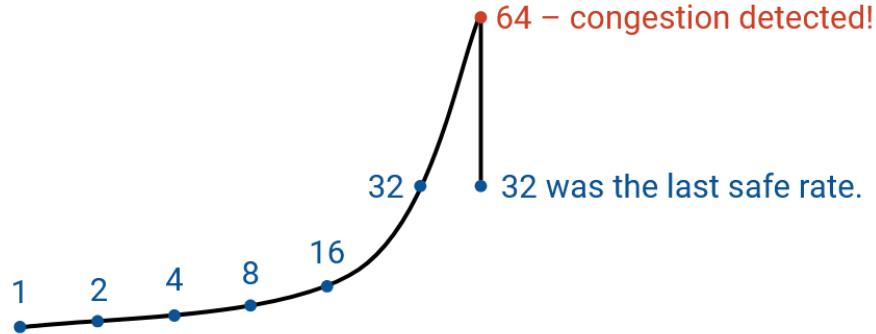
Discovering Initial Rate

When a connection first starts out, we have to figure out an initial rate for sending packets. We can learn an initial rate using a discovery process, where we try a few different rates to get an estimate of the available bandwidth.

We want this discovery process to be safe, so we should start with slow rates. We don't want to immediately flood the network with packets.

At the same time, we want the discovery process to quickly discover the available bandwidth, for efficiency. To achieve this, we will quickly increase the rate on each subsequent try. If the discovery process takes a long time, we've wasted time that we could have spent sending packets at the optimal rate. As an example, suppose we add 0.5 Mbps to the rate every 100ms, until we detect congestion (loss). If the available bandwidth is 1 Mbps, the discovery phase would take 2 iterations = 200 ms. However, if the available bandwidth is 1 Gbps = 1000 Mbps, then the discovery phase would take 2000 iterations = 200 seconds before we ramp up to a good rate, which is far too long. The Internet has a wide variety of link speeds, so both possibilities could occur in real life.

In order to support a slow start but a fast ramp-up, we'll increase the bandwidth by a multiplicative factor each time (instead of an additive factor). This solution is called **slow start**, though this is arguably an unintuitive name. In slow start, we start with a small rate that will almost always be much less than the actual bandwidth. Then, we increase the rate exponentially (e.g. doubling the rate each time) until we encounter loss. A safe rate to use is the one just before we encounter loss (we don't want to use a rate where we experienced loss). Formally, if loss occurs at rate R, then the safe rate is R/2.



Adjustments: Reacting to Congestion

Recall that after the discovery phase, we will be constantly adjusting the bandwidth, because the network itself is changing, and the available bandwidth is not constant.

The final choice point is deciding how much we should decrease the bandwidth if congestion is detected, and how much we should increase the bandwidth if no congestion is detected.

Our decision will determine how quickly a host adapts to changes in the available bandwidth, which in turn determines how effectively bandwidth is consumed. If we took a long time to adapt to changes and find a good rate, we would spend a lot of time operating at sub-optimal bandwidth, which is inefficient. Slow adaptation can also lead to fairness issues. For example, if I'm using a link's entire bandwidth, and another connection is opened, I need to quickly adapt and decrease my bandwidth in order to share the link.

Recall that our main goals in a congestion control algorithm are efficiency (use all available bandwidth) and fairness (connections share bandwidth equally). We will need to choose increase and decrease rules that achieve both of these goals.

What rules can we choose from? At a high level, we can either react quickly or slowly. More specifically, fast changes are multiplicative, e.g. doubling or halving the rate on each iteration. Slow changes are additive, e.g. adding 1 to the rate or subtracting 1 from the rate on each iteration. These options create four possible alternatives:

AIAD: additive increase, additive decrease

AIMD: additive increase, multiplicative decrease

MIAD: multiplicative increase, additive decrease

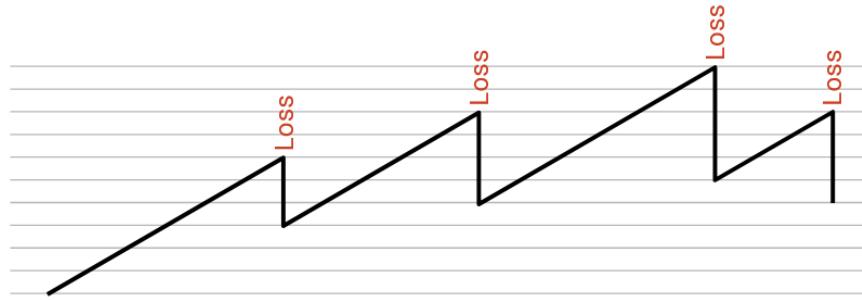
MIMD: multiplicative increase, multiplicative increase

Of these four alternatives, it turns out that AIMD (slow increase, fast decrease) is the best for achieving efficiency and fairness.

Intuitively, AIMD is a reasonable choice because sending too much is worse than sending too little. When our rate is too high, we cause congestion, and packets get dropped. When our rate is too low, we aren't using all the bandwidth, but at least we aren't causing congestion.

AIMD leads to the behavior where we slowly increase the rate when there's no congestion, creeping up to the maximal bandwidth. Then, as soon as we exceed maximal bandwidth and detect congestion, we rapidly

decrease. This way, we spend most of our time with the rate too low (preferable), and when the rate is too high (not preferable), we quickly decrease to avoid congestion.



Adjustments: Model

Why is AIMD the best choice for achieving efficiency and fairness? Let's do a more detailed analysis.

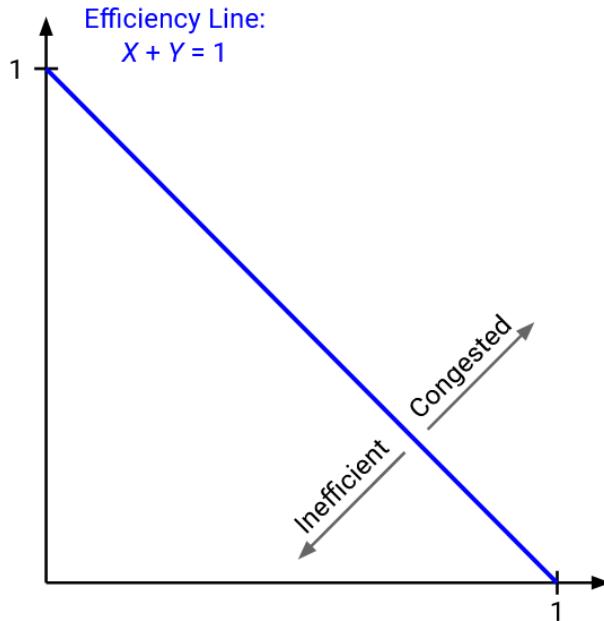
First, notice that all four options do a pretty good job at achieving efficiency. By increasing when we're below the optimal rate (not congested), and decreasing when we're above the optimal rate (congested), our rate should always be hovering around the optimal rate in the long run.

However, it turns out that of these four options, AIMD is the only option that leads to fairness. To see why, let's consider a simple model where there are two connections going over a single link of capacity C . The two connections are sending at rates X_1 and X_2 , respectively. We know that if X_1+X_2 is greater than C , the network is congested, and if X_1+X_2 is less than C , then the network is underloaded.

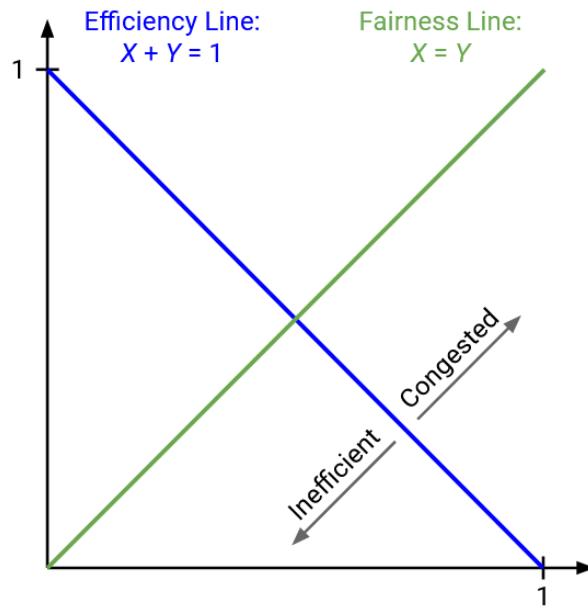
To achieve efficiency, we want the link to be fully utilized, i.e. $X_1+X_2 = C$. To achieve fairness, we want $X_1 = X_2$, so that both connections are sharing the capacity equally.

To visualize the space of possibilities, consider a 2D plot, where the x-axis is X_1 (user 1's rate), and the y-axis is X_2 (user 2's rate). Every point on this plot represents a possible scenario where each user is sending at a specific rate.

Suppose $C=1$. To achieve maximum efficiency, we want $X_1+X_2 = 1$. We can plot this line on the graph. Every point along this line is using the full available bandwidth.



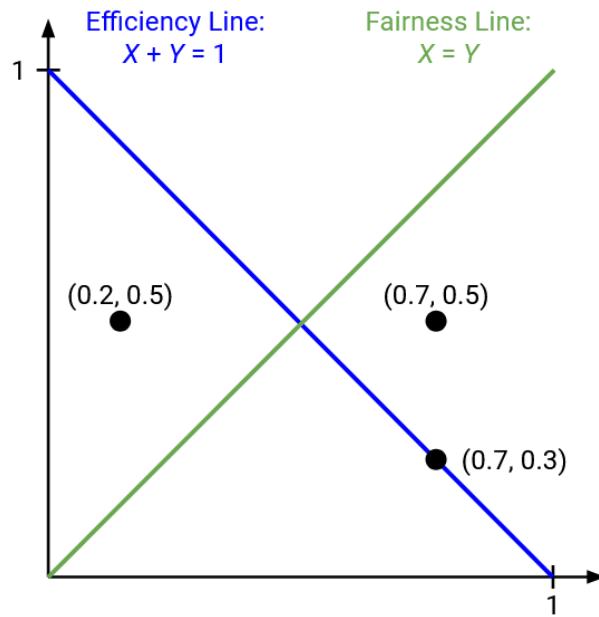
We know that the network is congested when X_1+X_2 is greater than 1. On the plot, this inequality is the half-plane above the line. We also know that the network is underused when X_1+X_2 is less than 1, which is represented by the half-plane below the line. This means that all points above the line represent a congested state, and all points below the line represent an underused state.



To achieve fairness, we want $X_1 = X_2$. We can also plot this line. Every point along this line represents a fair state, where both users are using the same amount of bandwidth. Any point not along this line is unfair.

The ideal state occurs at the intersection of the two lines, when $X_1 = X_2 = 0.5$. This point falls on both lines, so it is both fair and efficient.

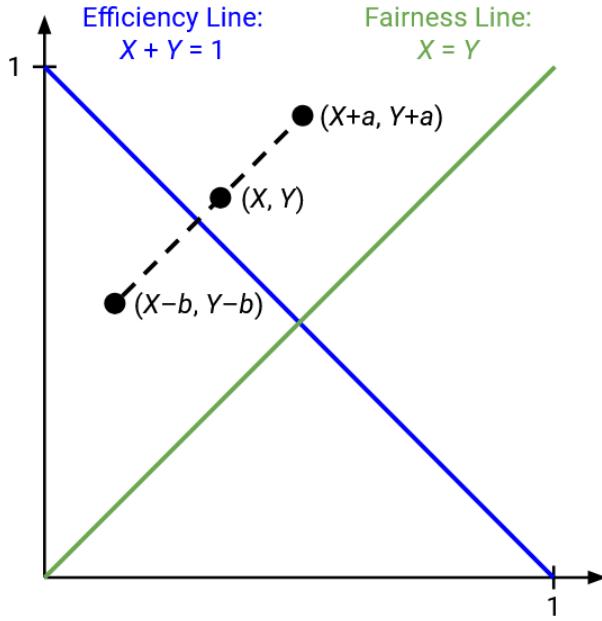
The point $(0.2, 0.5)$ is inefficient, because we are only using 0.7 bandwidth. Graphically, we are below the efficiency line. The point $(0.7, 0.5)$ is congested and therefore above the efficiency line. The point $(0.7, 0.3)$ is efficient (on the efficiency line), but is not fair (not on the fairness line).



Recall that in our dynamic adjustment algorithm, every sender is independently running the same algorithm to determine their own rate. This means that if the two users detect underuse, both will increase their rate in the same way (additive or multiplicative, depending on our choice of rule). Similarly, if the two users detect congestion, both will decrease their rate in the same way.

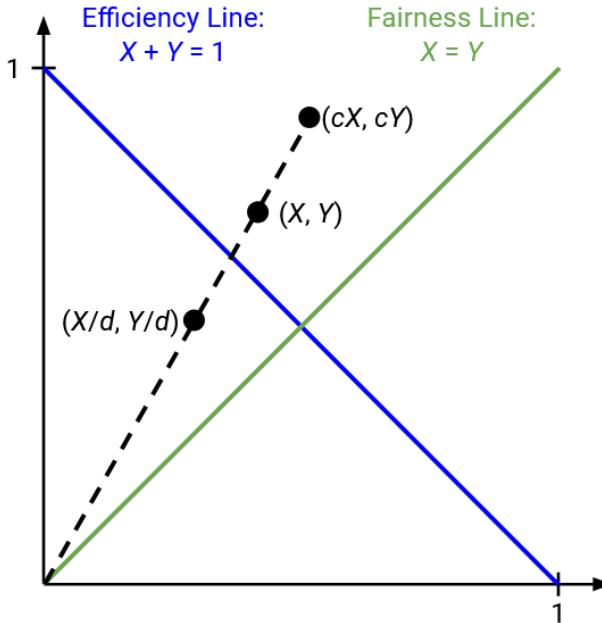
What happens if both users additively increase or decrease their rate? If both users increase their rate by adding b , the state (x_1, x_2) would become (x_1+b, x_2+b) . If both users decrease their rate by subtracting a , the state (x_1, x_2) would become (x_1-a, x_2-a) .

On the graph, if we make an additive change, the point representing our state moves along a line with a slope of 1.



What happens if both users multiplicatively increase or decrease their rate? Multiplying by c transforms (x_1, x_2) to (cx_1, cx_2) , and dividing by d transforms (x_1, x_2) to $(x_1/d, x_2/d)$.

On the graph, if we make a multiplicative change, the point representing our state moves along a line with slope x_2/x_1 . Equivalently, this is the line connecting (x_1, x_2) to the origin $(0, 0)$.



Now, we can apply this model to each of the four increase/decrease options, and see if they cause the point to approach, or move away from, the fairness line. Our goal is for the point to approach the fairness line as we adjust the rates.

Adjustments: AIAD Dynamics

Consider adding 1 on each increase, and subtracting 2 on each decrease. Suppose we have capacity of $C = 5$. Then from a given starting point, our point would move as follows:

$X_1 = 1, X_2 = 3$ (starting point, 4 less than 5, increase)

$X_1 = 2, X_2 = 4$ (6 more than 5, decrease)

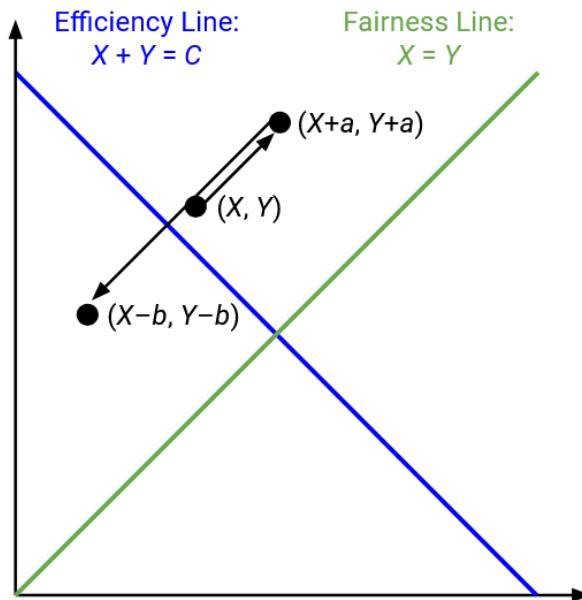
$X_1 = 0, X_2 = 2$ (2 less than 5, increase)

$X_1 = 1, X_2 = 3$

We've returned to where we started! Our initial allocation was not fair, and after a few iterations, we returned to the same unfair allocation.

In fact, if we look at the difference between X_1 and X_2 (fair gap is 0), the gap is the same (2) in every iteration. The iterations don't make our allocation any more or less fair.

We can see this behavior graphically. From a given starting point, if we increase and decrease additively, we will always move along a line of slope 1, never getting any closer to the fairness line.



Do note, though, that our point oscillates around the efficiency line, as desired. All four options will have this behavior.

We can also see this behavior algebraically. Suppose X_1 and X_2 are 5 apart (unfair allocation). If we add the same number to X_1 and X_2 , the resulting X'_1 and X'_2 are still 5 apart (equally unfair). The same happens if we subtract the same number from both X_1 and X_2 .

In summary, there is no way to close the fairness gap in this approach. If the allocation is initially unfair, it will stay unfair.

You might ask: What if we increased X1 by more (e.g. +2), and X2 by less (e.g. +1)? Remember, our decentralized approach means that everybody is running the same algorithm. Practically, we also have no way for a host to know how much it should add relative to other hosts.

Adjustments: MIMD Dynamics

Consider increasing by doubling, and decreasing by dividing by 4. Again, the capacity is $C = 5$. From a given starting point, the first few iterations would be:

$X_1 = 0.5, X_2 = 1$ (1.5 less than 5, increase)

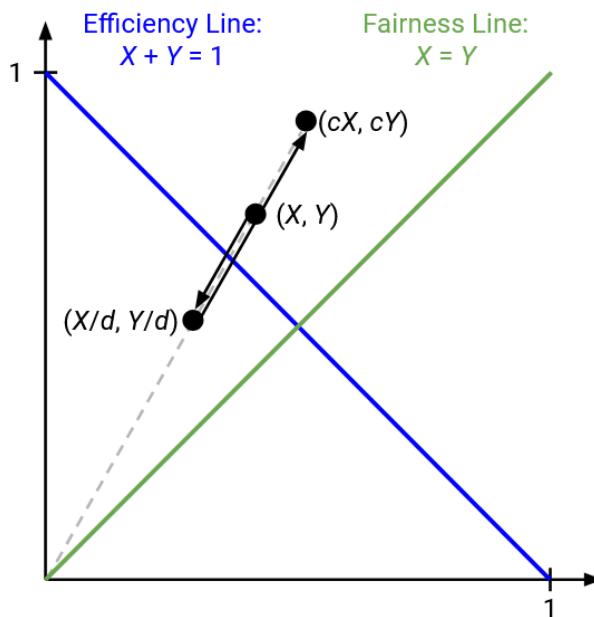
$X_1 = 1, X_2 = 2$ (3 less than 5, increase)

$X_1 = 2, X_2 = 4$ (6 more than 5, decrease)

$X_1 = 0.5, X_2 = 1$

Again, we've returned to where we started, with no improvement in fairness!

We can see this behavior on the plot. When we multiplicatively increase or decrease the rate, we are moving along the line between the point and the origin, and we are never getting any closer to the fairness line.



Algebraically, consider the ratio between X_2 and X_1 , i.e. X_2/X_1 (fair ratio would be 1). In the example above, the ratio is always 2, i.e. X_2 always has twice the bandwidth of X_1 . This ratio stays the same even if we multiply or divide both X_1 and X_2 by constant factor. Our adjustments don't get us closer to a fair ratio of 1.

Adjustments: MIAD Dynamics

This one is a little trickier. Consider increasing by doubling, and decreasing by subtracting 1. With $C = 5$, the first few iterations are:

$X_1 = 1, X_2 = 3$ (4 less than 5, increase)

$X_1 = 2, X_2 = 6$ (8 more than 5, decrease)

$X_1 = 1, X_2 = 5$ (6 more than 5, decrease)

$X_1 = 0, X_2 = 4$ (4 less than 5, increase)

$X_1 = 0, X_2 = 8$

At this point, X_1 has zero bandwidth. Every time we increase by doubling, X_1 will still have zero bandwidth. We have actually created the most unfair situation, where X_2 has all the bandwidth, and X_1 has none.

More generally, if you start with an unfair allocation, MIAD will make the allocation even more unfair, eventually reaching a point where one person has all the bandwidth, and the other person has zero.

To see this algebraically, consider the gaps between X_1 and X_2 . When we increase by doubling, the size of the gap also doubles, from $(X_2 - X_1)$ to $(2X_2 - 2X_1) = 2(X_2 - X_1)$. But, when we subtract 1 from both X_1 and X_2 , the gap stays the same. The gap either increases or stays the same, and given enough iterations of increasing and decreasing, the gap will reach maximal unfairness (one person has zero bandwidth forever).

Adjustments: AIMD Dynamics

Finally, consider increasing by adding 1, and decreasing by halving. With $C = 5$, the first few iterations are:

$X_1 = 1, X_2 = 2$ (3 less than 5, increase)

$X_1 = 2, X_2 = 3$ (5 not more than 5, increase)

$X_1 = 3, X_2 = 4$ (7 more than 5, decrease)

$X_1 = 1.5, X_2 = 2$ (3.5 less than 5, increase)

$X_1 = 2.5, X_2 = 3$ (5.5 more than 5, decrease)

$X_1 = 1.25, X_2 = 1.5$ (2.75 less than 5, increase)

$X_1 = 2.25, X_2 = 2.5$ (4.75 less than 5, increase)

$X_1 = 3.25, X_2 = 3.5$ (6.75 more than 5, decrease)

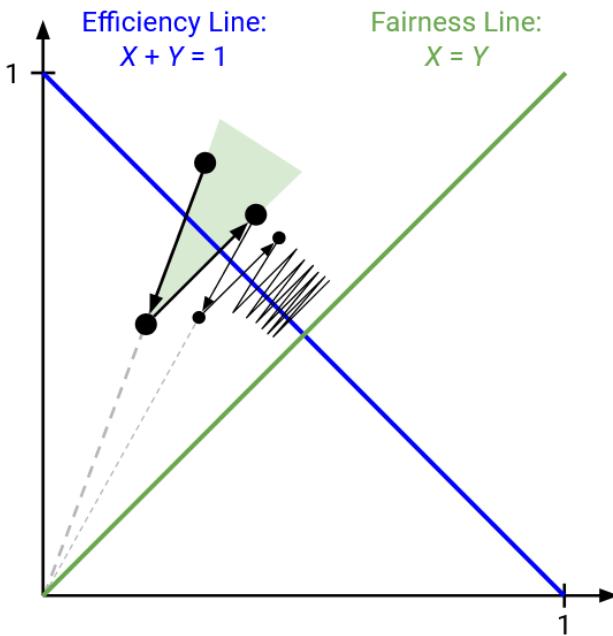
$X_1 = 1.625, X_2 = 1.75$ (less than 5, increase)

$X_2 = 2.625, X_1 = 2.75$

We can see that X_1 and X_2 are getting closer together, and in fact, they're approaching the fair allocation of $X_1 = X_2 = 2.5$.

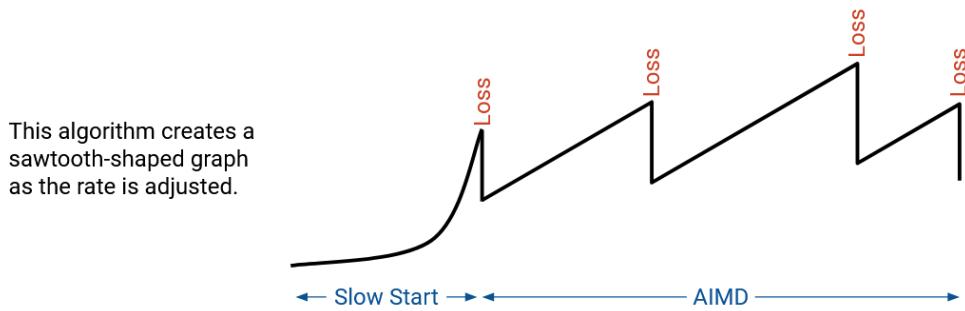
Algebraically, we can see that the gap between X_1 and X_2 is decreasing. Specifically, when we add a constant to both numbers, the gap stays the same. But, when we halve both numbers, the gap also halves,

from $(X_1 - X_2)$ to $(X_1 / 2 - X_2 / 2) = (X_1 - X_2) / 2$. As we alternate increasing and decreasing, the gap will keep halving and approaching 0.



We can see this graphically as well. When we multiplicatively decrease, we are moving along the line through the origin. This line is angled toward the fairness line, and moving downwards along this line means we're approaching the fairness line. As before, additive increases don't get us any closer to the fairness line, since we're moving along a line with slope 1 (parallel to fairness line). But the key realization is that adding don't move us any further, either. Our only two operations are moving closer, or not getting closer or further away. After many iterations, our point will slowly move closer toward the fairness line.

In summary: AIAD and MIMD retain unfairness, and make no improvements toward fairness. MIAD increases unfairness, and AIMD converges toward fairness.



Congestion Control Implementation

Recap: TCP Windows

So far, we've designed a conceptual sketch of a dynamic adjustment, host-based algorithm, where each source runs the same algorithm independently to arrive at an efficient, fair share of bandwidth.

First, use slow-start (start at low rate, exponentially increase) to discover an initial rate. Then, in each iteration, if we detect congestion (detect loss), we reduce R multiplicatively. If we don't detect congestion, we increase R additively.

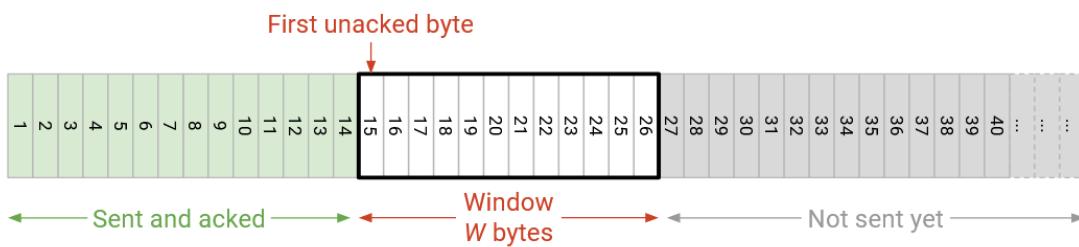
In this section, we will now see how TCP implements this algorithm. For better or worse, TCP's congestion control mechanisms are very intertwined with TCP's reliability mechanisms. (This is a result of the original design, where TCP was patched to account for congestion.) In this section, we'll see how TCP's implementation works to achieve both reliability and congestion control at the same time.

Recall that in TCP, the sender maintains a sliding window of consecutive bytes/packets in flight. The size of the window is determined by flow control (decided by buffer space at recipient) and congestion control (rate computed by sender).

More specifically, in flow control, the recipient sends an advertised window, indicating how many more bytes can be sent without overflowing the recipient's memory. This advertised window value is sometimes abbreviated **RWND (receiver window)**.

In congestion control, the sender maintains a value, sometimes abbreviated **CWND (congestion window)**, which denotes the rate the sender can send packets without overloading links. This value will be dynamically set and adjusted by the congestion control algorithm.

The sender's window is computed as the minimum of CWND and RWND. For this lecture, we'll assume that RWND is larger than CWND, so the bottleneck is the network, not the recipient's memory. This is usually, but not always true in practice.



Recall that we can view the sliding window as a range in the bytestream. The left side of the window is the first unacknowledged byte (everything to the left of the window has already been sent and acknowledged). The right side of the window is determined by the window size. Only packets inside this window are allowed to be in-flight.

When data on the left side of the window is acked, the window slides to the right, and additional data can now be sent.

To detect loss, we maintain a single timer for the left-most packet in the window. If the timer expires

without that packet being acked, we re-send the left-most packet in the window. Also, to detect loss, we count the number of duplicate acks, and re-send the left-most packet if we see 3 duplicate acks. This duplicate ack-based approach is sometimes called **fast retransmit**.

Windows and Rates

How do we adjust the rate for congestion control, and how do we compute the congestion window? It turns out that these two values are directly related, and adjusting the window is achieved by adjusting the rate. The window size and the rate of sending data are correlated by the following equation: rate times RTT = window size.

Intuitively, you can think of window size and rate as the same quantity, expressed in two different “units of measurement.” An increased window size means we’re sending data faster, and vice-versa.

To see why this equation holds, consider the first RTT. We can send [window size] number of packets during this first RTT (before any acks arrive), for a rate of window size / RTT.

Recall that our conceptual TCP design measured data in packets for simplicity, but in practice, TCP thinks in terms of bytes. In a real implementation, the window size is measured in terms of bytes, but for simplicity, we will consider the window size in terms of packets.

To convert between packets and bytes, recall that we defined the Maximum Segment Size (MSS), the number of bytes per packet. This tells us that MSS times number of packets = number of bytes. Again, intuitively, you can think of bytes and packets as two different units of measurement for the same quantity (amount of data).

Event-Driven Updates

In our conceptual model, our goal is to adjust the rate/window once per “iteration,” but we haven’t formalized how to measure each iteration. We can roughly define each iteration as one RTT, but the RTT itself is a dynamically changing value that we can’t accurately measure.

In order to update the window size in a more predictable, measurable way, we can consider the various events that the existing TCP implementation responds to, and update the window each time one of these events occurs. These are called **event-driven updates**.

The three TCP events where we need to update the window size are: new ack, 3 duplicate acks, and timeout.

When we see a new ack (for data that was previously not acknowledged), this is a sign that our data made it through the network without loss. In our model, we detect congestion by checking for loss, so a new ack is a sign that the network is not congested. Therefore, when we see a new ack, we can increase the window size (either during slow-start discovery, or AIMD adjustment).

When we see 3 duplicate acks, we mark a packet lost. This is a signal of isolated loss, which indicates mild congestion. We lost a packet, but subsequent packets are still being received. To react to this loss, we will decrease the window size (during AIMD adjustment).

When we encounter a timeout, we mark a packet lost. The fact that we detected the loss after a timeout, not duplicate acks, is a signal of many packets being lost (heavy congestion). To see why, consider a window

size of 100 packets. If we encounter a timeout, this means we didn't get an ack for the left-most packet in the window. But it also means that we failed to get 3 duplicate acks for any other packets in the window during the entire duration of the timer. A timeout means that very few, if any, packets are being received, and something bad has happened.

If we discover a timeout, something unexpected has happened (e.g. network changed), and we should no longer trust our current window size. To react, we should go back to the slow-start phase and re-discover a good window size. This isn't the only way to react to timeout, but this is what TCP decided.

Event-Driven Slow Start

In our conceptual model, we implemented slow start by choosing a slow rate, and increasing the rate exponentially (e.g. doubling on each iteration) until we encounter the first loss. We now need an event-driven way to double the window once per RTT.

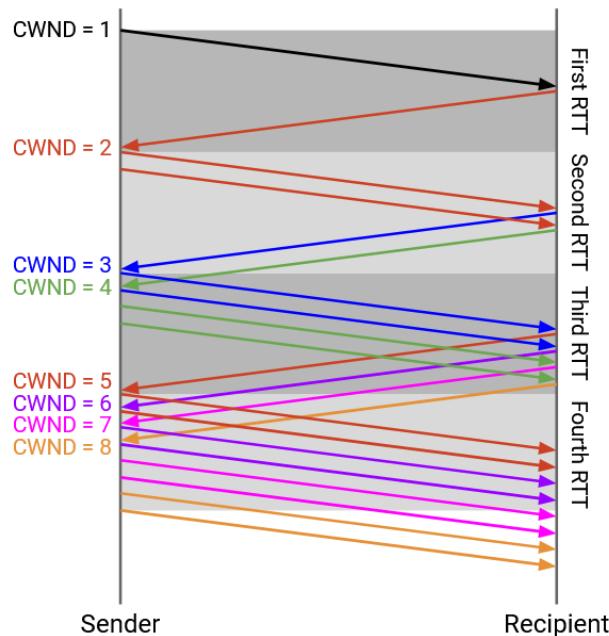
TCP starts with a small window of 1 packet. Recall, we can convert packet to bytes with the maximum segment size (MSS), and then convert bytes to rate by dividing MSS/RTT.

Every time we get an acknowledgement, we will increase the window size by 1 packet. The intuition for what will happen is:

Initially, the window size is 1 packet. We send 1 packet, and after an RTT, get 1 ack back. The ack lets us increase the window to 2 packets.

We now send 2 packets, and after an RTT, we get 2 acks back. The 2 acks let us increase the window by another 2 packets, for a new window size of 4 packets.

We now send 4 packets, and after an RTT, we get 4 acks back. The 4 acks let us increase the window by another 4 packets, for a new window size of 8 packets.



This intuitive picture assumes we are sending all 4 packets and receiving all 4 acks simultaneously, though. In practice, the sliding window behavior causes our window to increase by 1 each time we receive an ack, though the end behavior (doubling the window every RTT) is the same.

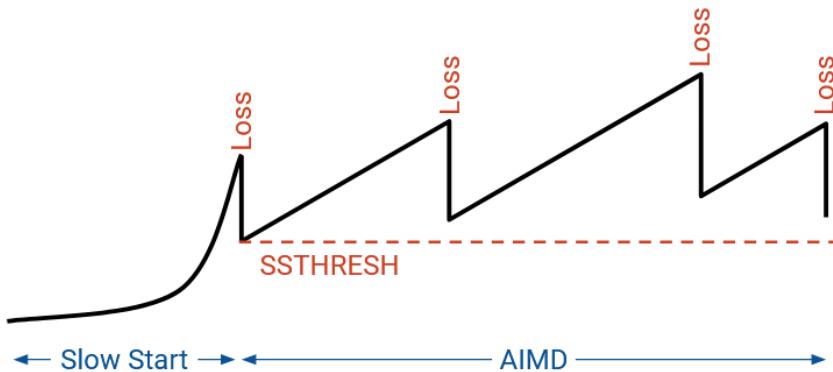
As before, we start with a window size of 1 packet. We send 1 packet (A), and after an RTT, get the ack for A back. The ack lets us increase the window to 2 packets, and there are zero packets in flight.

Next, we can send out 2 packets (B and C). When we get the ack for B, we increase the window to 3 packets. There is still 1 packet in flight (C), so we can send 2 more packets (D and E).

When we get the ack for C, we can increase the window to 4 packets. There are still 2 packets in flight (D and E), so we can send 2 more packets (F and G).

In general, assuming no loss and no re-ordering, every time we receive an ack, the sliding window allows us to send one more packet, and the increased window allows us to send another packet. Because every ack leads to 2 packets being sent, we get the behavior where the window doubles every RTT. For example, within an RTT interval where we receive 16 acks, each ack triggers 2 packets being sent, for a total of 32 packets. Then, in the next RTT interval, those 32 packets will be acked, triggering 64 packets being sent.

Eventually, after some time spent doubling the window every RTT (increasing the window by 1 for every ack), we'll encounter loss. This also means we've learned the maximum allowable "safe" rate for sending packets without encountering loss. We'll remember this rate in a new parameter called Ssthresh (slow start threshold). Specifically, as soon as we encounter packet loss, we'll set Ssthresh to half the window size. For example, if a window of 16 packets doesn't cause loss, but a window of 32 packets does cause loss, then we would set Ssthresh to 16.



Recall that after slow start, we will continually adjust the window size (AIMD). Ssthresh lets us remember the safe rate we learned from slow start, even as the rate starts changing later.

Implementing Additive Increasing

In our conceptual model, after slow start, we want to slowly (additively) increase the rate when there's no loss. We need an event-driven way to increase the window by 1 packet for each RTT.

We don't have an exact number for the RTT, but we do know that within a single RTT, we expect a window's worth of packets to be acked. For example, with window size 10, we receive 10 acks per RTT. If we increase the window by 1/10 packet per ack, then across a RTT, the window should increase by 1 packet, as desired.

Each time we receive an acknowledgement, we will take the current window size CWND and reassign it to CWND + (1/CWND). This increases the window by a fraction of a packet on each ack. After a full window's worth of packets (i.e. after one RTT), the window increases by 1 packet.

Formally, TCP measures the window in bytes, not packets, so (1/CWND) is equivalent to $\text{MSS} * (\text{MSS}/\text{CWND})$ in bytes. In (1/CWND), the numerator is 1 packet (total increase in an RTT), and the denominator is CWND measured in packets. Since the denominator is now measured in packets, we also have to measure the numerator in packets: 1 packet = MSS bytes.

But the fraction 1/CWND or MSS/CWND is still a ratio (dimensionless), representing the fraction to be increased on each ack. The total increase we want is 1 packet = MSS bytes, so we have to multiply this fraction by MSS bytes.

As an example, suppose our CWND was 3 packets = 150 bytes (assuming MSS = 50 bytes). In the packet view, we would add 1/3 packets to the window each time, for a total increase of 1 packet.

In the byte view, we can divide $\text{MSS}/\text{CWND} = 50/150$ to get the same 1/3 ratio that we need to step by each time, for a total increase of 1. But we still need to multiply by MSS so that the total increase is MSS instead of 1.

	Measured in packets:	Measured in bytes: ($\text{MSS} = 50$ bytes)
Old CWND:	3 packets	150 bytes (3 packets)
Fraction of increase per ack:	$1/\text{CWND} = 1/3$	$\text{MSS}/\text{CWND} = 50/150$
Total increase after 1 RTT:	+1 packet	$\text{MSS} = +50$ bytes
Increase after each ack:	$+1 \text{ packet} \times 1/3 = +1/3$	$+50 \text{ bytes} \times 50/150 = +50/3$
Increase after 3 acks:	$3 + (1/3) + (1/3) + (1/3) = 4$ packets	$150 + (50/3) + (50/3) + (50/3) = 200$ bytes

Note that the increase isn't perfectly linear, but provides a good enough approximation. For example, starting with CWND = 4, the first update is $4 + 1/4 = 4.25$, and the second increase is $4.25 + 1/4.25 = 4.49$. After four updates, the window size would be 4.92 in this approximation (we wanted it to be 5 in the exact model).

Implementing Multiplicative Decrease

If we detect loss from 3 duplicate acks, we divide the window size by 2.

Recall that if the retransmission timer expires, we interpret the timeout as multiple packets being lost (we didn't even get duplicate acks). We assume that the current window might be way off, and in order to be cautious, we'll rediscover a good rate from scratch.

First, we'll make a note that the current rate is too high, and the best known safe rate is half of our current rate (following the multiplicative decrease principle). To record this safe rate, we'll set STHRESH to half the current window.

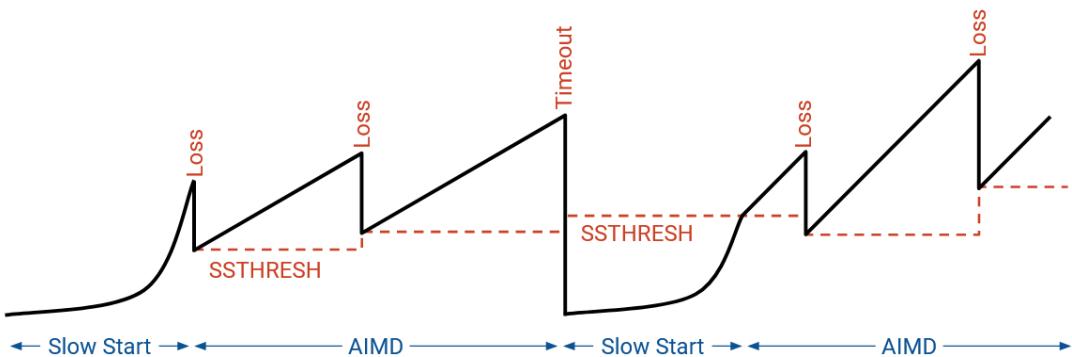
Then, we'll set the window size back to 1 packet, and repeat the slow start process again.

Note that when we re-try slow start, we need to be careful not to return to the dangerous rate with timeouts from earlier. Fortunately, we set Ssthresh to be just below the dangerous rate. Therefore, in subsequent slow start re-trys (where Ssthresh is set), as soon as our window exceeds Ssthresh, we should switch from multiplicative to additive increasing. On the first slow start, Ssthresh is unset (or infinity).

To summarize: In slow-start, we increase the window by 1 packet for each ack (results in doubling the rate on each RTT). When in AIMD, we increase the window by a fraction of the window size for each ack (results in increasing by 1 for each window's worth of data). We decrease the window by halving it when receiving 3 duplicate acks, and changing it to 1 on a timeout.

Note that when decreasing, we never drop the window size to less than 1 packet. In the worst case, we need to allow 1 packet to be in-flight.

TCP Sawtooth



If we plot rate over time, we see the initial exponential growth (slow start). As soon as we experience loss, we cut the rate in half, and switch to AIMD mode. Now, we increase linearly until we encounter loss, and we halve the rate each time we encounter loss.

Fast Recovery: Running Example

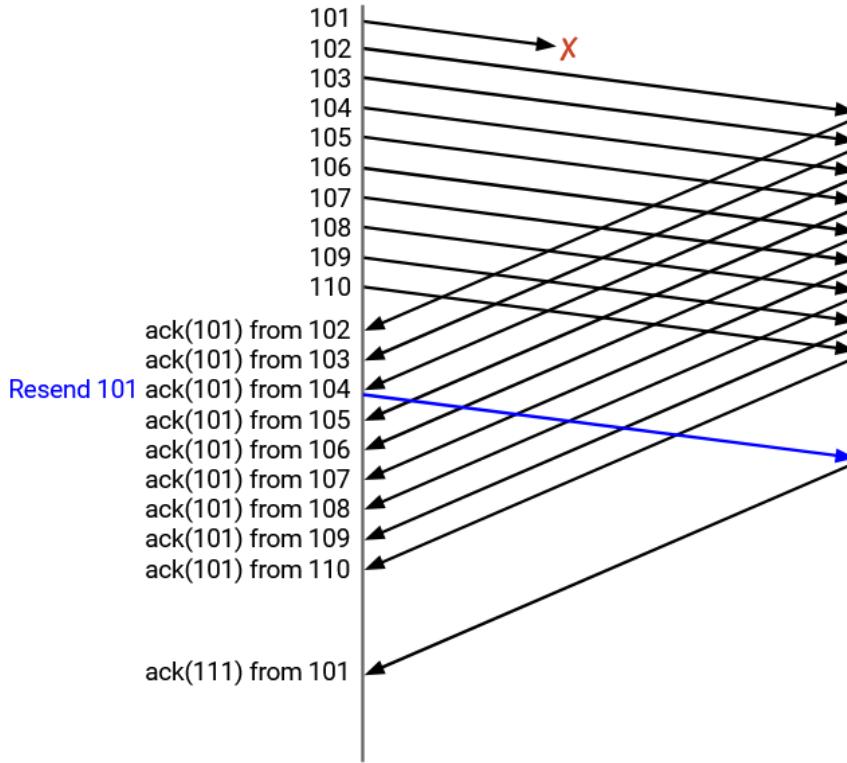
There's one final problem we have to deal with in our congestion control implementation. When we encounter an isolated packet loss, the congestion window is halved, as intended. However, this has the unintended side effect of causing the sender to stall for some time before it can continue sending packets.

To see this in action, let's consider a running example. We send 10 packets, numbered 101 through 110. The first packet (101) is dropped.

As a result, the other 9 packets, 102 through 110, are all acked as ack(101), because the next expected byte is still 101.

After the third duplicate ack(101) (generated by receiving 102, 103, and 104), the sender re-sends 101.

Eventually, the ack for the re-sent 101 arrives. It says ack(111), because packets 102 through 110 were all received earlier, and with the receipt of 101, the next expected byte is 111.



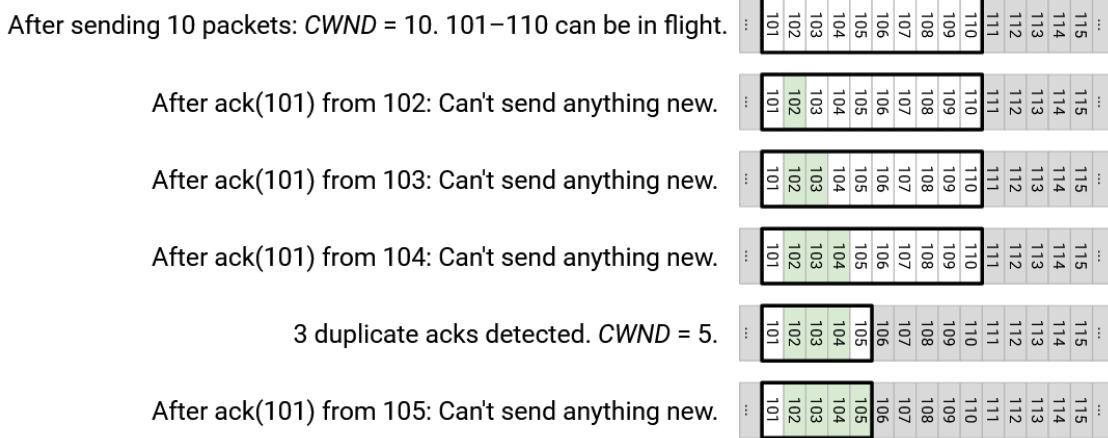
To summarize: At the sender's end, we send 101 through 110, and 101 gets dropped. We get ack(101) from 102, ack(101) from 103, and ack(101) from 104. At this point, we re-send 101. Then, we get ack(101) from 105 through 110. Finally, we eventually get ack(111) from 101.

At the recipient's end, we receive 102 through 110, and send back ack(101) each time, since the next unreceived byte is still 101. Eventually, we receive the re-sent 101, and we send back ack(111) because the next unreceived byte is 111.

What does CWND look like during this running example? Remember that the window starts at the first unacknowledged byte, and extends for CWND contiguous bytes. The only way to shift the window forward is to receive the first unacked byte. If we receive acks for some other bytes in the window, the window stays the same, because the window is determined by the first unacked byte.

Fast Recovery: The Problem

Let's assume that CWND starts at 10. Packets 101 through 110 are allowed to be in flight. The sender sends 101 through 110, but 101 is dropped.



The sender sees ack(101), generated from the other side receiving 102. At this point, the first unacked byte is still 101, so the window stays unchanged. The only packets allowed to be in flights are still 101 through 110, and the sender cannot send anything new (e.g. 111 can't be sent).

Next, the sender sees ack(101), generated from the other side receiving 103. Again, the first unacked byte is still 101, so the window is unchanged. The window still starts at 101 and extends to 110, and the sender can't send anything new.

Next, the sender sees ack(101), generated from the other side receiving 104. This is the third duplicate ack, so we must decrease CWND to 5. The first unacked byte is still 101, and CWND is 5, so packets 101 through 105 are allowed to be in flight. The sender still can't send anything new. We re-send 101 (left-most packet in window) because we saw the third duplicate ack.

Next, the sender sees ack(101), generated from the other side receiving 105. The window is still 101 (first unacked byte) through 105 (CWND bytes later), so we can't send anything new.



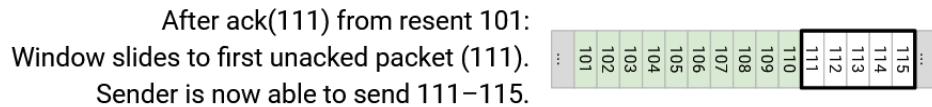
Next, the sender sees ack(101), generated from the other side receiving 106. Again, the window doesn't change, and we can't send anything new.

The sender gets ack(101), ack(101), ack(101), ack(101) from the other side receiving 107, 108, 109, 110. In every case, 101 is still the first unacked byte, so the window is still 101 through 105, and the sender can't send anything new.

What happened here? Only a single packet was dropped, but as a result, the sender had to completely stop sending for a long time.

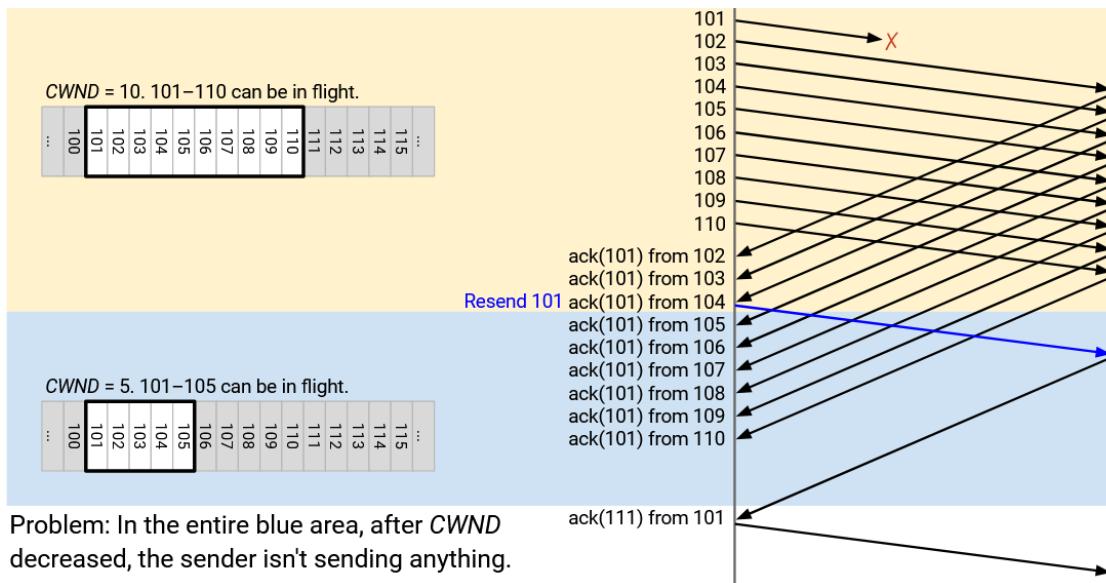
The window is defined by the first unacked byte, so the window refuses to move forward until 101 is re-sent and acked. Even though all the other packets (102 through 110) arrive, the window is still stuck at 101, and later packets (111 onward) can't be sent. The sender is stalled!

Eventually, the sender receives ack(111) from the re-sent 101. This causes the window to leap forward and slide to the new first unacked packet, 111. CWND is still 5, so the sender is now able to send 111 through 115.

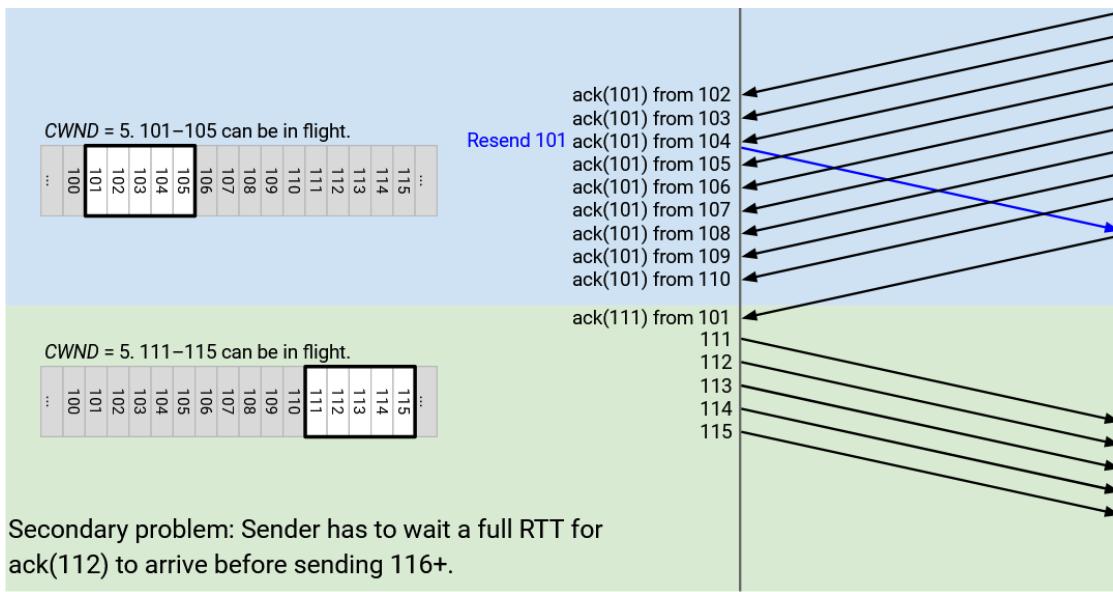


What happened here? We now have a secondary problem. The sender stalled for a long time, but as soon as 101 was acked with ack(111), the window leapt forward all the way to 111-115, and the sender suddenly has to scramble to send 111-115 all at the same time.

The sender stalled for a long time, sending nothing, and then suddenly scrambled to send 111-115 at the same time. Now, the sender has to wait another full round-trip for 111-115 to get acked, before it can send 116 and beyond.



In summary: The isolated packet loss caused the window to get stuck, which causes the sender to stall and send nothing. Eventually, when that packet is re-sent and acked, the window leaps forward, causing the sender to scramble and send a bunch of new packets at once. The sender now has to wait another round-trip for those new packets to get acked, before it can resume business as usual.



A few notes about this problem:

If the problem is still kind of hard to grasp, it might help to note that this problem is more due to the TCP sliding window scheme, and not really due to the congestion control scheme. Congestion control causes the window to shrink, but even if the window didn't shrink, the sender would still be forced to stall until 101 is received and the window leaps forward.

When we think about this problem intuitively, it helps to draw the diagrams of the sender's window, marking off the bytes that have been acked. For example, after the triple duplicate acks, we mark 102, 103, 104 as received, and the window allows 101 (first unacked byte) through 105 to be in flight.

However, this isn't really what the sender sees. Remember, the sender only sees cumulative acks, so it doesn't actually know that 102, 103, and 104 were received. The sender can deduce that 3 packets in the window (that are not 101) were received, but it doesn't know which 3 packets exactly.

Finally, note that after we get 3 duplicate ack(101) messages, we re-send 101, and we never re-send 101 again, even if more duplicate ack(101) messages come in. This is just the TCP rule for re-sending on duplicate acks.

Fast Recovery: The Idea

So, how do we solve this problem? Ideally, we don't want the sender to stall, and we want the sender to keep sending later packets (111 onwards), even if 101 gets lost.

Notice that even though the sender can't deduce exactly which packets arrive, the sender can deduce that later (non-101) packets are getting received.

When we see ack(101), generated from 102 being received, we don't actually know that 102 was received, but we know some packet (non-101) got received. Therefore, only 9 packets remain in flight.

When we see another ack(101), generated from 103 being received, we again don't know that 103 specifically was received, but we know that another packet (non-101) got received. Therefore, only 8 packets remain in

flight.

As we keep receiving duplicate ack(101) messages, we can deduce that fewer packets remain in flight:

After ack(101) from 102: 9 packets in flight.

After ack(101) from 103: 8 packets in flight.

After ack(101) from 104: 7 packets in flight.

After ack(101) from 105: 6 packets in flight.

After ack(101) from 106: 5 packets in flight.

After ack(101) from 107: 4 packets in flight.

After ack(101) from 108: 3 packets in flight.

After ack(101) from 109: 2 packets in flight.

After ack(101) from 110: 1 packet in flight.

Eventually, after we get ack(101) nine times (from 102 through 110 being received), we know that only 1 packet remains in flight, namely 101.

After the isolated loss, we really want CWND to be 5, which means we want 5 packets in flight at any given time. By the time we get ack(101) from 107, we can deduce that only 4 packets remain in flight. (In reality, they are 101, 108, 109, 110, though the sender doesn't know that.)

At this point, we'd like to be able to send 111, for a total of 5 packets in flight. But the window won't let us do that, because the window is still stuck at 101 (first unacked byte) through 105 (CWND bytes later).

The key idea that will un-stall the sender is: Let's grant the sender temporary credit for each duplicate ack.

When a duplicate ack arrives, we can deduce that one fewer packet is in flight, though we don't know which one. To account for this, we will artificially extend the window by 1 packet, to allow the sender to send one more packet.

Fast Recovery: The Solution

Let's take this idea of artificially extending the window for each duplicate ack, and apply it to the example from before.

As before, the window starts at 101 through 110, and we send out 10 packets.

After sending 10 packets: CWND = 10.
101–110 can be in flight.



After ack(101) from 102: Can't send anything new.



After ack(101) from 103: Can't send anything new.



After ack(101) from 104: Can't send anything new.



As before, we get ack(101) from 102, the window stays unchanged, and we can't send anything new.

As before, we get ack(101) from 103, the window stays unchanged, and we can't send anything new.

3 duplicate acks detected. CWND = 5.
Extend window to account for the 3 acks: CWND = $5 + 3 = 8$.



After ack(101) from 105: Extend window to CWND = 9.
Can't send anything new.



As before, we get ack(101) from 104, the window stays unchanged, and we can't send anything new.

The third duplicate ack means we decrease CWND to 5, so the window is now 101 through 105.

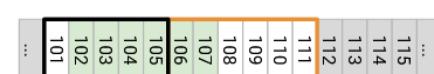
However, we got 3 acks, so we artificially extend the window by 3 to account for those acks. Thus, CWND is actually set to $5 + 3 = 8$.

Next, we get ack(101) from 105. This allows us to extend the window again, to 9. Now the window spans 101 through 109, so we still can't send new packets.

After ack(101) from 106: Extend window to C = 10.
Can't send anything new.



After ack(101) from 107: Extend window to C = 11.
We can send 111!



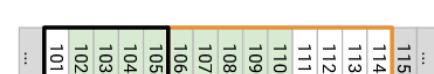
After ack(101) from 108: Extend window to C = 12.
We can send 112!



After ack(101) from 109: Extend window to C = 13.
We can send 113!



After ack(101) from 110: Extend window to C = 14.
We can send 114!



Next, we get ack(101) from 106. We extend the window again to 10, spanning 101 through 110, and we can't send anything new.

Next, we get ack(101) from 107. We extend the window again to 11, spanning 101 through 111. We can now send out 111!

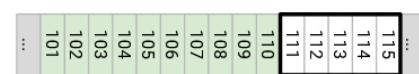
Next, we get ack(101) from 108. We extend the window again to 12, spanning 101 through 112. We can now send out 112!

Next, we get ack(101) from 109. We extend the window again to 13, spanning 101 through 113. We can now send out 113!

Next, we get ack(101) from 110. We extend the window again to 14, spanning 101 through 114. We can now send out 114!

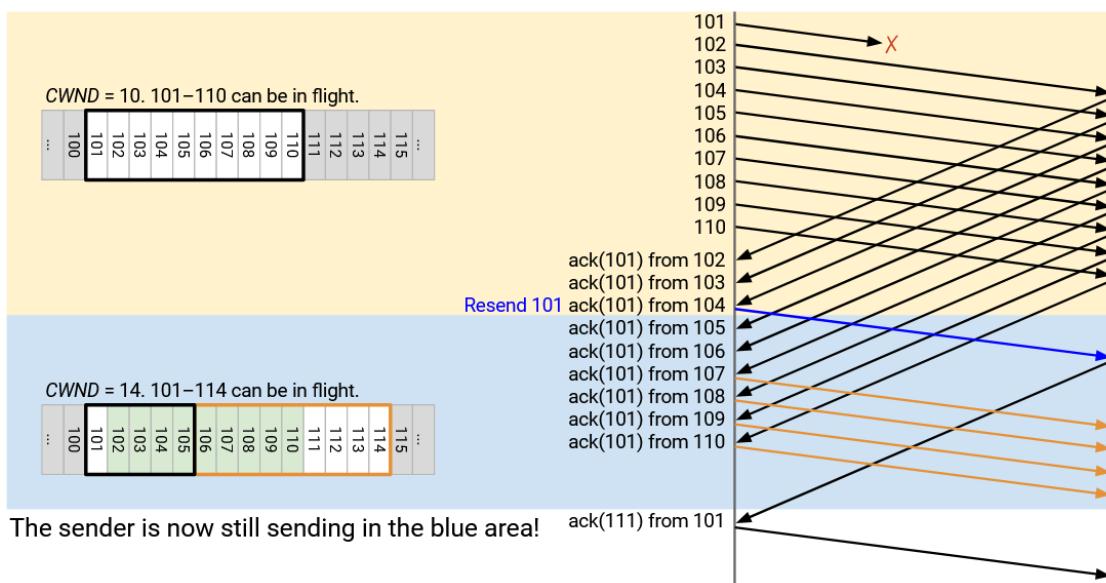
After ack(111) from resent 101: Back to normal. C = 5.

We can send 115.



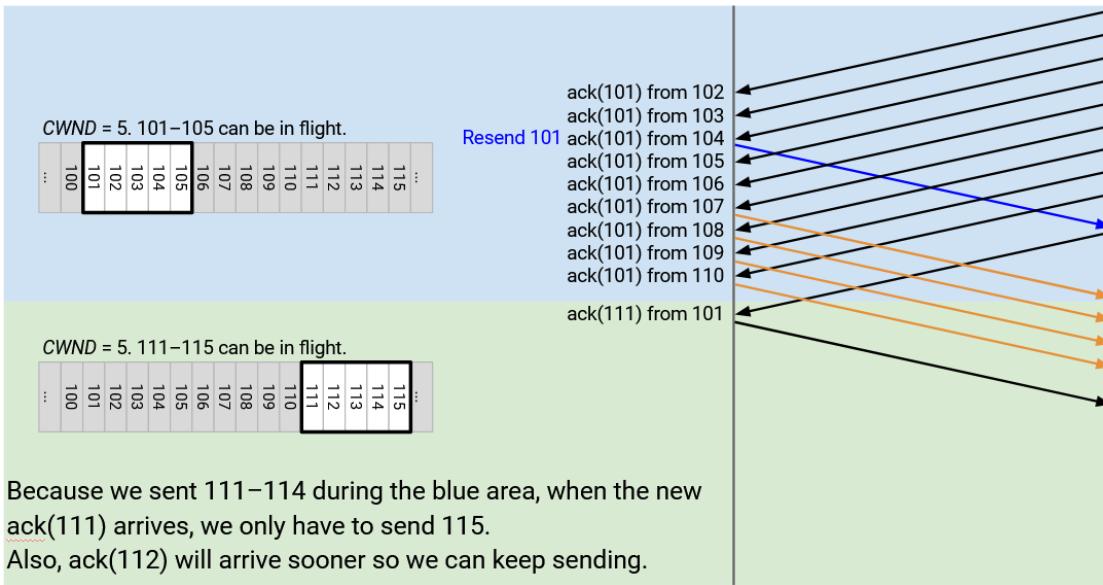
Eventually, we get ack(111) from the re-sent 101. At this point, we can reset CWND to its original intended value of 5, so that the window spans 111 through 115. This allows us to send out 115!

With this fix, we have solved our problem of the stalled sender. Originally, the sender had to wait for the re-sent 101 to be acked before sending new packets. Now, the sender is now able to keep sending packets before the re-sent 101 is acked.



We also solved the secondary problem from earlier, where the window leapt forward and we sent a burst of new packets (111 through 115). Now, 111 through 114 were sent out earlier, and when the window leapt forward, we only had to send out 115.

Without this fix, we had to stall for another round-trip while waiting for the burst of 111 through 115 to be acked. Now, because we stayed busy earlier and sent out 111 through 114, they'll get acked earlier, and we can keep sending 116 and beyond without that entire RTT of stalling.



Another way to look at this fix is to focus on the packets in the artificially-extended window.

When we get the third duplicate ack, the CWND shrinks to 5, but we artificially extend for the 3 duplicate acks for a CWND of 8. If you look at this extended window, 3 of the packets are acked (102, 103, 104, though we don't know it's these), and the other 5 are in-flight. This achieves our intended window of 5 packets in-flight.

Next, when we get another ack(101) from 105, the window extends to 9. Again, if you look in this window, 4 of the packets are acked (we don't know which), and the other 5 are in-flight, giving us our intended window of 5 in-flight packets.

When we get ack(101) from 106, the window extends to 10, including 5 received packets (from 5 duplicate acks), plus 5 in-flight packets (intended window size).

At every step, in our extended window, if you don't count the packets that have been acked, there are exactly 5 packets in-flight in the window. Again, we don't know exactly which packets in the window are acked, but we can use the duplicate acks to count how many packets were acked, and use that count to keep 5 in-flight packets.

When we get ack(101) from 107, the window extends to 11, including 6 received packets (from 6 duplicate acks). The other 5 packets in the window are allowed to be in-flight.

At this point, we sent 10 packets originally, and we got 6 duplicate acks, which tells us that only 4 packets remain in flight. This allows us to send out 111. The artificially-extending window captures this reasoning, because it extends the window to include 111.

When we get ack(101) from 108, we deduce that now, one fewer packet is in flight. So we artificially extend the window again to 12, which allows 112 to be sent.

Fast Recovery: Implementation

When we detect packet loss from duplicate acks, we temporarily enter the **fast recovery** mode, where additional duplicate acks artificially extend the window to prevent stalling.

Fast recovery mode is triggered when we receive 3 duplicate acks. Instead of just halving CWND, as we did before, we now set CWND to $CWND/2 + 3$, where the window is artificially extended by 3 for the 3 duplicate acks we got. We also set SSTHRESH to $CWND/2$, so that we remember the new safe rate for later.

While in fast recovery mode, every additional duplicate ack causes CWND to increase by 1, allowing the window to artificially extend.

Eventually, when we receive a new, non-duplicate ack, we leave fast recovery mode and set CWND to SSTHRESH. Note that while we were artificially extending the window, SSTHRESH always helped us remember the original halved rate that we want to ultimately send at.

TCP State Machine

We are finally ready to put all the pieces together and implement TCP, with congestion control.

The sender maintains 5 values:

The duplicate ack count helps us detect loss earlier than timeouts. It's initialized to 0.

The timer is used to detect loss. There's just a single timer.

RWND is used for flow control (don't overwhelm recipient buffer).

CWND is used for congestion control. It's initialized to 1 packet.

SSTHRESH helps the congestion control algorithm remember the latest safe rate. It's initialized to infinity.

The recipient maintains a buffer of out-of-order packets.

The sender responds to 3 events: Ack for new data (not previously acked), duplicate ack, and timeout.

The recipient responds to receiving a packet, by replying with an ack and a RWND value.

Let's see how the sender responds to each of the 3 events.

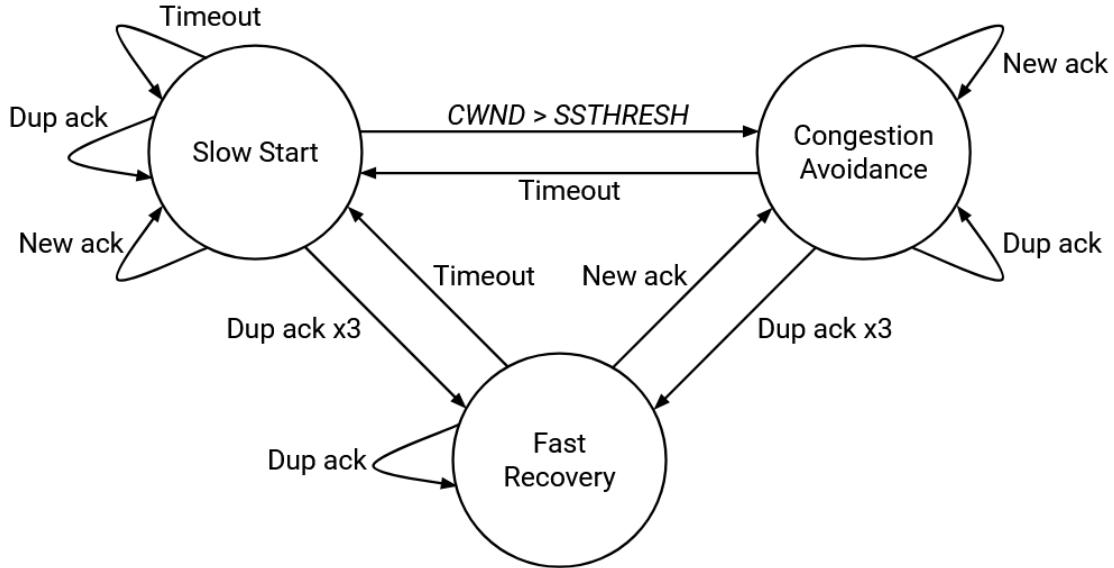
When we receive an ack for new data, not previously acked: If in slow-start mode, we increase CWND by 1. This allows the CWND to double each RTT. If we're in fast-recovery mode, we set CWND to SSTHRESH, so that we leave fast recovery (since we just got a new ack). If we're in congestion avoidance mode, we add $1/CWND$ to CWND, so that CWND increases by 1 per RTT (additive increase). We also reset the timer, reset the duplicate ack count, and, if the window allows, send new data.

When we receive a duplicate ack, we increment the duplicate ack count. If the count reaches 3, we re-send the left-most packet in the window. This is sometimes called fast retransmit. We also enter fast-recovery mode by setting SSTHRESH to $CWND/2$ (remember last safe rate) and set CWND to $CWND/2 + 3$ (adding 3 to artificially extend the window for duplicate acks). If the count exceeds 3, we stay in fast-recovery mode and artificially extend the CWND by 1 for every subsequent duplicate ack.

When the timer expires, we re-send the left-most packet in the window. We also go back to slow-start mode,

setting STHRESH to CWND/2 (remembering the last safe rate), and resetting CWND back to 1 packet.

The congestion control state machine shows the 3 possible modes that TCP can be in, and the conditions that trigger transitions between the modes.



We enter fast-recovery mode if we receive 3 duplicate acks. Once we're in this mode, any further duplicate acks keep us in fast-recovery mode (keep artificially extending window). To leave fast-recovery mode, either a timeout switches us back to slow-start mode, or a new ack lets us go back to congestion-avoidance mode.

A timeout triggers slow-start mode. Any further acks (duplicate or new) keep us in slow-start. Eventually, if CWND exceeds STHRESH (the safe rate), we enter congestion avoidance mode. Or, if we detect loss, we halve the rate and enter fast-recovery mode for a bit before going to congestion-avoidance mode.

In congestion avoidance mode, new acks keep us in this mode (additive increase), but duplicate acks send us to fast-recovery mode, and timeouts send us to slow-start mode.

TCP Congestion Control Variants

There are several variants of the TCP congestion control algorithm, all implemented in the end host's operating system. Fun fact: The names are related to the Berkeley Software Distribution (BSD) operating system.

In TCP Tahoe, if we get three duplicate acks, we reset CWND to 1, instead of halving CWND.

In TCP Reno, if we get three duplicate acks, we halve CWND. On timeout, we reset CWND to 1.

TCP New Reno is the same as Reno, but adds fast recovery. This is what we just implemented.

Other variants exist too. In TCP-SACK, we add selective acknowledgments where acks contain more detail (e.g. received all up to 13, and 18 too).

How can all these different variants co-exist? Why don't we need a single uniform protocol that everybody speaks? Remember, congestion control is implemented at the end hosts, so the sender can do whatever they want to adjust their rate. Ultimately, the network and the other end hosts just see TCP packets being sent at a (hopefully reasonable) rate, and they don't care how the rate is being computed. The underlying TCP packet format doesn't change with the different congestion control algorithms.

Not all protocols are compatible, though. If you use the TCP-SACK variant with selective acknowledgements, and I use TCP Tahoe, we have a problem. You expect selective acks, but I'm only providing cumulative acks.

TCP Throughput Model

Modeling Assumptions

In the previous sections, we developed an algorithm for congestion control. This algorithm told us how to adjust rate in response to congestion, but it didn't actually tell us what that rate is.

In this section, we'll develop a model for estimating the throughput of a TCP connection along a specific path. Specifically, we want a simple equation that gives us throughput as a function of a path's RTT and loss rate. This equation can allow operators and customers to estimate the rate of a TCP connection.

To simplify our model, we'll make a few assumptions: There is a single TCP connection. We'll ignore the slow-start phase. We'll assume the RTT is some fixed number.

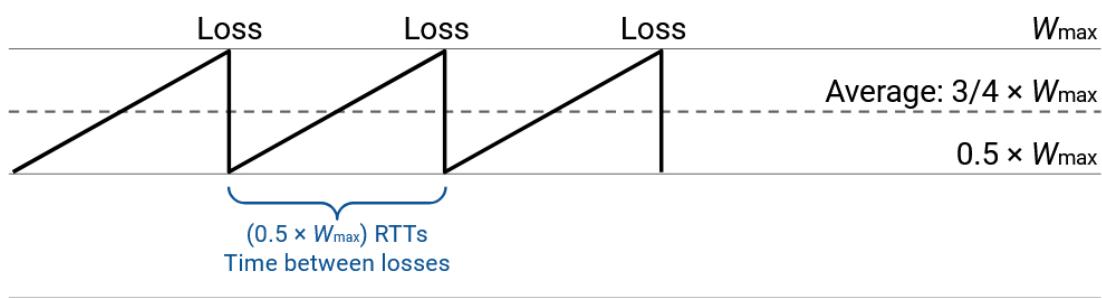
When the window size reaches the maximum bottleneck bandwidth W_{\max} (some constant), we'll assume we get exactly one packet loss. Since we only lose one packet, our loss will be detected by duplicate acks (no timeouts).

Throughput in Terms of Window Size

In this simplified model, we detect loss when the window size reaches W_{\max} , and our window size changes to $\frac{1}{2}W_{\max}$ as a result.

Then, for each subsequent RTT, our window size will increase by 1: $\frac{1}{2}W_{\max} + 1$, then $\frac{1}{2}W_{\max} + 2$, then $\frac{1}{2}W_{\max} + 3$, etc. Eventually, the window size will reach W_{\max} again and be halved, and this process will repeat.

Starting at $\frac{1}{2}W_{\max}$ and reaching W_{\max} takes $\frac{1}{2}W_{\max}$ RTTs (adding 1 per iteration, and each iteration is one RTT). This also tells us that there are $\frac{1}{2}W_{\max}$ RTTs between each loss.



Within each RTT, the average window size is $\frac{3}{4}W_{\max}$ (right in between $\frac{1}{2}W_{\max}$ and W_{\max}).

This window size is measured in packets (since we were adding 1 packet per iteration). Each packet can contain MSS bytes (maximum segment size), so the average window size in bytes is $\frac{3}{4}W_{\max} \times \text{MSS}$.

The window size tells us how much data we can send in each RTT. Thus, to compute the rate, we divide window size (data) by RTT (time) to get an average rate of $\frac{3}{4}W_{\max} \times \frac{\text{MSS}}{\text{RTT}}$.

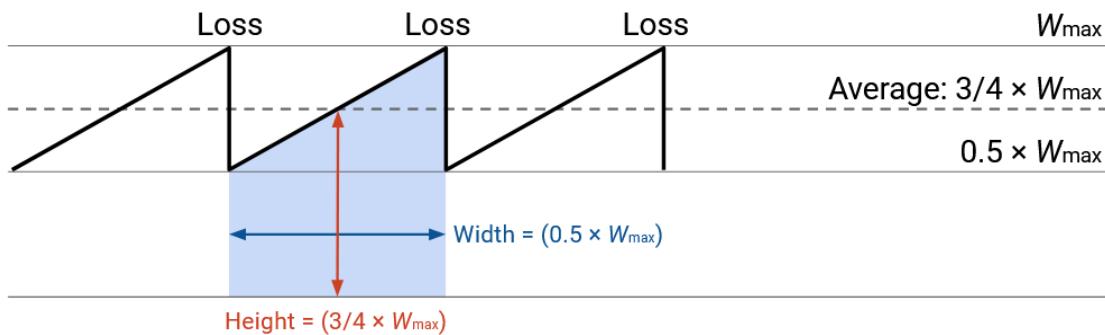
Throughput in Terms of Loss Rate

Our equation for throughput so far is: $\frac{3}{4}W_{\max} \times \frac{\text{MSS}}{\text{RTT}}$.

But our goal is to express throughput in terms of RTT and loss rate (denoted p). So, we now need to express W_{\max} in terms of the loss rate p .

From earlier, we deduced that a packet is lost once every $\frac{1}{2}W_{\max}$ RTTs. This was the time it took after a drop to climb back up to W_{\max} and encounter another drop.

So, to determine the loss rate, we just need to figure out how many packets are sent in $\frac{1}{2}W_{\max}$ RTTs.



Graphically, the number of packets sent is the area of this shape (rate times time), or equivalently, the area under the curve (the curve shows rate, and we want integral of rate).

We know from earlier that the average window size is $\frac{3}{4}W_{\max}$, so this is the number of packets sent per RTT. Therefore, across $\frac{1}{2}W_{\max}$ RTTs, we expect to send $(\frac{1}{2}W_{\max}) \times \frac{3}{4}W_{\max} = \frac{3}{8}W_{\max}^2$ packets.

Now that we know the number of packets sent between losses, we know that the loss rate is one lost packet, divided by the number of packets sent between losses. (For example, if we send 100 packets between losses, the loss rate is roughly 1/100).

Therefore, our loss rate is $p = 1/(\frac{3}{8}W_{\max}^2) = \frac{8}{3W_{\max}^2}$.

Now, we have a relation between W_{\max} and p , so we just need to do algebra to isolate W_{\max} in terms of p .

$$\begin{aligned} p &= \frac{8}{3W_{\max}^2} \\ 3W_{\max}^2 p &= 8 \\ W_{\max}^2 &= \frac{8}{3p} \\ W_{\max} &= \frac{2\sqrt{2}}{\sqrt{3p}} \end{aligned}$$

Now, we can do some more algebra to take our throughput equation from earlier and replace W_{\max} with p :

$$\begin{aligned}
 \text{throughput} &= \frac{3}{4} W_{\max} \times \frac{\text{MSS}}{\text{RTT}} \\
 &= \frac{3}{4} \left(\frac{2\sqrt{2}}{\sqrt{3p}} \right) \times \frac{\text{MSS}}{\text{RTT}} \\
 &= \sqrt{\frac{3}{2}} \times \frac{\text{MSS}}{\text{RTT}\sqrt{p}}
 \end{aligned}$$

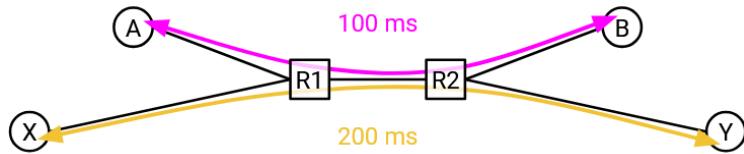
Implications of Equation

We now have an equation for throughput, expressed in terms of RTT and loss rate. What does it tell us?

Throughput is inversely proportional to the square root of the loss rate. Intuitively, if the loss rate is higher, then the throughput is lower. This makes sense, because losing more packets means that the window size gets halved more often.

Throughput is inversely proportional to RTT. Intuitively, if the RTT is lower, then the throughput is higher. This makes sense, because the window size increases every time we receive an ack, and a lower RTT means we get more acks more often.

This relationship between RTT and throughput can be a problem if we have multiple connections with different RTTs.



The connection with the lower RTT is going to be receiving acks more quickly, which means this connection also increases its window size and sends packets faster. In this case, it turns out the lower-RTT connection gets twice as much bandwidth as the higher-RTT connection.

Fundamentally, TCP is unfair when RTTs are heterogeneous (not the same). A shorter RTT improves propagation time, but it also helps TCP ramp up its rate faster. We accept this as a feature of TCP, and there's nothing we do about this in practice.

Rate-Based Congestion Control

Our congestion control protocol results in choppy throughput. As seen in the graph, the rate repeatedly swings between $W/2$ and W . Some applications don't like the constantly-changing rate, and would prefer to send data at a steady rate (e.g. streaming applications).

One possible solution for these applications is **equation-based** or **rate-based congestion control**, which abandons the rules for dynamically adjusting the rate, and instead simply follows the equation. To send data at a smooth rate, you can measure RTT and loss rate, plug them into the throughput equation, and

constantly send at the calculated rate. This solution also maintains fairness (doesn't hog bandwidth), because the equation ensures that we consume no more bandwidth than TCP would in a similar setting. (See RFC 5348 for more details.)

Formally, alternative implementations (including rate-based congestion control, and others) are considered **TCP-friendly** if they co-exist well with TCP by reducing their rate when necessary. TCP-friendly alternative algorithms lead to fair bandwidth sharing, even when some hosts run TCP and others run alternative algorithms.

Congestion Control Issues

Confusing Corruption and Congestion

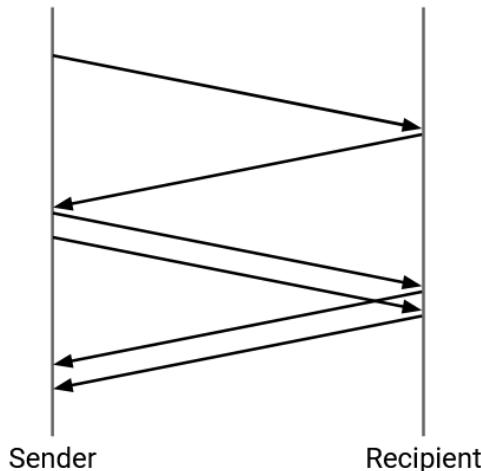
TCP detects congestion by checking for packet loss, but congestion isn't the only reason packets would be lost. Packets could also be lost from corruption, and TCP cannot distinguish between loss due to corruption or congestion. If a packet is corrupted, TCP will still drop its rate, even if the network isn't congested.

We can also see this in our equation, which related throughput to loss rate. The throughput and loss rate are inversely proportional, even for non-congestion losses. The equation can be helpful for estimating how a lossy link (e.g. a wireless link that frequently corrupts packets) would affect TCP.

Short Connections

Most TCP connections in real life are very short-lived. 50% of connections send fewer than 1.5 KB, and 80% of connections send less than 100 KB. Very few packets (maybe only one) are sent during these connections.

Suppose we had a connection where the sender only had 3 packets to send. What would TCP congestion control do? We'd start with window size 1 and send the first packet. Then, we'd wait for the ack, increase the window size to 2, and send the remaining two packets. Then, we'd wait for two more acks, and finish.



This connection took two RTTs to send 3 packets, resulting in an incredibly low throughput (1.5 packets per RTT).

More generally, these short connections never leave the slow-start phase, and never reach their fair share of bandwidth. This causes short connections to suffer from unnecessarily long transfer times.

Another problem with short connections is handling loss. Recall that we detect loss when there are 3 duplicate acks, but in a short connection, we might not have enough packets to trigger these duplicate acks. For example, if we had 4 packets to send, and we lost the second packet, we'd never get 3 duplicate acks. Instead, we'd have to wait for the timeout to trigger. At typical real-world timeout values of roughly 500ms, this can also cause short connections to take unnecessarily long.

How can we fix both of these problems? One partial fix is to start with a higher initial window (e.g. 10 packets instead of 1). Now, connections with 10 or fewer packets can just send all the data at the start of the connection.

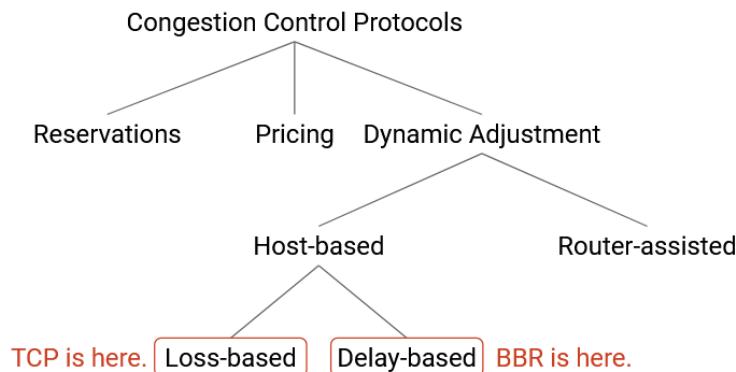
TCP Fills Up Queues

TCP detects congestion using loss, and the congestion control algorithm deliberately increases the rate until triggering loss. In order to trigger loss, queues need to fill up. This means that TCP introduces queuing delays throughout the network, and the delays affect everybody in the network.

Suppose we had one heavy-duty connection transferring a 10 GB file, and later, we start a small connection transferring a single packet. Both the connections share the same bottleneck link. The heavy-duty connection will increase its rate until the bottleneck link's queue fills up. Now, when the small connection starts, it is stuck waiting in the queue, behind the heavy-duty connection packets.

This problem is made worse if routers keep extremely large queues. Routers having excessive memory for long queues is called **bufferbloat**. An example of bufferbloat occurs in home routers, which might have a huge queue, but very few connections (only the ones in your home) using that queue. Now, any connections you make will cause large queuing delays for other connections.

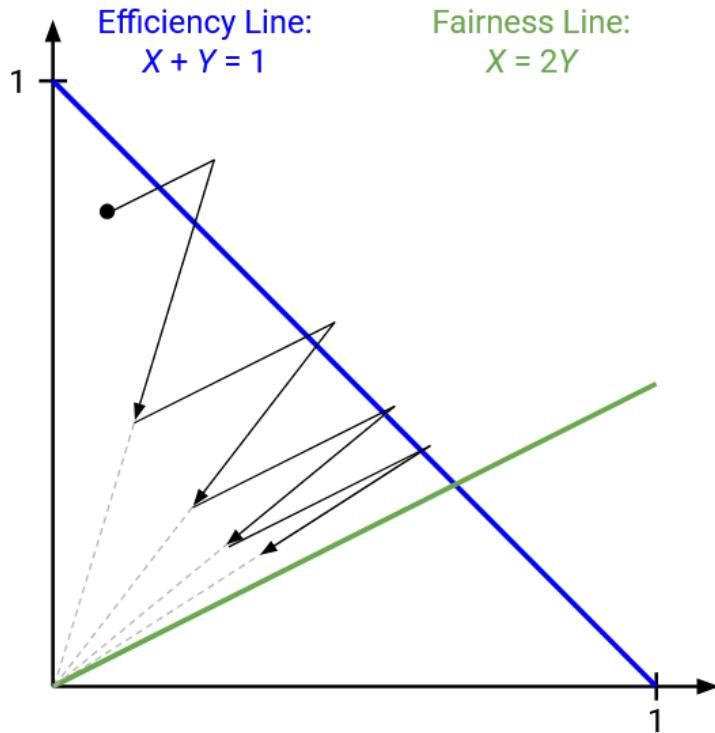
To avoid queues filling up, we could find a way to measure congestion that doesn't involve deliberately triggering losses. In particular, we could detect congestion when the RTT starts increasing, which indicates delay. This is the idea behind Google's recent BBR algorithm (2016). The sender learns its minimum RTT, and decreases its rate if it starts noticing the RTT exceeding the minimum.



Cheating

There is nothing enforcing that senders have to follow the TCP congestion control algorithm. Senders could cheat to get an unfairly large share of the bandwidth.

For example, a sender could increase the window faster (e.g. +2 every RTT, instead of +1). If we applied our graphical model to one cheating sender and one honest sender, AIMD updates would actually converge on a bad fairness line where the cheating sender gets twice the bandwidth of the honest sender.



Many other ways to modify the protocol also exist, such as starting with very large initial congestion window.

In practice, because TCP is implemented in the operating system, in order to cheat, the sender would have to modify the code in their operating system, which the vast majority of Internet users don't do.

If a small number of senders abuse the system, those senders will get more bandwidth. If a large number of senders abuse the system (e.g. Microsoft releases a version of Windows that abuses TCP), the millions of Windows users are still competing with each other, and it's unlikely that anybody will end up with more bandwidth.

Another way to cheat, without modifying TCP, is to open many connections. TCP only ensures that each connection gets a fair share. If a cheating sender opened 10 connections and an honest sender opened 1, the cheating sender would get 10 times more bandwidth. Many applications intentionally open more connections to improve bandwidth.

If cheating is possible, why hasn't the Internet suffered another congestion collapse? It turns out, researchers don't really know the answer either. One possibility is: cheaters who modify the congestion control algorithm might get an unfair share of bandwidth, but if they're still following the principles of congestion control (e.g. reducing rate when loss occurs), then they aren't overwhelming the network. By contrast, in the original 1980s congestion collapse, senders kept re-sending packets at high rates, with no notion of adjusting rate.

If cheating is possible, how much cheating occurs in practice? Again, we don't really know. It's hard to measure cheating (e.g. you don't know the windows being used at every sender).

Congestion Control and Reliability are Intertwined

The mechanisms for congestion control and reliability are tightly coupled. As we saw, congestion control was implemented by taking the code for TCP reliability and tweaking a few lines of code.

We can also see this dependence in the algorithm itself. The window is updated on acks and timeouts because the reliability code was written to respond to those events. We detect loss with duplicate acks because the reliability implementation uses cumulative acks.

Combining reliability and congestion control is a design choice. One benefit is that congestion control was a small code patch that could be widely deployed in response to the 1980s congestion collapse. However, since then, the combination of the two features has complicated evolution of our algorithms. For example, if we wanted to change something about our congestion control algorithm, we'd likely have to change the code for reliability as well. Or, if we wanted to change the reliability implementation (e.g. change from cumulative to full-information acks), we'd have to update congestion control as well.

From a design perspective, this is a failure of modularity, not layering. Congestion control and reliability are operating at the correct layer of abstraction (transport layer). However, within the transport layer, we haven't cleanly separated different functionality into different parts of our code.

Because congestion control relies on reliability, it's hard to achieve congestion control without reliability. Some applications (e.g. video streaming) might not want reliability, but still want congestion control. But there's no way to disable reliability and keep only congestion control.

Likewise, it's hard to achieve reliability without congestion control. For example, if we had a lightweight connection that sent one packet every 10 minutes, we probably don't need congestion control for this connection. But we can't easily disable congestion control for only some connections.

Router-Assisted Congestion Control

Congestion Control with Routers

Previously, we saw some issues with host-based congestion control algorithms. Many of these issues could be fixed with some help from routers!

TCP confuses congestion and corruption. TCP fills up queues, has choppy rates, and performs poorly on short flows, all because hosts need to constantly adjust their rates to detect congestion. If routers could tell the sender about congestion, or even directly tell the sender the ideal rate, then many of these problems could be solved.

Also, if routers enforce fair sharing, then it becomes much harder for hosts to cheat.

More philosophically, it's a natural design choice to have routers participate in congestion routing. Congestion happens at the routers, so they often have more information about congestion than the hosts.

Router-assisted congestion control can be very effective and result in near-optimal performance (high link utilization, low delays), but deploying these protocols can be challenging. Routers now need to support additional functionality, and sometimes that functionality can be quite complex. Some protocols might even require every router to agree to add that functionality.

Enforcing Fair Queuing

How can a router ensure that every connection gets its fair share?

So far, a router is receiving packets, queuing them if needed, and sending them out, in first-in-first-out (FIFO) order. The router doesn't care which connection a packet comes from.

In our new model, the router would need to classify packets into connections. (For now, assume the connections are all TCP connections.) This means the router has to look inside the packet to learn the source and destination IP addresses and ports.

To formally define fairness, the router could maintain a separate queue for each connection. When a packet arrives, the router adds the packet to the appropriate queue. Then, the router just needs to pick a queue each time, sending a packet from the front of that queue. As long as the router is picking queues in some fair way, then the router is enforcing fairness across connections.

If all packets are the same size, then the router could pick queues round-robin (send from the first queue, then the second queue, etc.). It turns out that this works, even if not all connections need the same bandwidth. Some connections might queue up packets more slowly than others. If we apply round-robin service to connections of different bandwidth, how do we compute the bandwidth allocated to each connection? For example, suppose we can send 10 packets per second, and A, B, and C sent 8, 6, and 2 packets per second, respectively.

A has 8 packets to send:

A1	A2	A3	A4	A5	A6	A7	A8
----	----	----	----	----	----	----	----

B has 6 packets to send:

B1	B2	B3	B4	B5	B6
----	----	----	----	----	----

C has 2 packets to send:

C1	C2
----	----

If we sent out packets round-robin, how many packets per second of each type would be sent? We can model this as a resource allocation problem and solve it.

Packets sent:

A1	B1	C1	A2	B2	C2	A3	B3	A4	B4
----	----	----	----	----	----	----	----	----	----

- We sent: 4x A packets, 4x B packets, and 2x C packets.

Packets not sent:

A5	A6	A7	A8	B5	B6
----	----	----	----	----	----

For example, suppose we have a link capacity of 10. Connection A requests 8, B requests 6, and C requests 2. How should we distribute the capacity among the three connections? If we tried to be fair, everybody would receive 3.33. But C only asked for 2, so let's give C the 2 it asked for, with no extra.

Now we have 8 left over, and A and B still need allocations. If we were fair, each would receive 4. This is less than what they asked for, but we have no way to satisfy their request, so we'll give each their fair share of 4.

Formally, to define max-min fairness, suppose C is the total bandwidth available to the router. Each connection r_i has a bandwidth demand, and we have to allocate a bandwidth a_i to each connection. The max-min bandwidth allocations are $a_i = \min(f, r_i)$, where f is the unique value (same value for all connections) such that $\sum a_i = C$. In this equation, the min term ensures that nobody gets more than they asked for, and the sum constraint ensures that no bandwidth is unused. Intuitively, f is the fair share that we equally allocate to everybody (hence one f value for all connections).

Another way to read this equation is: There is some magic fair-share number that we can distribute equally to everybody. If you requested less than the fair share, you get the fair share (no extra). If you requested more than the fair share, you are capped at the fair share, but nobody else gets more than you.

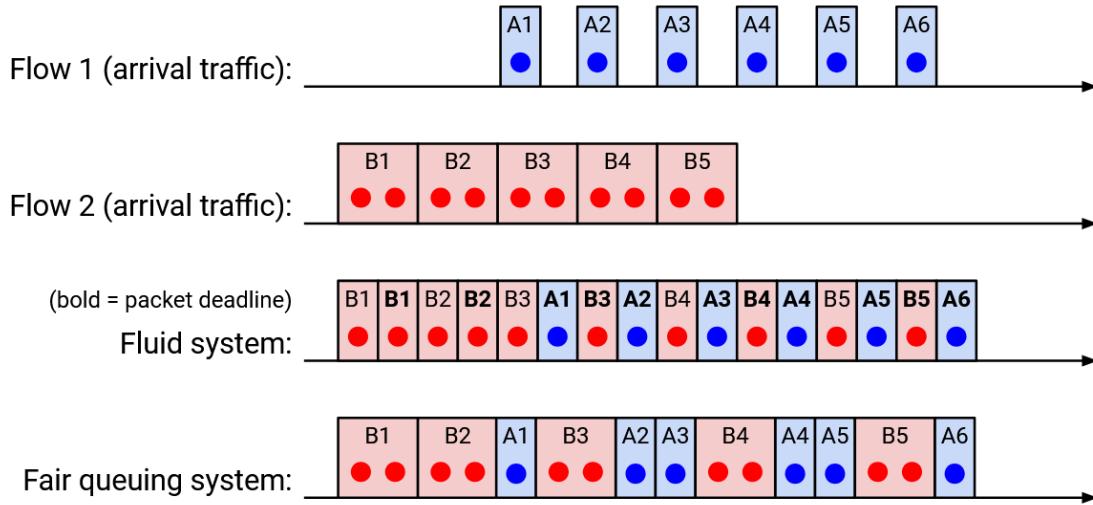
In the previous example, f was 4. A and B received f (they wanted more), and C received 2 (it wanted less).

If we apply max-min fairness, the equation guarantees that if you don't get your full demand, nobody else gets more than you. The round-robin approach is max-min fair (assuming equal packet size).

What if we don't assume equal packet size? In real life, packet sizes can vary widely (e.g. 40 bytes vs. 1500 bytes). Ideally, we'd like to perform bit-by-bit round robin, where we take turns sending one bit from each connection's queue. This isn't practical (we don't send one bit at a time), but if we applied this theoretically, we could write down the time when the last bit of a packet is sent out, for every packet. We'll call this the deadline for that packet. Then, a fair approximation would be sending out packets in order of deadline (when their last bit would have been sent ideally).

Fun fact: The paper about simulating fair queuing is extremely influential, and two of the co-authors are Scott Shenker (UC Berkeley faculty) and Srinivasan Keshav (EECS PhD student at the time).

Here's an example of exact bit-by-bit fair queuing on two connections (when a tie occurs, we pick the packet that arrives first).



Fair Queuing in Practice

What's good about fair queuing? It ensures isolation between connections, and prevents cheating connections from getting more bandwidth. Connections wouldn't need to implement TCP (or a TCP-friendly alternative), and can pick their own (possibly unfriendly) congestion control algorithm.

Fundamentally, the benefit of fair queuing is its resilience to external factors like cheating and RTT variations. No matter what, everyone gets a fair share of a given link. But, we still need end hosts to discover and adapt to their fair share (e.g. slow down if they're requesting too much).

What's bad about fair queuing? It's much more complicated than FIFO queuing. The process of computing deadlines is tricky and we have not shown an algorithm for doing so here. Also, routers would need to maintain multiple queues, and do extra parsing work on every packet.

In practice, we can't implement perfect fair queuing in routers (too complicated to run at high speeds), but approximations do exist (e.g. Deficit Round Robin). Modern routers typically implement approximations, though with fewer queues. Fewer queues means that instead of one queue per connection, isolation is more coarse-grained (e.g. one queue per customer).

Fair queuing cannot eliminate congestion. It is only an alternative way to manage congestion. For example, consider this bottleneck link: It might allocate 0.5 Gbps to each connection, which defeats cheating. But, if the top connection runs at 0.5 Gbps, then 0.4 Gbps will get dropped at the immediate next link. A better allocation would be to send 0.1 Gbps along the top connection, and 0.9 Gbps along the bottom connection.

Fundamentally, the problem is that this bottleneck link doesn't know what will happen at future (downstream) links. The only way to fix this is to make the sender host slow down (router queuing cannot help).

Fair queuing gives us per-connection fairness, but philosophically, we still have to ask if this is the right model of fairness. As we saw earlier, per-connection fairness means that someone with more connections still gets more bandwidth. Should we instead enforce fairness per source-destination pair, or perhaps per source? Should we penalize connections that use more congested links (hogging more scarce resources)?

Router-Assisted Congestion Control

Fair queuing enforces fairness over a specific link, but it doesn't tell the sender anything. What if the routers passed information back to the sender to help the sender adjust its rate?

One solution is to have routers directly tell senders the rate they should use. We could add a rate field in packets, and have routers fill in that field with the connection's fair share. When the packet arrives at the sender, the sender can read the header and set the rate to what the routers said. Now, the sender doesn't need to dynamically adjust to discover a good rate.

Another solution is to have routers notify senders about congestion (without specifying an exact rate). This is deployed in the form of an **Explicit Congestion Notification (ECN)** bit in the IP header. If a packet passes through a congested router, the router sets that bit to 1. When the recipient gets a packet with the ECN bit on, the ack reply will also have the ECN bit set, so the sender learns about congestion.

There are many options for when routers set this bit. The router could be paranoid and set the bit frequently, which reduces delay but may lead to underused links. Or, the router could be more reckless and rarely set the bit, which increases delay but results in high link utilization.

There are also many options for how the host reacts when this bit is set. For example, the host could pretend the packet was dropped and adjust accordingly.

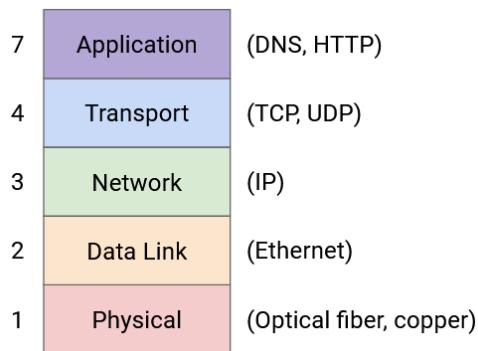
What's good about ECN? It solves the problem of confusing corruption and congestion. It allows routers to warn hosts about congestion earlier (e.g. before the queue is full), which can reduce delays. It's also lightweight to implement.

In practice, effective ECN requires most or all routers to support this protocol and turn the bit on when necessary. In the modern Internet, the ECN bit is deployed on some, but not all routers. However, the ECN bit can be effective in a small network (e.g. inside a datacenter's local network) where all routers agree to enable the bit.

DNS

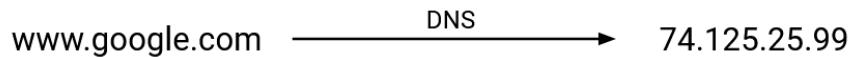
What is DNS For?

In this section, we'll be looking at a few common applications (Layer 7) that operate on top of the layers we've built so far.



The first application we'll look at is DNS, which is a protocol that operates on top of the layers (1-4) to provide the important network functionality of name resolution.

The Internet is commonly indexed in two different ways. Humans refer to websites using human-readable names such as `google.com` and `eecs.berkeley.edu`, while computers refer to websites using IP addresses such as `172.217.4.174` and `23.195.69.108`. DNS, or the **Domain Name System**, is the protocol that translates between the two.



Brief History of DNS

To understand why DNS was designed the way it was, it helps to go back and look at the history of its development.

On the original Internet and its predecessor (ARPANET), there were three key applications. This is before the World Wide Web and web browsers, back when most applications ran on the command line.

Remote terminal (`telnet`) allowed users to connect to another machine remotely, and run commands on that remote computer. You might have heard of `SSH`, which is a modern, secure version of this protocol.

File transfer allowed users to transfer files between their local computer and a remote computer. You might have heard of `FTP`, which is the application-level protocol for file transfers.

Email allowed users to exchange messages to each other. Modern email comes with a web client in your browser, but originally, users had to type a command in the terminal like `mail alice@46.0.1.2`, where `46.0.1.2` is the IP address of the recipient, and `alice` is the recipient user on that machine.

In all three cases, performing an operation requires specifying a remote host. But, as we mentioned, remembering the IP addresses of remote hosts is hard and not user-friendly.

The first attempt to fix this problem was to assign a **hostname** to every IP address. Every computer would have a file called `hosts.txt` mapping every hostname to its IP address. For example, we can map hostname `ucb-arpa` to `10.0.0.78`, and rewrite `mail mosher@10.0.0.78` as `mail mosher@ucb-arpa`.

This concept actually still exists today. If you've ever started a server on your own computer, it probably has the IP address `127.0.0.1` and the hostname `localhost`. Typing either one in your browser gives you the same result.

We have to ensure that `hosts.txt` is the same across different computers. If you went to a different computer and typed `mail mosher@ucb-arpa`, you should probably be sending mail to the same person. The hosts file was originally maintained by a single person (Elizabeth Feinler), and passed between users by physically copying a paper document.

The original paper hosts file was human-readable. It mapped hostnames to addresses, but also included information like the user's full name, the protocols they ran (e.g. TCP, FTP), and even their phone number.

HOST NAMES			
HOSTNAME	HOST ADDR (Dec)	LIAISON	STATUS
AFWL-TIP	176	D Hyde (505)247-1711 x3803	TIP, Up 3-74
ALOHA-TIP	164	R Binder (808)948-7066	TIP
AMES-11	208	J Hart (415)965-5935	USER, up 12-73
AMES-67	16	W Hathaway (115)965-6033	SERVER
AMES-TIP	144	W Hathaway (115)965-6033	TIP
ANL	?	L Amiot (312)739-7711 x4309	SERVER, up 2-74
ARPA-DMS	28	S Crocker (202)694-5037	USER, Agency use only
ARPA-TIP	156	S Crocker (202)694-5037	TIP

Everybody agreed (as mentioned in RFC606 in 1973) that this was an absurd situation. If you get a paper copy of the file, you'd have to manually input it into your computer. Also, the file was passed around on paper, so you might have an outdated file.

The first improvement was to make the list machine-readable. Now, instead of paper copies, we could at least use protocols like FTP to share the file. But this still isn't scalable. We can't ask a single person to maintain this file forever. Also, as the file grew big, downloading the file could get very slow, and if the network connection broke, you might end up with a partial file.

DNS was first proposed in 1983 (RFC882) as a solution to these problems. There have been some modifications since, but the fundamental system is still in use to this day.

Fun fact: The first software written for running DNS servers on Unix was BIND (1984, UC Berkeley), and it's still pretty common today.

DNS Design Goals

This brief history informs some of the design goals of DNS that we should keep in mind.

DNS must be scalable. The Internet has a huge number of hosts, and a huge number of lookups are performed every second. Hosts can also be added and removed frequently.

DNS must be highly-available, lightweight, and fast. Almost every Internet connection starts with a DNS lookup to translate host name to IP address. Therefore, DNS should be extremely fast, or else every connection would be slowed down. Also, there shouldn't be a single point of failure, or else the Internet would stop working on a failure.

Name servers

It would be great if there was single centralized server that stored a mapping from every domain to every IP address that everyone could query, but unfortunately, there is no server big enough to store the IP address of every domain on the Internet and fast enough to handle the volume of DNS requests generated by the entire world. Instead, DNS uses a collection of many **name servers**, which are servers dedicated to replying to DNS requests.

Each name server is responsible for a specific zone of domains, so that no single server needs to store every domain on the Internet. For example, a name server responsible for the `.com` zone only needs to answer queries for domains that end in `.com`. This name server doesn't need to store any DNS information related to `wikipedia.org`. Likewise, a name server responsible for the `berkeley.edu` zone doesn't need to store any DNS information related to `stanford.edu`.

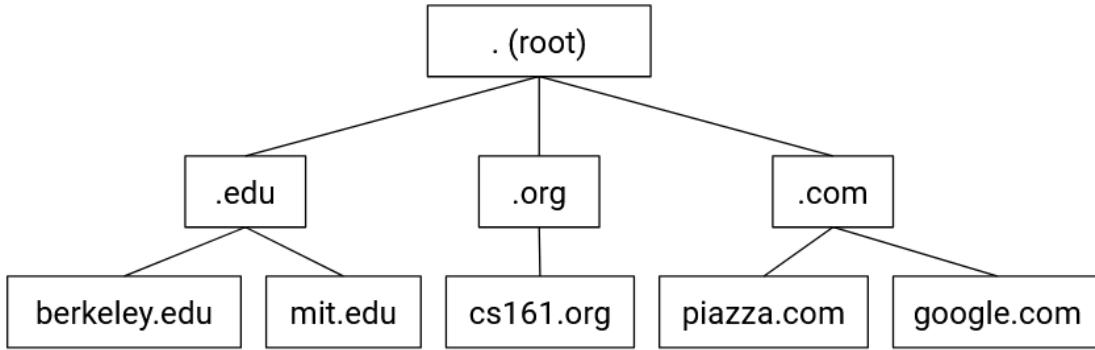
Even though it has a special purpose (responding to DNS requests), a name server is just like any other server you can contact on the Internet—each one has a human-readable domain name (e.g. `a.edu-servers.net`) and a computer-readable IP address (e.g. `192.5.6.30`). Be careful not to confuse the domain name with the zone. For example, this name server has `.net` in its domain, but it responds to DNS requests for `.edu` domains.

Name server hierarchy

You might notice two problems with this design. First, the `.com` zone may be smaller than the entire Internet, but it is still impractical for one name server to store all domains ending in `.com`. Second, if there are many name servers, how does your computer know which one to contact?

DNS solves both of these problems by introducing a new idea: when you query a name server, instead of always returning the IP address of the domain you queried, the name server can also direct you to another name server for the answer. This allows name servers with large zones such as `.edu` to redirect your query to other name servers with smaller zones such as `berkeley.edu`. Now, the name server for the `.edu` zone doesn't need to store any information about `eecs.berkeley.edu`, `math.berkeley.edu`, etc. Instead, the `.edu` name server stores information about the `berkeley.edu` name server and redirects requests for `eecs.berkeley.edu`, `math.berkeley.edu`, etc. to a `berkeley.edu` name server.

DNS arranges all the name servers in a tree hierarchy based on their zones:



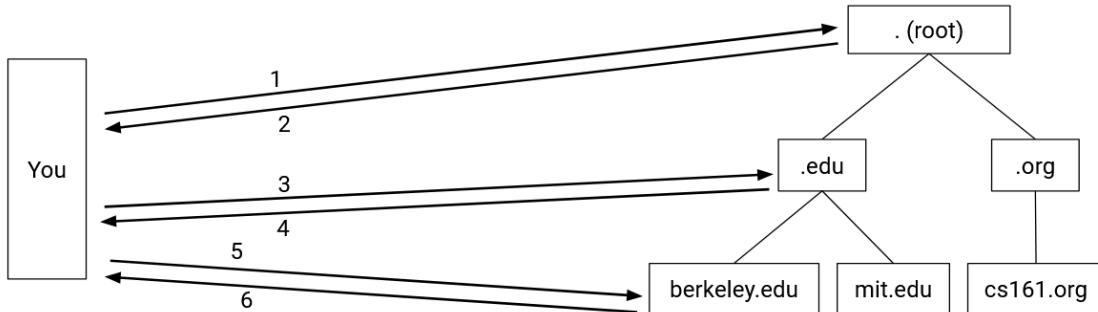
The **root server** at the top level of the tree has all domains in its zone (this zone is usually written as `.`). Name servers at lower levels of the tree have smaller, more specific zones.

DNS Lookup (Conceptual)

DNS queries always start at the root. The root will direct your query to one of its children name servers. Then you make a query to the child name server, and that name server redirects you to one of its children. The process repeats until you make a query to a name server that knows the answer, which will return the IP address corresponding to your domain.

To redirect you to a child name server, the parent name server must provide the child's zone, human-readable domain name, and IP address, so that you can contact that child name server for more information.

As an example, a DNS query for `eecs.berkeley.edu` might have the following steps. (A comic version of this query is available at <https://howdns.works/>.)



1. You to the root name server: Please tell me the IP address of `eecs.berkeley.edu`.
2. Root server to you: I don't know, but I can redirect you to another name server with more information. This name server is responsible for the `.edu` zone. It has human-readable domain name `a.edu-servers.net` and IP address `192.5.6.30`.
3. You to the `.edu` name server: Please tell me the IP address of `eecs.berkeley.edu`.
4. The `.edu` name server to you: I don't know, but I can redirect you to another name server with more information. This name server is responsible for the `berkeley.edu` zone. It has human-readable domain name `adns1.berkeley.edu` and IP address `128.32.136.3`.

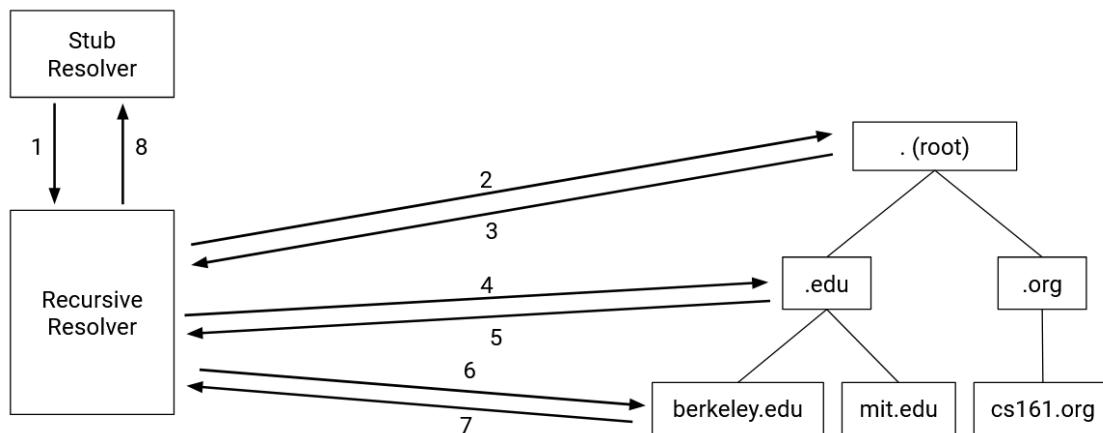
5. You to the berkeley.edu name server: Please tell me the IP address of eecs.berkeley.edu.
6. The berkeley.edu name server to you: OK, the IP address of eecs.berkeley.edu is 23.185.0.1.

Once we get the answer, we can store it in our cache so that we don't have to ask again if we need this record again later.

Stub Resolvers and Recursive Resolvers

Originally, the end host (e.g. your computer) would perform the DNS lookups, contacting each name server.

Today, your local computer usually delegates the task of DNS lookups to a **DNS Recursive Resolver**, which queries the name servers for you. When performing a lookup, the **DNS Stub Resolver** on your computer sends a query to the recursive resolver, lets the resolver do all the work, and receives the response back from the resolver.



How do we learn the IP address of the resolver? When you first connect to the Internet, somebody can tell you the resolver address. You can also manually enter the address of a resolver you want to use.

Some well-known resolvers on the Internet include 1.1.1.1 (run by Cloudflare) and 8.8.8.8 (run by Google). They often have memorable IP addresses so we don't have to refer to them by name. Otherwise, we'd have to do a DNS lookup to find their IP address, and the whole point of these servers is to do DNS lookups for us.

In addition to tech companies, ISPs also operate resolvers, as part of the Internet service they sell to customers. (It would be pretty bad if you paid for Internet service, but had to type IP addresses in your web browser.)

The router in your home can also act as a resolver. DNS queries can be faster if you use a router that's physically close to you (less delay between you and the router).

One major benefit of a resolver is better caching. The resolver processes queries from lots of different end hosts (not just your own computer), so it builds a much larger cache. If you ask the resolver about **eecs.berkeley.edu**, and somebody before you asked the resolver that same question recently, the resolver can give you the answer without contacting any additional name servers.

Note that even though recursive resolvers store larger caches, the stub resolver can still maintain its own separate cache. Some queries can get answered by the stub resolver's cache, without even asking the recursive resolver.

Redundancy

So far, we've been talking about "the berkeley.edu name server," but in reality, each zone has multiple name servers. All the name servers for a zone are functionally identical, and each name server can answer any query in the zone.

This ensures that we have availability for that zone, and if one name server goes down, the others can keep answering queries for that zone. By convention, zones are required to have two name servers, though in practice, most zones will have at least three.

Usually, one of the name servers is designated as the primary server that actually manages the zone. The rest of the servers are secondary mirror servers that just store and serve a copy of the information on the primary server.

Now, in the DNS lookup, name servers might reply to you with: "I don't know, but you should ask the .edu zone. This zone has 13 name servers. Here are the domains and IP addresses for each of them." Then, you can pick which of the name servers to contact next.

DNS APIs

Now that we have a conceptual picture of DNS, let's see how it's actually implemented.

For starters, how do programmers use DNS to perform lookups?

There are relatively simple, common, and stable APIs to perform DNS lookups. The API is fairly similar across different languages.

In the standard C library, `gethostbyname("foo.com")` can be used to look up the IP address corresponding to `foo.com`. This function is limited to IPv4, though, so it's now deprecated (though you'll still see it used).

The updated version, also in the standard C library, is `getaddrinfo("example.com", NULL, NULL, &result)`, which looks up the IP address of `example.com` and stores the answer in the `result` struct. Don't worry too much about the specifics or the extra arguments (set to null here). This replacement API supports more than IPv4 (e.g. IPv6).

As a programmer, you don't need to worry about DNS complexities like recursive resolvers or specific name servers. You can just call a standard library function. These functions usually make requests to the stub resolver in your operating system.

DNS uses UDP, Not TCP

Fundamentally, DNS is a client-server protocol. One person (the client) sends a question, and the other person (the server) sends the answer. The client is usually a user host or recursive resolver, and the server is usually a name server. How should the client and server format their messages?

DNS uses UDP (best-effort packets, no TCP handshakes) to make DNS lightweight and fast. We don't have to wait for a 3-way TCP handshake to complete. We don't care about packets arriving in order, because queries and responses often fit into a single UDP packet.

With UDP, servers don't need to keep state per connection (contrast with TCP, where the server has to maintain a buffer). Every packet can be processed independently. This also helps keep DNS lightweight, since name servers receive lots of requests, and opening a new connection for each one would be expensive.

How do we deal with UDP being unreliable and packets being dropped? We can solve this with a simple retry mechanism. If you don't get a reply within some time limit, send the query again. The timeout varies between operating systems, though it can be fairly slow.

UDP being unreliable and timeouts being slow is why it's important to have a resolver that's nearby and can be contacted reliably. For example, a resolver in your home router is close to you, and probably pretty reliable (not too much congestion in your home network). You can also improve reliability by having multiple backup resolvers (e.g. home router and 8.8.8.8).

As we mentioned, DNS queries and responses usually fit into a single packet. One notable exception is when we transfer a zone between a primary name server and secondary name servers. The secondary name server has to say: Give me all your records, so that I can help you serve them. The response is probably going to be very big, so these transfers are often done over TCP.

Recent advances in DNS have added security features (e.g. stop an attacker from changing the records in flight), which might also require switching from UDP to TCP.

Recall that UDP implements ports to support multiple applications on a single server. By convention, all name servers listen for DNS queries on UDP port 53. This port number is well-known and constant so that everybody can contact the correct port on name servers.

DNS Message Format

16 bits	16 bits
Identification	Flags
# Questions	# Answer RRs
# Authority RRs	# Additional RRs
Questions (variable # of RRs)	
Answers (variable # of RRs)	
Authority (variable # of RRs)	
Additional info (variable # of RRs)	

The first field is a 16 bit **identification field** that is randomly selected per query and used to match requests to responses. When a DNS query is sent, the ID field is filled with random bits. Since UDP is stateless, the DNS response must send back the same bits in the ID field so that the original query sender knows which DNS query the response corresponds to.

The next 16 bits are reserved for flags. The QR bit specifies whether the message is a query (bit is 0) or a response (bit is 1). The RD bit indicates whether you want the resolver to perform a recursive lookup, or just return whatever the name server says (even if it's "I don't know").

Theoretically, you can specify a query type in the flags, though the IQUERY type is obsolete, and the STATUS type is not really defined, so the QUERY type is used for basically everything.

The flags can also specify whether the query was successful (e.g. the NOERROR flag is set in the reply if the query succeeded, the NXDOMAIN flag is set in the reply if the query asked about a non-existent name).

The next field specifies the number of questions asked (in practice, this is always 1). The three fields after that are used in response messages and specify the number of **resource records** (RRs) contained in the message. We'll describe each of these categories of RRs in depth later.

The rest of the message contains the actual content of the DNS query/response. This content is always structured as a set of RRs, where each RR is a key-value pair with an associated type.

For completeness, a DNS record key is formally defined as a 3-tuple <Name, Class, Type>, where Name is the actual key data, Class is always IN for Internet (except for special queries used to get information about DNS itself), and Type specifies the record type. A DNS record value contains <TTL, Value>, where TTL is the time-to-live (how long, in seconds, the record can be cached), and Value is the actual value data.

There are three main types of records in DNS. **A type records** map domains to IPv4 addresses. The key is a domain, and the value is an IP address. Similarly, **AAAA type records** map domains to IPv6 addresses. **NS type records** map zones to domains. The key is a zone, and the value is a domain.

You might sometimes encounter a CNAME type record, which is used for aliasing or redirecting. The name and value are both domains, and the record indicates that both domains have the same IP address.

Important takeaways from this section: Each DNS packet has a 16-bit random ID field, some metadata, and a set of resource records. Each record falls into one of four categories (question, answer, authority, additional), and each record contains a type, a key, and a value. There are A type records and NS type records.

DNS Lookup (Implementation)

Now, let's walk through a real DNS query for the IP address of eecs.berkeley.edu. You can try this at home with the dig utility—remember to set the +norecurse flag so you can unravel the recursion yourself.

Every DNS query begins with the root server. The names and IP addresses of the root servers are usually hardcoded into resolvers, in the form of a root hints file. Here's a root hints file if you're curious: <https://www.internic.net/domain/named.root>.

The first root server has domain a.root-servers.net and IP address 198.41.0.4. We can use dig to send a DNS request to this address, asking for the IP address of eecs.berkeley.edu.

```
$ dig +norecurse eecs.berkeley.edu @198.41.0.4

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 26114
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 13, ADDITIONAL: 27

;; QUESTION SECTION:
;eecs.berkeley.edu.      IN      A
```

```

;; AUTHORITY SECTION:
edu.          172800  IN  NS  a.edu-servers.net.
edu.          172800  IN  NS  b.edu-servers.net.
edu.          172800  IN  NS  c.edu-servers.net.
...
;; ADDITIONAL SECTION:
a.edu-servers.net. 172800  IN  A   192.5.6.30
b.edu-servers.net. 172800  IN  A   192.33.14.30
c.edu-servers.net. 172800  IN  A   192.26.92.30
...

```

In the first section of the answer, we can see the header information, including the ID field (26114), the return flags (NOERROR), and the number of records returned in each section.

The **question section** contains 1 record (you can verify by seeing QUERY: 1 in the header). It has key `eecs.berkeley.edu`, type A, and a blank value. This represents the domain we queried for (the value is blank because we don't know the corresponding IP address).

The **answer section** is blank (ANSWER: 0 in the header), because the root server didn't provide a direct answer to our query.

The **authority section** contains 13 records. The first one has key `.edu`, type NS, and value `a.edu-servers.net`. This is the root server giving us the zone and the domain name of the next name servers we could contact. Each record in this section corresponds to a potential name server we could ask next.

Notes: Usually, the server with the earliest name (alphabetically or numerically) is the primary server (here, `a.edu-servers.net`), and the rest are mirrors. Also, note that it's okay that there are multiple records with the same name (`.edu`). This just tells us there are multiple name servers that can all answer queries for this zone.

The **additional section** contains 27 records. The first one has key `a.edu-servers.net`, type A, and value `192.5.6.30`. This is the root server giving us the IP address of the next name server by mapping a domain from the authority section to an IP address.

Together, the authority section and additional section combined give us the zone, domain name, and IP address of the next name server. This information is spread across two sections to maintain the key-value structure of the DNS message.

For completeness: 172800 is the TTL (time-to-live) for each record, set at 172,800 seconds = 48 hours here. The IN is the Internet class and can basically be ignored. Sometimes you will see records of type AAAA, which correspond to IPv6 addresses (the usual A type records correspond to IPv4 addresses).

Sanity check: What name server do we query next? How do we know where that name server is located? What do we query that name server for?¹

```
$ dig +norecurse eecs.berkeley.edu @192.5.6.30
```

¹Query `a.edu-servers.net`, whose location we know because of the records in the additional section. Query for the IP address of `eecs.berkeley.edu` just like before.

```

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 36257
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 3, ADDITIONAL: 5

;; QUESTION SECTION:
;eeecs.berkeley.edu.      IN   A

;; AUTHORITY SECTION:
berkeley.edu.        172800  IN   NS    adns1.berkeley.edu.
berkeley.edu.        172800  IN   NS    adns2.berkeley.edu.
berkeley.edu.        172800  IN   NS    adns3.berkeley.edu.

;; ADDITIONAL SECTION:
adns1.berkeley.edu. 172800  IN   A     128.32.136.3
adns2.berkeley.edu. 172800  IN   A     128.32.136.14
adns3.berkeley.edu. 172800  IN   A     192.107.102.142
...

```

The next query also has an empty answer section, with NS records in the authority section and A records in the additional section which give us the domains and IP addresses of name servers responsible for the `berkeley.edu` zone.

```

$ dig +norecurse eecs.berkeley.edu @128.32.136.3

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 52788
;; flags: qr aa; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; QUESTION SECTION:
;eeecs.berkeley.edu.      IN   A

;; ANSWER SECTION:
eeecs.berkeley.edu.  86400  IN   A     23.185.0.1

```

Finally, the last query gives us the IP address corresponding to `eeecs.berkeley.edu` in the form of a single A type record in the answer section.

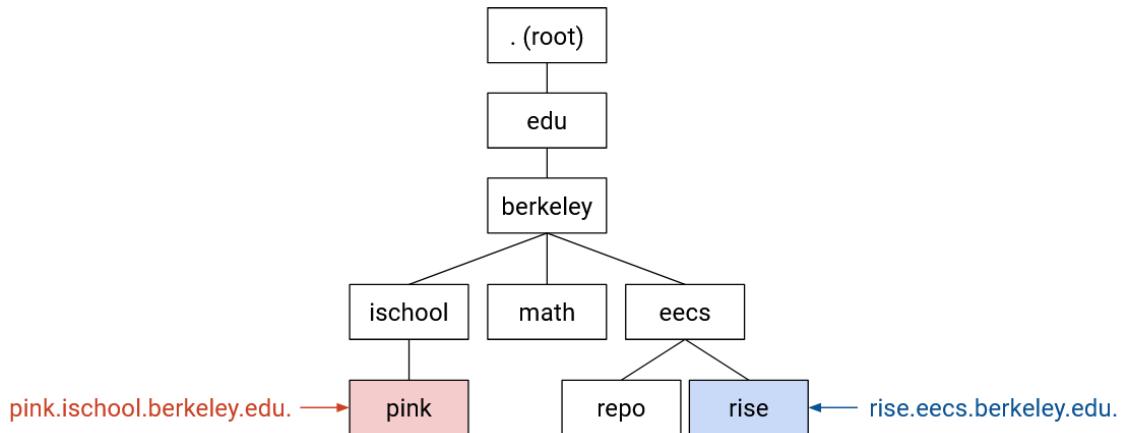
In practice, because the recursive resolver caches as many answers as possible, most queries can skip the first few steps and used cached records instead of asking root servers and high-level name servers like `.edu` every time. Caching helps speed up DNS, because fewer packets need to be sent across the network to translate a domain name to an IP address. Caching also helps reduce request load on the highest-level name servers.

Now that we know how DNS is implemented to support queries, we can look at how DNS actually works in practice, and explore the real-world business side of DNS.

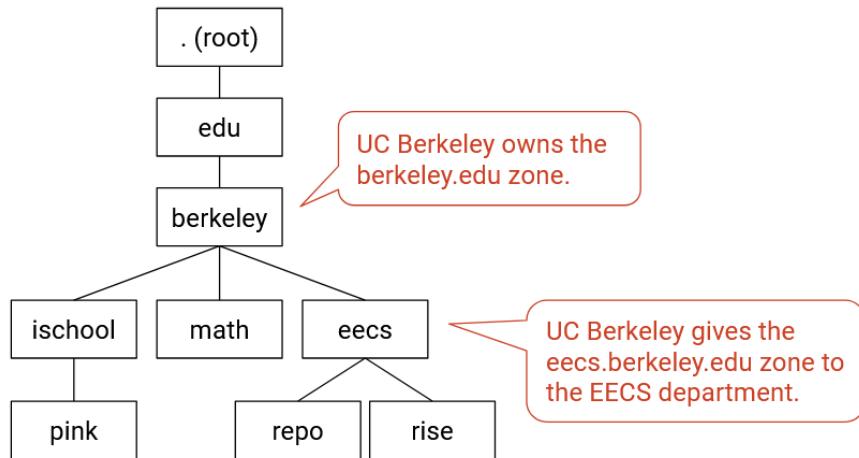
DNS Authority Hierarchy

The tree hierarchy that we've drawn actually represents three different ways in which DNS is hierarchical.

As we've already seen, DNS names are hierarchical. This is why our domain names, like `eecs.berkeley.edu`, are multiple words separated by dots.



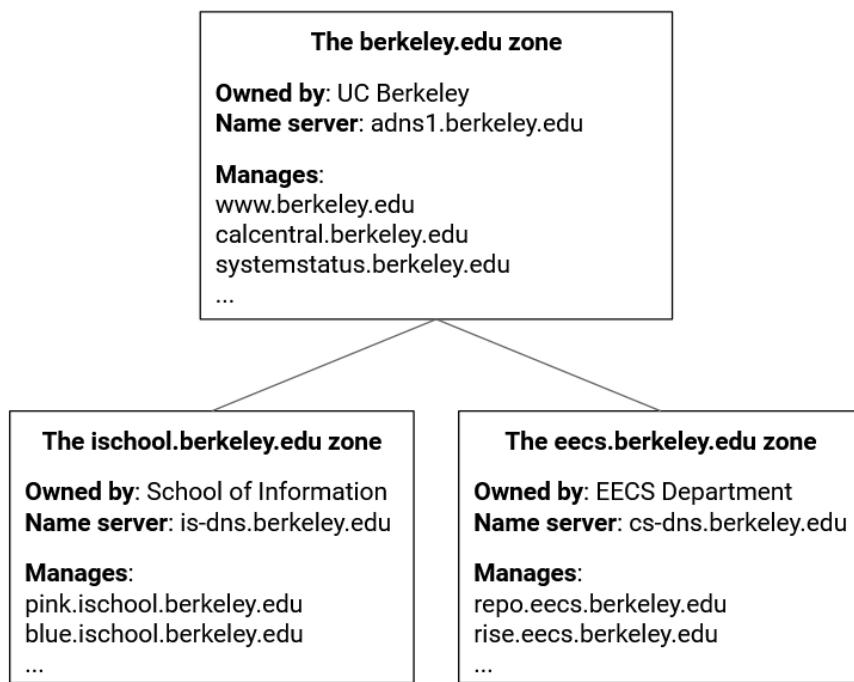
We've also seen that the infrastructure of DNS is hierarchical. We can organize name servers into the tree hierarchy, where each name server only knows about its own part of the tree.



The third hierarchy that we'll introduce is authority. This tells us who defines the names that exist in the tree. For example, the organization that operates the `.edu` name server is responsible for all the domains in the `.edu` zone. The `.edu` organization can then delegate authority for parts of its zone to subordinates in the tree.

For example, the `.edu` organization can say: `berkeley.edu` (and all subdomains) is in my zone, and I will transfer control of this part of my zone to the UC Berkeley organization. Now, we've created a new zone owned by UC Berkeley, and the `.edu` organization doesn't need to be aware of updates in this new

`berkeley.edu` zone. UC Berkeley has the authority to create new domains in its zone, or perhaps delegate some parts of its zone to further sub-organizations.



When we draw this tree with all three hierarchies in mind, we can be more precise and say that each node represents a zone. A zone is formally defined as an administrative authority responsible for some part of the hierarchy.

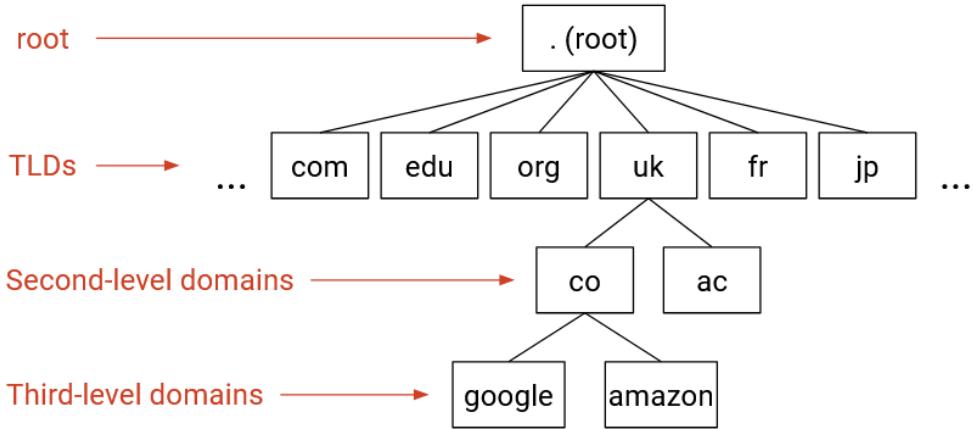
From the naming perspective, each zone contains one or more name-IP mappings, where the names are subdomains of that zone. The `eecs.berkeley.edu` zone can contain the name `eecs.berkeley.edu` or the name `iris.eecs.berkeley.edu`, but cannot contain the name `math.berkeley.edu`.

From the infrastructure perspective, that zone is supported by one or more name servers answering queries about the domains in that zone.

From the authority perspective, the zone is controlled by some organization, who was given authority by its parent to manage the names in that zone. For example, UC Berkeley, who controls the `berkeley.edu` zone, can delegate the `eecs.berkeley.edu` zone to the EECS department.

DNS Zones in Practice

Who are these organizations in real life?



The root zone is controlled by ICANN (Internet Corporation for Assigned Names and Numbers). They are responsible for delegating parts of the root zone (which represents the entire Internet) to specific top-level domains.

All of the zones directly beneath the root in the tree are **top-level domains (TLDs)**. The Internet was originally developed in the US, so the earliest TLDs split the Internet into zones based on purpose, such as .com (commercial), .gov (government), .edu (education), and .mil (military). Eventually, TLDs for countries were created, such as .fr (France) and .jp (Japan).

Historically, there were relatively few TLDs. More recently, ICANN started selling new TLDs for a registration price of \$150,000 (plus ongoing maintenance costs), which led to an explosion of new TLDs. As of 2024, there are over 1,500 TLDs, including company-specific TLDs like .google, and more exotic TLDs like .travel or .pizza.

Each TLD is run by some organization, and that organization can decide its own structure for how to further divide up that TLD. For example, the .uk TLD is managed by Nominet, and they decided to split up the .uk zone by purpose, creating zones such as .co.uk (commercial), .ac.uk (academia).

Zones further down in the tree can be referred to by their depth. example.com is a second-level domain, and blog.example.com is a third-level domain. These zones are usually controlled by various organizations and companies, who buy those zones from the parent zone. When you buy a zone, you also need to tell the parent zone about the name servers you're using. This allows the parent zone to redirect users to your name servers.

The name servers further down the tree are usually often run by domain name registrars. Registrars are companies that own specific zones, and allow users to host their services on specific domains in that zone. For example, I might pay a monthly fee to host my website on blog.foo.com. The registrar will usually offer to add the corresponding domain-to-IP mapping to its name servers.

In addition to registrars, companies like Google might also run their own name servers for their own applications (e.g. provide the record for maps.google.com). Amazon Web Services also has a name server called Route 53, where users can add records to be served.

Root Server Availability with Anycast

It would be really bad if the root servers were unavailable. Someone with an empty cache would be unable to make any DNS requests. Eventually, everyone's caches would empty out as TTLs expire, and the Internet would stop working.

For redundancy, there are 13 root servers located around the world. We can look up the IP addresses of the root servers, which are public and well-known.

13 still seems like kind of a low number, considering the entire Internet relies on them. To provide extremely high availability, there are actually way more than 13 root servers, but the list only contains 13 IP addresses, because of a clever trick called anycast.

In the **anycast** trick, we deploy many mirrors of a root server, and use the same IP address for all of them. If you contact the domain `k.root-servers.net` or its corresponding IP address, you could be talking to any one of its mirrors.

Each of the 13 root servers is actually made up of a bunch of mirrors. For example, `f.root-servers.net` has over 3,000 instances. The mirrors could be run by different network operators (e.g. Google and Cloudflare might help run root mirrors).

To implement anycast, during the routing protocol, every mirror announces the same IP address. The rest of the routing protocol still works the same. You might hear many announcements about routes to `k.root-servers.net`, and you can accept any of them, and you'll end up routing packets to one of the mirrors. If one of the mirrors goes down, the rest of the mirrors are still sending announcements, and you can accept a different route to maintain availability.

Anycast also means that root IPs very rarely change, even as mirrors are added and removed. As a result, the root hints file contains records for the root name servers with very long TTLs (42 days).



Here's a map of all the mirrors for `k.root-servers.net`. All of them advertise the same IP address, and your router probably chooses to talk to the closest mirror. `k.root-servers.net` is operated by RIPE (in Europe), which might explain why there are so many mirrors in Europe.

Public resolvers like Google's 8.8.8.8 can also be anycast for high availability.

DNS for Email

DNS can be used to store and serve more than domain-to-IP mappings. For example, if you want to send email to an address like `evanbot@berkeley.edu`, your computer still needs to know where to send packets.

To translate email addresses to mail servers, we can use MX type records. These records map domains like `berkeley.edu` to mail servers like `aspmx.l.google.com`. Note that it's okay if the mail server is in a different zone (e.g. here, the mail servers for UC Berkeley are operated by Google).

Historically, an email address corresponded to a user on a specific machine, so the mail server would be that machine. Nowadays, you probably want to be able to access your email from anywhere. That's why the MX records map to mail servers like `aspmx.l.google.com`, which receive your mail and let you access your mail from any computer.

One key difference in MX records is that the values also contain a priority. If you receive multiple MX records, all mapping the same domain to different mail servers, you should try the mail server with the highest priority (lowest number) first.

DNS for Load Balancing

We saw in the demo that the final name server returned a single A type record, but it's actually possible to receive multiple A type records, mapping a single domain to multiple IP addresses. There's no order to these records, and the server might shuffle the order before returning them. Any of them are valid, and the client can pick any of them (usually the first one). Providing multiple IP addresses is useful for load-balancing and redundancy.

It's also possible for the name server to send back different A type records, depending on who sent the query, and where they sent the query from. This can be useful if we want to balance the load for a popular domain like `www.google.com` across multiple servers. For example, we can try to send the user to the nearest server.

The name server now needs additional (possibly proprietary) logic to decide which record(s) to send in the reply. The name server could check who the recursive resolver is. It could also check who the end client is, though this requires an extension to DNS so that the resolver's query includes the client address. It could even check the geographic location of the end user, though this requires some mapping of IP address to physical location. Commercial databases like MaxMind exist for mapping IP addresses to physical locations.

Load balancing based on the user's geographical location isn't perfect. Even if we knew our servers' physical locations and the user's physical location, we have to do some guesswork about which server is closest from a network perspective. We also don't know about the performance (e.g. network bandwidth) between the user and the different servers.

From California:

```
$ dig google.com +short  
142.251.46.238
```

From Oregon:

```
$ dig google.com +short  
74.125.135.113
```

Here's an experiment to see how well Google's geographic load balancing performs. We looked up

`www.google.com` in San Francisco and Oregon and got two different IP addresses.

We then did a reverse lookup to match each IP address to a specific server name (different for each server, not `www.google.com`). The result is a PTR type record, which maps IP address to name (reverse of A type record). We can see that the IP address that Google gave us in San Francisco is mapped to some server with the name `sfo03s25`. Assuming `sfo` stands for San Francisco, that's pretty good!

If we look at the latency, connecting from the San Francisco computer to the IP address served to San Francisco takes 20 ms, and connecting from the San Francisco computer to the IP address served to Oregon takes 35 seconds. Google did a good job giving the San Francisco computer a closer server!

HTTP

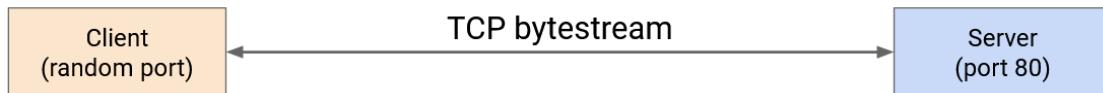
Brief History of HTTP

In 1989, Tim Berners-Lee was working at CERN (Switzerland research lab) and needed to exchange information between scientists. At the time, protocols like FTP existed for transferring files over the Internet. However, a file will often have links to other resources on the Internet. His goal was to create a protocol and file format that would allow linking pages to each other and fetching those pages.

The original HTTP specification was given version number HTTP/0.9 and released in 1991. HTTP/1.0 was standardized in 1996, and HTTP/1.1 was standardized in 1997. Unless otherwise specified, this section is referring to HTTP/1.1, since this is the most common version in use today. More recent versions do exist (see end of this section), but the fundamentals of the protocol have stayed the same for over 20 years.

HTTP Basics

HTTP runs over TCP. Two people who want to send data over HTTP will first start a TCP connection. Then, they can use its bytestream abstraction to reliably exchange arbitrary-length data. Hosts running HTTP don't have to worry about packets being reordered, dropped, and so on.



HTTP is a **client-server** protocol. We designate one person as the client (e.g. you, the end user), and one person as the server (e.g. Google, a bank website, etc.). The client is almost always running HTTP in a web browser (e.g. Firefox or Chrome), though HTTP can also be run in other ways (e.g. directly on the terminal).

When forming an HTTP connection, the server must listen for connection requests on the well-known, constant port number 80. (HTTPS, a more recent secure version, uses port 443). The client can choose any random ephemeral port number to start the connection, and the server can send replies to that port number.

HTTP is a **request-response** protocol. For each request that the client sends, the server sends exactly one corresponding response.

HTTP Requests

The HTTP request message is formatted in human-readable plaintext, which means you can type raw HTTP requests into the terminal. The request contains three parts: method, URL, version, and optional content.

The message ends with a newline (technically, CRLF, look it up if curious), which you can think of as the user pressing the Enter key after typing in the HTTP request in their terminal.

The version number specifies what version of HTTP you're using, e.g. HTTP/0.9, HTTP/1.0, HTTP/1.1, etc.

The requested URL identifies a resource on the server. You can think of the URL as the filepath of what you're trying to retrieve from the remote server. For example, in the URL `http://cs168.io/assets/lectures/lecture1.pdf`, we're trying to retrieve a file in the assets/lectures folder, named lecture1.pdf, on the cs168.io remote server. (Servers aren't required to work this way, but it's a useful intuition to have.)

The method identifies what action the user wants to perform. Initially, HTTP only had one method, GET, which allows a client to retrieve a specific page (indicated by the URL) from the server.

Later, HTTP was extended to add other methods. Notably, the POST method was added, which allows the client to supply information to the server as well. For example, if the user fills out a form and clicks Submit, that data is sent to the server in a POST request.

Some less-used methods exist as well. HEAD retrieves the headers (metadata) of the response, but not the actual content of the response. Other methods like PUT, CONNECT, DELETE, OPTIONS, PATCH, and TRACE extend HTTP into a protocol that lets the user interact with content on the server. The user can now make changes to the content, as opposed to the original design, where the user could only retrieve content. These extra methods make HTTP very flexible for all sorts of different application.

Note that with other methods like POST, we still have to provide a URL to indicate how to interpret the data we're sending. On a bank website, sending a name to the /send-money URL will probably do something different from sending that same name to the /request-money URL.

For GET requests, the content of the request is usually empty, since we're asking for a page from the server and not sending any of our own information. By contrast, for a POST request, the content of the request contains the data we want to send to the server.

HTTP Responses

Each HTTP request corresponds to one HTTP response. The response is also in human-readable plaintext, which means you can read raw HTTP responses in the terminal. The response contains four parts: version, status code, optional message, and content.

As before, the version specifies the version of HTTP being used.

The content is where the server would put, for example, the page that the user requested in a GET request.

The status code is a number that allows the server to indicate the result of the client's request. Each status code has a corresponding human-readable message.

The status codes are classified into various categories according to numeric values:

100 = Informational responses.

200 = Successful responses. 200 OK indicates a successful request, where the definition of success depends on the method of the request and the application using HTTP (remember, status codes are in every response, regardless of what method, GET/POST/etc., the request was). 201 Created indicates that the request succeeded and some new resource was created. This is usually seen in POST or PUT requests.

300 = Redirection messages. These allow the server to tell the client that they should go look for the resource (specified by the URL) somewhere else. Two common ones are 301 Moved Permanently and 302

Found (a weird name for moved temporarily). Sometimes, the status code itself doesn't provide enough context (as seen with these redirects). Therefore, the response will also contain additional information about where the resource has moved (e.g. another URL).

The use of more specific status codes allows the client to determine its future behavior based on the code. For example, 301 Moved Permanently tells the client to stop looking in the original location, while 302 Found (moved temporarily) might tell the client to come back and check again later.

400 = An error attributable to client action. 401 Unauthorized says that the client is not allowed to access this content, but if they authenticate their identity (e.g. log in), then they might be able to access the content. 403 Forbidden says that the client is authenticated, and the server knows their identity, but they're still not allowed to access the content.

Again, using more specific codes lets the client determine future behavior. 401 Unauthorized might cause the client browser to show a login window, while 403 Forbidden might cause the client browser to show an error message (since the user has already logged in).

500 = An error attributable to server action. 500 Internal Server Error and 503 Service Unavailable are common. There's not much the client can do about these errors, except maybe try again later.

Some error codes like 404 (File Not Found) and 503 (Service Unavailable) are very recognizable.

Sometimes, the appropriate status code to use can be ambiguous. For example, if we sent an HTTP request to Google using version 0.9, the appropriate request might be 505 (HTTP Version Not Supported). Instead, Google responds with 400 (Bad Request). Usually, the goal is to provide an error from the correct category (e.g. 400 and 500 indicate errors) that elicits the correct behavior from the client.

HTTP Headers

If the client has additional information they'd like to send to the server, they can include additional metadata called **headers**. In HTTP/1.1, no headers are mandatory, so it's legal to not include any (though the server/client might expect a header and error).

For example, the Location header can be used in HTTP 300 responses to indicate where the resource has moved.

Sometimes, the header information is optional. For example, the User-Agent header in the request lets the client tell the server about the client browser or program (e.g. Firefox or Chrome). This could allow the request to be processed differently depending on the header field (e.g. whether the user is on Chrome or the terminal).

Other times, header information is more critical. For example, Content-Type tells us whether the payload is an HTML page, image, video, etc. This tells the browser how to display the HTTP response. If a server is hosting multiple websites, the Host header can be used in requests to specify what website to request.

Some headers are relevant in requests. These allow the client to pass information to the server. For example, the Accept header lets the client tell the server which content type the client is expecting (e.g. HTML for human-readable pages, JSON for machine-parsable data). The User-Agent header indicates the type of client being used, and the Host header indicates the specific host being accessed (in case a server is hosting multiple websites). The Referer header indicates how the client made the request (e.g. if they clicked on a link from Facebook to make this request).

Other headers are relevant in responses. Remember, headers are metadata about the content, not the content itself. For example, Content-Encoding tells us how the bits of the response should be interpreted (e.g. Unicode/ASCII for human-readable text, or gzip for a compressed file). The Date header tells us when the server generated the response.

Some headers are representation headers, which are used in both requests and responses to describe how the content is represented. For example, the Content-Type header specifies the type of the document (e.g. text, image) and can be in POST requests, or GET responses. Representation headers let us carry different types of content over HTTP, which allows the protocol to be generalized and usable by all sorts of applications.

HTTP Examples

In your terminal, you can type `telnet google.com 80` to connect to Port 80 (HTTP) on Google's server. The terminal will then allow you to type a raw HTTP request, with headers, like:

```
GET / HTTP/1.1
```

```
User-Agent: robjs
```

This is a GET request for the root page on the server, running on HTTP version 1.1. The User-Agent header indicates the type of client we're using.

Likewise, the response is also human-readable.

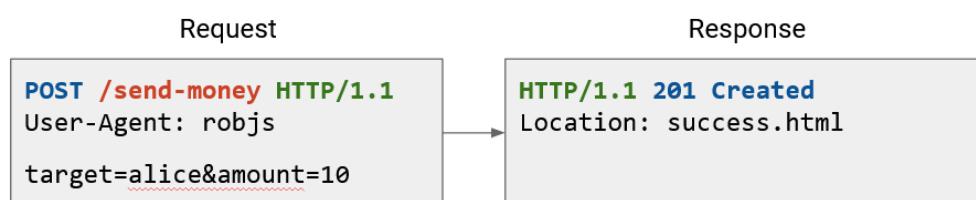
```
HTTP/1.1 200 OK
```

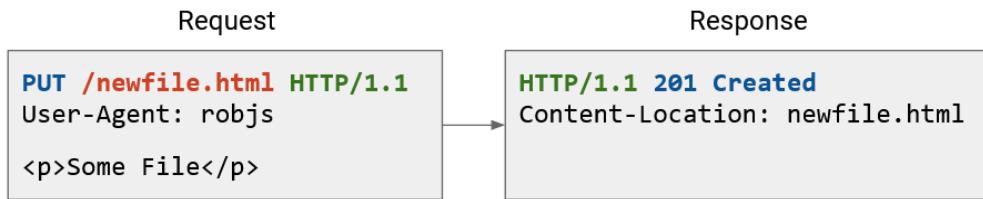
```
Date: Sat, 16 Mar 2024 18:33:08 GMT
```

```
Content-Type: text/html; charset=ISO-8859-1
```

```
<!doctype html><html lang="en"><head><meta content="Search the world's information, including webpages, images, videos and more. Google has many special features to help you find exactly what you're looking for." name="description">...
```

The `HTTP/1.1 200 OK` tells us the version, and the status code (200) with its corresponding message (OK). There are two headers attached, the date of the response, and the content type. Then, the content contains the raw HTML of the webpage. If we opened this HTML in a web browser, it would look like an actual webpage.





Here are some other examples. Notice that the content section is blank in the GET request, but contains data in the POST and PUT request. Conversely, the POST and PUT responses have no contents, but the GET response does.

The status code and header tells us useful metadata about the request. For example, status 201 Created tells us that the file we send was successfully stored on the server. The header tells us where on the server the file was stored (and we might use that location to retrieve the file later).

Speeding Up HTTP with Pipelining

Loading a single page in your web browser can require several HTTP requests. When you make a request for a YouTube video, your browser has to make separate requests for the video itself, the HTML with the other text on the webpage (e.g. video title, comments), thumbnails of related videos, and so on. Many of these requests probably go to the same server (e.g. YouTube's server in this case).

Recall that HTTP runs over TCP. In the naive case, every separate request would require starting a new TCP connection with a 3-way handshake. After the request, we close the connection and then immediately re-do a handshake for the next request.



HTTP 1.1 optimized this by allowing multiple HTTP requests and responses to be pipelined over the same connection. Now, we no longer need a separate TCP connection (with a separate handshake) for every request.

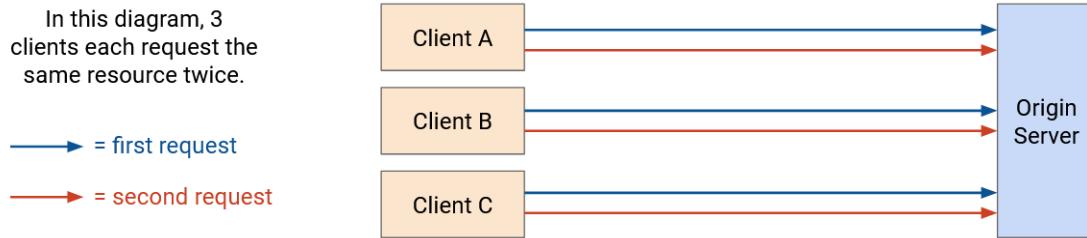


One downside to this optimization is, the server now has to keep more simultaneous open connections. The server needs to have some way to time out connections. If the server gets overloaded with open connections, the client might get an error like 503 Service Unavailable. Attackers could exploit this in a denial-of-service attack.

Speeding Up HTTP with Caching: Types

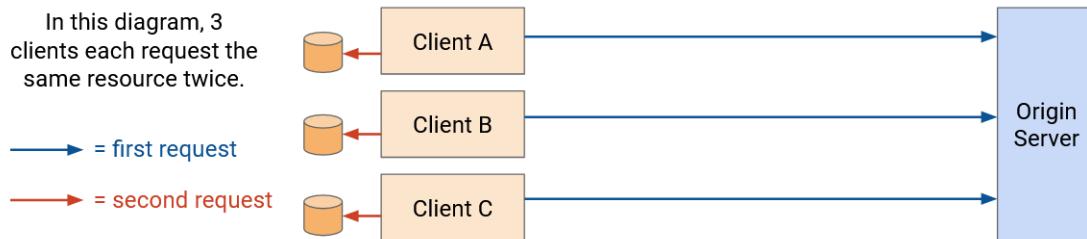
Another strategy for speeding up HTTP is caching responses to avoid making duplicate requests for the same data.

If we don't cache, every request must reach the server.

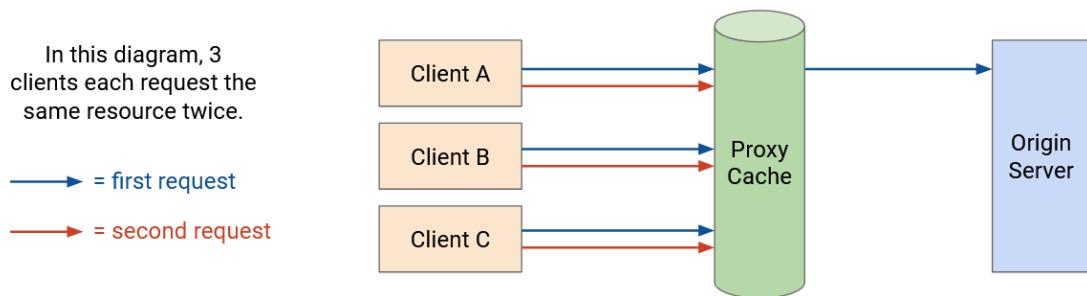


There are three types of HTTP caches:

Private caches are associated with a specific end client connecting to the server (e.g. the cache in your own browser). Now, if the same user requests the same resource a second time, they can fetch the resource from their local cache. However, private caches are not shared between users.



Proxy caches are in the network (not on the end host), and are controlled by the network operator, not the application provider. These caches can be shared between lots of users, so a user requesting a resource for the first time might get the data from the proxy cache instead of the origin server.



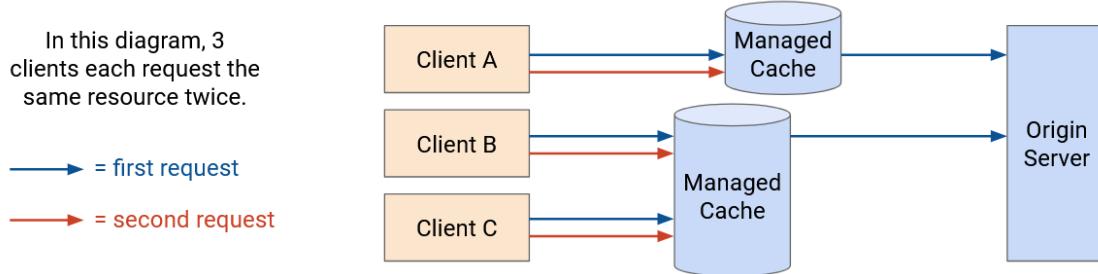
One problem with proxy caches is, the clients need some way to be redirected to the proxy cache. The application isn't running the proxy cache, so the origin server doesn't necessarily know about the proxy

cache. The network operator needs some way to control the end client to inform them about the proxy cache.

One common approach is lying in DNS responses, which is possible if the network operator controls both the proxy cache and the recursive resolver. When the client makes a request to the origin server, it has to look up the origin server's IP address. The recursive resolver can lie and say, "The IP address of the origin server is, 1.2.3.4 (proxy cache's IP address)." Now, requests to the origin server go to the proxy cache instead, who can serve cached responses. Or, if the requested resource isn't in the proxy cache, the proxy cache can make a request to the origin server, and then the cache can serve the request back to the user.

Another problem with proxy caches is, the application isn't managing the proxy cache. The origin server has to trust that the proxy cache is doing the right thing (e.g. respecting cache expiry dates, serving the correct data).

Managed caches are in the network, and are controlled by the application provider. Note that managed cache servers are deployed separately, and are not the original server that generated the content. Because these caches are controlled by the application provider, this gives the application more control.



Because applications control both the origin server and the cache, they can redirect users to the caches themselves. For example, if you request a YouTube video page from the origin server, the reply might contain the HTML (video title, comments). The HTML might then include links to specifically fetch the video and images from the proxy caches (e.g. load from static.youtube.com instead of www.youtube.com).

Speeding Up HTTP with Caching: Benefits and Drawbacks

Caching benefits everybody. The client gets to load pages faster, because they can avoid making duplicate requests, and use nearby proxies. The ISPs benefit because there are fewer HTTP requests/responses being sent over the network, so they can build less bandwidth. Servers benefit because users make fewer requests, and they don't need to process as many requests.

Clients, ISPs, and servers all care about giving good performance to the client. The client wants to watch videos in high quality, and ISPs and applications will get more customers by delivering good performance. Caching helps everybody achieve this, because the client can get their request served more efficiently from a closer cache (local, or in-network), with less latency. Also, recall that TCP throughput and RTT are inversely proportional, so a shorter RTT to a closer server means that we get higher throughput. This is especially helpful for large content like videos.

When thinking about caching, we have to consider whether the content will change on future requests.

Some HTTP resources are static. If you make a request for the Google logo, it stays the same across multiple requests.

Other HTTP resources are dynamic. If you make a Google search request, the response might change depending on who asks and when they ask. The server needs to dynamically generate a different response for every request.

Some resources are static and can be cached and served from proxy or managed caches, while other resources must be dynamically generated. For example, if you make a Google search, the HTML response probably needs to be dynamically generated by the origin server. However, the HTML can include a link to fetch the Google logo, a static resource, from one of the managed cache servers.

Conveniently, larger resources like images and videos are static, and can be cached aggressively. Dynamic content, like customized HTML pages, tend to be smaller. Clients can request the dynamic content from the origin server (far away), and use caches and proxies (closer) for all the static content.

Speeding Up HTTP with Caching: Implementation

To implement caching, we'll need to use headers to carry some metadata about caching (e.g. how long to cache the data). This is another example of headers allowing extensibility of the original protocol (which did not support caching).

The original legacy caching functionality in HTTP/1.0 used the Expires header, which just specified how long the data can be cached. In HTTP/1.1, a more sophisticated Cache-Control header was introduced. To support compatibility, some web servers will return data with both headers. HTTP/1.0 clients won't understand the newer Cache-Control header and will ignore it. HTTP/1.1 clients will prioritize the newer Cache-Control header over the older Expires header.

The Cache-Control header specifies what types of caches can cache the data, and how long the data can be cached. For example, if the resource is dynamic and changes per user, but stays the same across time for a specific user, then the server could reply with: Cache-Control: private, max-age:86400. This says that this content should only be stored in a user's local cache (not in shared proxy/managed caches), and can be stored for one day (86400 seconds).

Some data cannot be cached (e.g. dynamic content that changes frequently). In this case, the server can set Cache-Control: no-store to indicate that the client and proxy cannot cache the content.

The Cache-Control header is optional, so there's no guarantee that the client will read or respect the header. You can think of this header as a request from the server cache something. This is especially a concern for proxy caches, which are not operated by the application provider. By contrast, a private cache is run by the client (i.e. their browser) and breaking rules only affects the client themselves. A managed cache is run by the same application provider, so they can enforce that rules from the origin server are obeyed by the managed caches.

This header can be used for more complex policies as well. For example, the server might say, you can cache this data, but before you use the cached data, please make an HTTP HEAD request to re-request the header and re-validate the data. If the header indicates that the data has changed, invalidate the cache.

Content Delivery Networks (CDNs)

Earlier, we saw that managed caches are a good strategy for caching and improving user performance. Unlike private caches, they're shared between users (e.g. a user requesting something for the first time can be served by the cache). Also, unlike proxy caches, they're controlled by the application provider, which gives the application more control. The application can ensure that the caches follow rules set by the origin server, and the origin server can control which caches the user is redirected to.

Deploying managed caches across the network leads us to the idea of **content delivery networks (CDNs)**, which are sets of servers in the network serving content (e.g. HTTP resources).

For good performance, we try to put CDNs close to end users. Here, close means geographically close, but also close from a network perspective (fewer hops).

CDNs give us all the benefits of caching. Users get higher-performance delivery of content, since servers are closer. We can reduce the amount of network bandwidth and infrastructure needed, since users are making most requests to nearby servers instead of a single origin server (possibly far away).

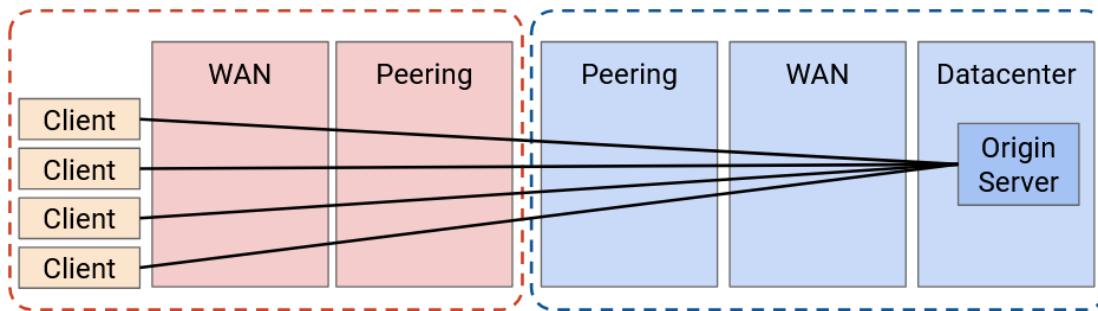
CDNs allow providers to scale their server infrastructure more easily. With a single origin server, we'd have to scale that server by making it incredibly powerful and giving it incredibly high bandwidth. By contrast, with CDNs, we can scale just by adding more small servers throughout the Internet.

CDNs also provide better redundancy for providers. If a single origin server goes down, the service might become unavailable. By contrast, with CDNs, if one server goes down, users can still be redirected to other servers.

CDN Deployment

Recall our model of the Internet: The client's request is forwarded through WAN routers (owned by the ISP) until it reaches a peering location. Then, the request goes to a peering location in the application provider's network. The request goes through the application's WAN networks until it reaches a datacenter network, where the origin server lives.

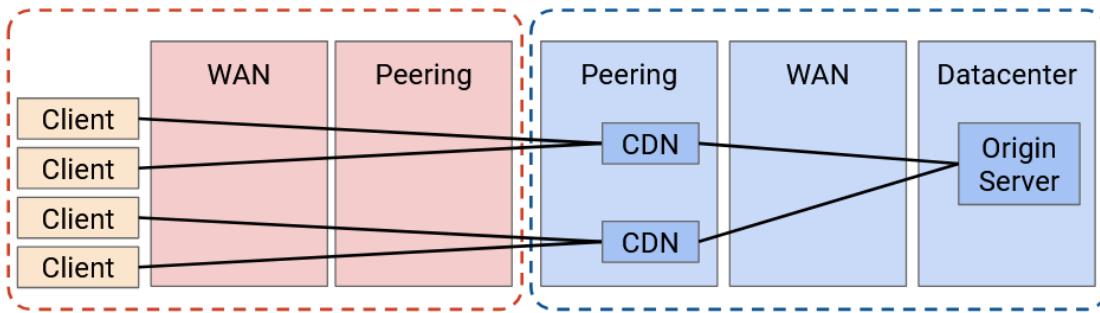
If we don't deploy any CDN, every request has to reach the origin server. This has the maximum latency (compared to later options with CDNs), resulting in the lowest performance. Also, this requires the most bandwidth to be traversed, which means we have to build more bandwidth. Finally, this requires the origin server to scale to handle every request.



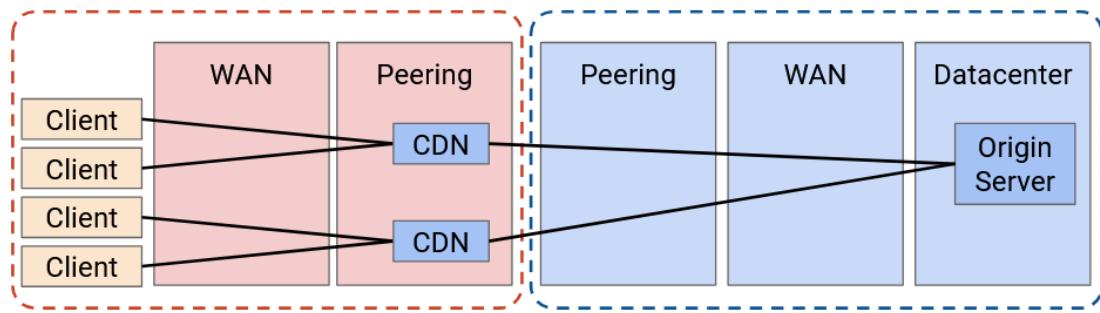
A better option would be to deploy some CDN servers at the edge of the application provider network. For example, if Google's network peers with ISP networks in New York, we could put some CDNs there.

Now, the amount of bandwidth sent over the application provider's network is much lower. The origin server sends the video to the CDN once, and the CDN can serve that video to many users. The application network no longer needs to scale its WAN network.

Also, as we saw earlier, we can now scale by adding more CDNs, instead of upgrading a single origin server. We also have more redundancy.



We can do even better and push caching deeper into the network. Now, the application is deploying servers inside the ISP's network.

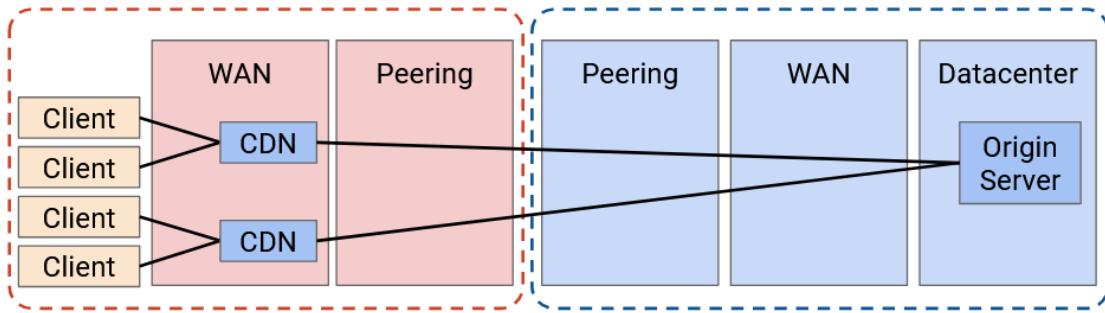


Why would an ISP agree to let the application deploy a CDN in their network? It turns out this is mutually beneficial for everybody. The ISP's customers will get better performance because they can use this new closer CDN. Also, the heavy traffic between users and the CDN is now all contained within the ISP's network. This means that ISP needs less bandwidth in the peering connection between the ISP and the application (since the content is sent only once across that peering connection).

In practice, the ISP and CDN often cooperate to deploy servers. For example, the application provides the servers for free, and the ISP connects the server to the network for free. In some cases, the ISP and CDN need to negotiate on some payment (CDN to ISP, or ISP to CDN), depending on where in the network the server is deployed, and the costs of the server and the connectivity. Still, both parties have an interest in deploying these servers.

We could try to go even further, but eventually, we encounter cost-benefit trade-offs. In the most extreme case, we could deploy a CDN in everybody's home, but the cost probably outweighs the benefit. In particular,

CDNs work best when multiple users are using it. The collective cache is larger, and one deployment can reach many users.



More generally, there's a trade-off between the cost of adding new CDNs, and the money you save from building less bandwidth. In practice, CDNs do exist in ISP networks because they're still profitable to install there.

A 2023 Sandvine (packet inspection company) report showed that 15% of all Internet traffic is from Netflix, 11.4% of traffic is from YouTube, and 4.5% is from Disney+. If an ISP installs servers for just these three applications in their network, they could potentially build 25% less network capacity.

Major application providers like Google, Netflix, Amazon, and Facebook deploy CDNs, both in their own networks, and in ISP networks.

If you're an application provider, you might not be a tech giant like Google or Amazon, but you still want your content to be served through a CDN for good performance. These smaller applications probably can't afford to install their own CDNs. However, companies like Cloudflare, Akamai, and Edgio have deployed CDNs, and you can pay those companies to deploy your content on their existing CDN. These CDN providers also deploy infrastructure in both their own networks and in ISP networks.

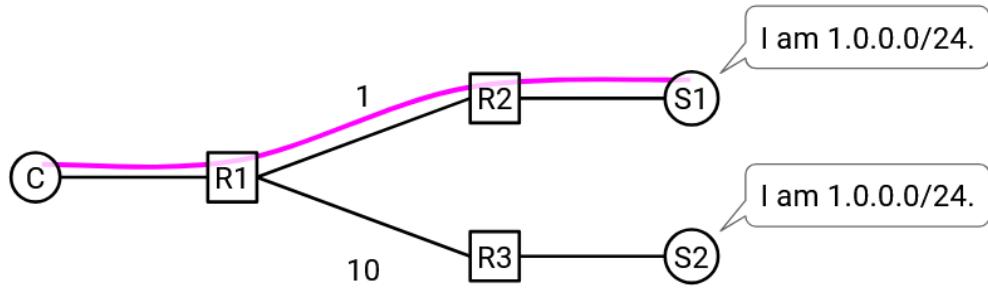
CDNs can also be used by ISPs, because ISPs themselves can also have applications serving content. When you pay for Internet service, the ISP might also offer TV service (live TV, or video on demand). These ISPs install their own CDNs to efficiently serve that TV content to you.

Fundamentally, the servers in CDNs are the same as any other servers on the Internet providing content, though they are often highly optimized for storing and delivering large amounts of content. Some servers might be better at storing and serving large amounts of content, while others might be better at rapidly serving smaller pieces of content to a large number of customers.

Directing Clients to Caches

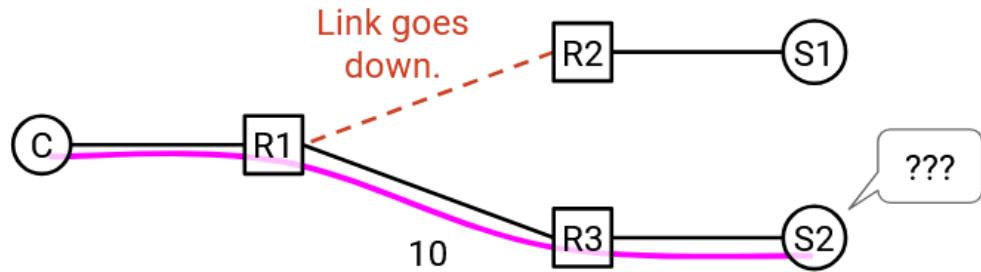
In a CDN, many different servers throughout the Internet are providing the same content. How does the client know which server to contact?

Some of the tricks from DNS can also apply to CDNs. We could use anycast, where multiple servers advertise the same IP prefix. This allows the routing algorithm to find the best path to any one of the servers.



One problem with anycast is with long-running connections. Suppose the client has an ongoing TCP connection with one of the servers. During the connection, some intermediate link in the network fails. Since all the servers have the same IP address, from an intermediate router's perspective, forwarding to any of the servers is valid. The intermediate router may now start forwarding packets to a different server (with the same IP address). However, the TCP connection was with the original server, and this new server has no way to continue the original connection.

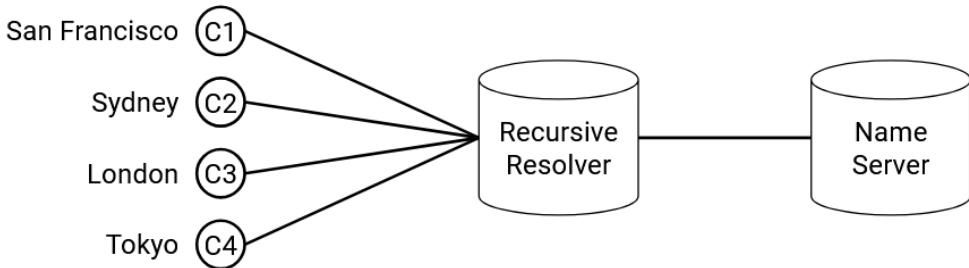
Note that this problem didn't apply when we used anycast in DNS, because DNS connections are very short (usually just one UDP packet).



We could also use DNS to load-balance. Unlike in anycast, the servers now have different IP addresses, though they still all have the same domain. When the client queries for the domain-to-IP mapping, the DNS name server can provide a different IP address depending on the client's location.

This DNS-based approach doesn't have the same problem with long-lived connections that anycast did, because the servers now have different addresses. The router won't suddenly start forwarding packets to a different server.

One problem with the DNS-based approach is lack of granularity. As an extreme example, suppose everybody in Comcast's ISP used the same recursive resolver. This means that everybody sends their DNS queries to the resolver, who then makes the query to the application name server. The application name server can only see that the DNS request came from Comcast, and has to give a single IP address back to Comcast. Now, every user in Comcast's network is using the same server, even if the users are all over the world.



A more robust approach than anycast or DNS is application-level mapping. When the origin server receives an HTTP request, the links in the response can point to different servers (e.g. static1.google.com or static2.google.com, two servers in different places), depending on where the request came from. Or, the origin server can reply with an HTTP 300-level status code to redirect the user to the appropriate server.

This application-level approach doesn't have the granularity problem of DNS, because the application can see the client's address in the HTTP request. This also doesn't have the anycast problem, since different servers can have different IP addresses.

However, just like in DNS load-balancing, the application still needs some way to guess the closest server to the client (where close might be geographic or based on network topology).

One benefit of application-level mapping is additional granularity depending on the content. For example, popular videos can be deployed to lots of servers, allowing every client to get the video from a nearby server. By contrast, unpopular videos that are rarely accessed can be deployed to fewer servers, and require users to go further for the content.

Newer HTTP Versions

As the Internet grew, HTTP started to be used by more and more applications, because it's a very generalizable protocol.

Eventually, HTTP security became an increasing concern. A bank server running HTTP probably doesn't want information to be sent in human-readable plaintext over the network, where intermediate routers or malicious attackers can read it.

HTTPS is an extension to HTTP that introduces extra security. A protocol called TLS (Transport Layer Security) is built on top of TCP, where users exchange secret keys and encrypt messages before sending them through the bytestream. HTTPS has the same fundamental protocol, but now runs on top of TLS (which itself is over TCP), instead of directly over raw insecure TCP. In recent years, there's been a push for websites to upgrade to HTTPS, and as of 2024, over 85% of websites now default to using HTTPS.

HTTP/2.0 was introduced in 2015, and was the first major revision to the protocol since 1997. The main goal of the revision was to improve performance by reducing latency and improving page load speed.

HTTP/2.0 introduced server-side pushing, where the server can send a response even if the client doesn't make a request. This allows the server to predict and preemptively serve something the user might need, without waiting for the user to make a request. For example, if we make a Google search, the HTML of the results comes back. Then, the user's browser parses the HTML, realizes it needs the Google logo, and makes another HTTP request. With HTTP/2.0, the server can preemptively give the Google logo to the

user, without waiting for the user request.

HTTP/2.0 had other performance improvements. Headers can be compressed to save space. Requests and responses can have priorities set, so that high-priority content (e.g. text of the search results) is delivered before low-priority content (e.g. the Google logo). Simultaneous requests can be multiplexed more efficiently. If the first request has a 4 GB response and the second request has a 1 KB response, a naive implementation might cause the second response to be stuck waiting for the first one to finish. HTTP/2.0 allows for smarter management of these responses.

HTTP/2.0 is widely adopted by both clients (e.g. modern browsers) and servers (e.g. CDNs).

HTTP/3.0 was introduced in 2022 (not long after HTTP/2.0, compared to the gap between 1.1 and 2.0). The semantics are the same as HTTP/2.0, but it replaces the underlying transport layer protocol. Instead of running over the TCP bytestream, HTTP/3.0 runs over a new transport protocol called QUIC, which is custom-built to work well with HTTP/3.0. QUIC = Quick UDP Connections, was designed at Google, and standardized in the IETF.

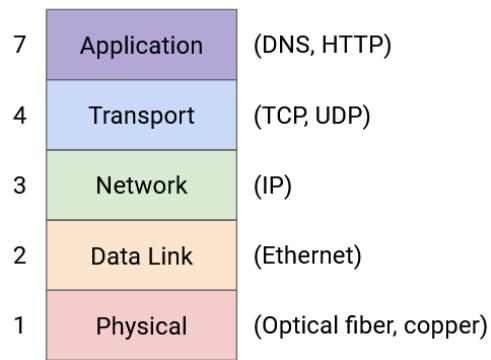
HTTP/3.0 is an example where we intentionally abandon one of the core networking paradigms (layering) in exchange for better efficiency. By giving designers the freedom to customize both the transport layer (QUIC) and application layer (HTTP/3.0) protocols, we can design both protocols to work well together.

Ethernet

Local Networks

In this section, we'll focus on what happens inside a local area network, such as the network in your home with your computer and your home router. This is in contrast with the wide-area networks we've been seeing so far, which span longer distances.

In particular, we'll look at forwarding and addressing at Layer 2. We'll have to define how packets are forwarded from a local host to a router. We'll also see how hosts in the same local network can exchange messages at Layer 2, without a need to contact routers at all. The predominant protocol at Layer 2 is Ethernet.

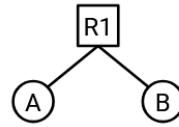


Connecting Local Hosts

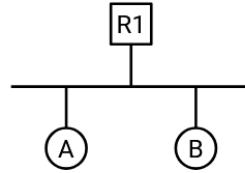
So far, we've drawn links connecting exactly two machines. In the local network, we drew a line connecting each host to the router.

In reality, a single wire might be used to connect multiple machines. In the local network, the hosts and the router can all be on the same wire. We can abstract even further and note that at Layer 2, the router is really just a machine like any other (that happens to run routing protocols at higher layers). Ultimately, the wire doesn't really care what the connected machines are doing with the data they exchange.

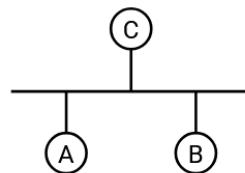
So far, we've assumed that every link connects exactly two machines:



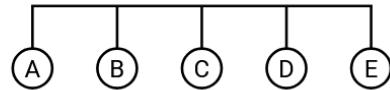
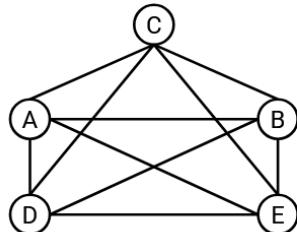
In reality, a single wire can connect multiple computers:



From the wire's perspective, the router is just a machine like any other:

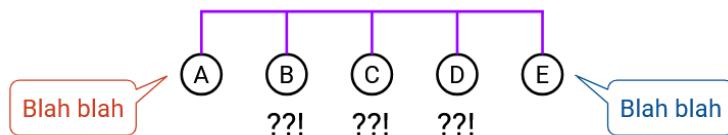


What is the best way to wire up computers in a local network? Earlier, when we first introduced routing, we thought about using a mesh topology to connect all pairs of computers in the world. We also considered using a single wire to connect up all the computers. Ultimately, we decided that for a global network, neither approach was practical, and we needed to introduce routers.



We can consider these topologies again in the local network. A mesh topology is still pretty impractical. If a new host joins, we'd have to add a wire connecting it to every other host. However, a **bus** topology, where we connect all the computers along a single wire, is pretty common and practical in a local network.

The single-wire bus topology introduces the notion of a **shared media**. When we drew links connecting two machines, only those two computers used that link to communicate. Now, a packet from A to C, and a packet from B to D, might be on the wire at the same time, and the electrical signal on that wire cannot hold both packets simultaneously.

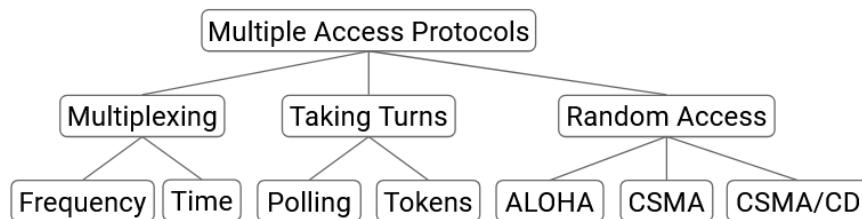


As an analogy, consider multiple people on a group call, sharing a single phone line: Any two people can talk to each other, but you can't have two simultaneous conversations, or else nobody understands what's being said.

We've drawn links as wires with electrical signals on them for simplicity, but in reality, the link technology could use other shared media. For example, in a wireless link technology, all hosts connected by the link share the same part of the electromagnetic spectrum.

Communicating over Shared Media: Coordinated Approaches

In a network with a shared medium, there's a risk that transmissions from different nodes may interfere or collide with each other. If two computers try to transmit data simultaneously, their signals will overlap and interfere. The recipients may be unable to decode the signal, and they can't tell who sent the signal. To solve this problem, we need a **multiple access protocol** that ensures that multiple computers can share the link and transmit over it.

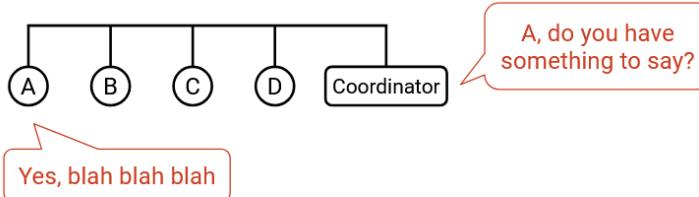


One possible category of approaches is to allocate a fixed portion of resources to each node on the link. There are two ways we could consider dividing up the resources. In **frequency-division multiplexing**, we allocate a different slice of frequencies to each computer. (Consider AM/FM radio or broadcast TV, which divide up frequencies into channels.) In **time-division multiplexing**, we divide time into fixed slots and allocate a slot to every connected node.

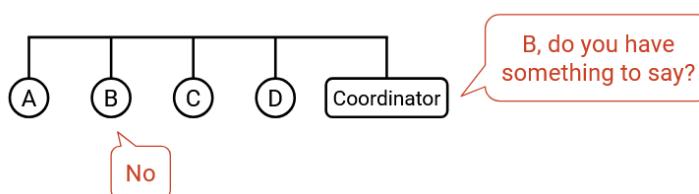
Fixed allocation of resources has some downsides. There's only a limited amount of frequency/time to distribute. Also, not everyone has something to say all the time, so the frequency/time we allocate might go unused most of the time. This approach is wasteful, because it confines computers to their specific allocated slice, even while other slices might be unused.

Instead of fixed allocation, another category of approaches are based on the nodes taking turns, without any fixed allocations. In this category, we're dynamically partitioning by time, so that nodes use only the time they need during their turn, with no wasted time. There are two ways we could consider having nodes take turns.

In a **polling protocol**, a centralized coordinator decides when each connected node gets to speak. The coordinator goes to each node one by one and asks if the node has something to say. If the node says yes, the coordinator lets the node speak for some time. If the node says no, the coordinator immediately moves on to the next node, and the node doesn't waste any resources. Bluetooth is a real-world protocol using this idea.

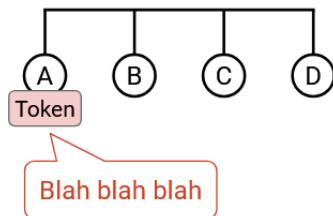


A can speak for as long as it needs.

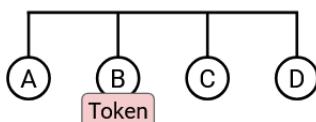


B has nothing to say.
Coordinator can immediately move on to C.

The other way to let nodes take turns is **token passing**. Instead of having a centralized coordinator, we have a virtual token that can be passed between nodes, and only the node with the token is allowed to speak. If a node has something to say, it holds onto the token while transmitting, then passes it to the next node. If a node doesn't have anything to say at the moment, it immediately passes the token to the next node. IBM Token Ring and FDDI are real-world examples of protocols that use this idea.



A holds the token.
A can speak for as long as it needs.
When A is done, it passes the token to B.



B has nothing to say.
It can immediately pass the token to C.

One downside to these turn-based approaches is complexity. We have to implement some form of inter-node communication, which could get complicated. In token passing, we might need some dedicated frequency channel for nodes to reliably pass the token between each other. We might also have to deal with complications like two nodes both thinking they have the token and causing a collision. In a polling protocol, we need to designate a central coordinator to communicate with nodes, and implement a way for the coordinator to talk to the nodes. In Bluetooth, your smartphone can be the central coordinator talking to auxiliary devices, but in other networks, it might not be obvious who the coordinator is.

Communicating over Shared Media: Random Access Approaches

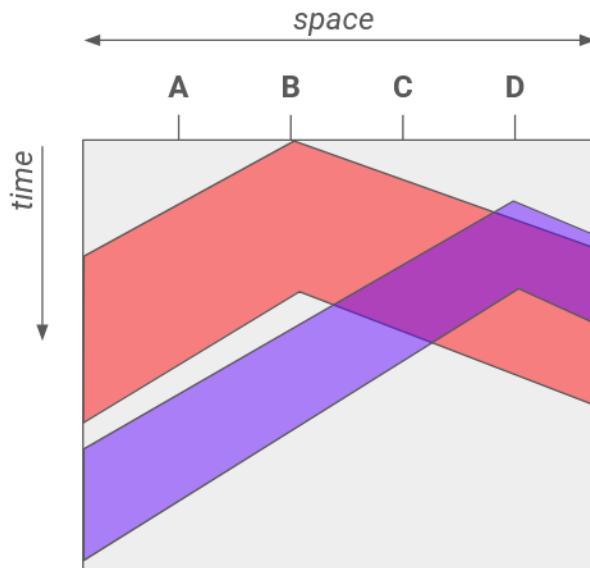
A third category of approaches, besides fixed allocation or taking turns, is **random access**. In this approach, we just allow nodes to talk whenever they have something to say, and deal with collisions when they occur. The nodes don't coordinate between each other, and just send data whenever they have something to send.

One major benefit of random access protocols is simplicity. Unlike the turn-based approaches, we don't need to implement inter-node communication.

When the recipient gets a packet, it replies with an ack. If two nodes send data simultaneously, the collision causes their packets to be corrupted, so no ack is sent. If the sender doesn't see an ack, it waits some random amount of time and re-sends. Waiting some random amount of time, instead of re-sending immediately, helps us avoid collisions when the packets are resent.

The naive random access protocol is “rude” because nodes start talking whenever they want, and deal with collisions afterwards. A more “polite” variant of this protocol is called **Carrier Sense Multiple Access (CSMA)**. Nodes listen to the shared medium first to see if anybody is speaking, and only start talking when it is quiet. Here, “listen” refers to sensing a signal on the wire.

Note that CSMA does not help us avoid all collisions. If signals instantaneously propagated along the entire length of the wire, there would be no collisions in CSMA. However, propagation delay can introduce issues. Suppose node A on one end of the wire hears silence and starts transmitting. The signal might not have propagated to node B yet, on the other end of the wire. Node B hears silence and also starts transmitting, causing a collision.



This 2D diagram demonstrates how propagation delay can cause conflicts. A horizontal cross-section shows the wire at an instant in time, and lets us see how far the signal has propagated across the wire at that instant. A vertical cross-section shows a single location on the wire across time, and lets us see when that location sees the first and last bits of the transmission. Both H2 and H4 hear silence before they start transmitting, but their signals still collide.

To mitigate this problem, we can use **CSMA/CD** (Carrier Sense Multiple Access with **Collision Detection**), which extends the idea of CSMA. In addition to listening before speaking, we also listen while we speak. If you start hearing something while you're transmitting, you stop immediately. Note that CSMA/CD still doesn't fix the problem of collisions, but it allows us to detect collisions sooner.

If there's only one speaker, there won't be any collisions, and all of our random access schemes should work fine. If there are only a few speakers, there might be occasional collisions, but all of our schemes can deal with them. However, if many senders want to talk simultaneously, we may have problems with repeated collisions, and waiting a random amount of time to re-send won't help.

To deal with repeated collisions, CSMA/CD uses **binary exponential backoff**. Each time we detect a collision on a retransmission attempt, we wait up to twice as long before the next retransmission. Note that we still randomly choose the retransmission time, but each time we detect a collision, we choose the random number from a range with a limit that's twice as high. For example, if we chose a random time in the range [0, 4] and detected a collision, the next random time we choose is in the range [0, 8].

Binary exponential backoff works well in both scenarios. When there are a few nodes speaking, repeated collisions are uncommon, so we can retransmit after a short wait time. When there are many nodes speaking, there are many repeated collisions, so the delay increases exponentially until there are no collisions (e.g. enough nodes have been delayed far into the future, and there are fewer nodes competing right now). This approach ensures we only slow down when many nodes want to speak, and maintains fast transmission when few nodes want to speak.

Brief History of Layer 2: ALOHANet

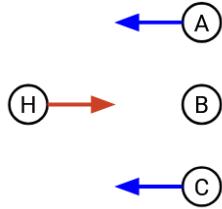
In 1968, Norman Abramson had a problem at the University of Hawaii. There was a central computer at the University of Hawaii, and he needed a way for computers on other islands to access this central computer. The resulting design was very influential to modern Layer 2 protocol designs.

The resulting protocol was called ALOHANet (Additive Links On-line Hawaii Area), which allowed wireless communication from other islands to the central computer. ALOHANet was wireless and used a shared medium, where everybody is sending data over the same link.

ALOHANet used a combination of fixed allocation and random access, because of its asymmetric setup. The central computer (hub) used its own dedicated frequency to transmit outgoing messages, and all remote nodes listened on this frequency to receive messages. With only one sender on a dedicated frequency, there's no risk of collisions.

By contrast, all the remote nodes transmit on a separate shared frequency, and the hub listened to this frequency. The hub won't collide with the remote nodes, because they use different frequencies, but the remote nodes could collide with each other.

This asymmetric design worked well for ALOHANet because the hub probably has more to send than the remote nodes.



ALOHANet was one of the first systems to use a random access protocol to handle collisions, and this approach would later be used in Ethernet. ALOHANet used the naive rude approach to random access. Later protocols like Ethernet used the more polite approach of CSMA/CD, where we listen for collisions before and during transmission, and we back off exponentially when there are collisions.

LAN Communication: MAC Addresses

Because multiple computers can be connected along the same Ethernet link, we can actually use Layer 2 protocols to send messages between local computers on the same link, without using any Layer 3 protocols at all (e.g. no routers forwarding packets). In the postal system analogy, two people in the same room can pass letters between each other, without sending the letter to the post office.

One problem with sending messages over a shared media is: When we transmit the message, everybody on the link gets the message, not just the intended recipient. To send a message to just one person, we need an addressing system at Layer 2 so that we can identify which machine the message is intended for. In the postal system analogy, if I speak in a room, everyone gets the message. To talk to one specific person, I need to refer to them using their name.

At Layer 2, every computer has a **MAC address** (Media Access Control). MAC addresses are 48 bits long, and are usually written in hexadecimal with colons separating every 2 hex digits (8 bits), e.g. `f8:ff:c2:2b:36:16`. MAC addresses are sometimes called ether addresses or link addresses.

MAC addresses are usually permanently hard-coded (“burned in”) on a device (e.g. the NIC in your computer). Most OSes will let you override the MAC address in software, but every device already comes with a MAC address installed. MAC addresses are allocated according to the manufacturer that creates the hardware. The first two bits are flags, then the next 22 bits identify the manufacturer, then the last 24 bits identify the specific machine within that manufacturer’s address space.

Why not just use IP addressing? Hosts on a link might want to exchange messages, without ever being connected to the Internet (i.e. they don’t have an IP address at all).

This permanent addressing scheme is different from IP, where you receive an address when you first join a network, and the address depends on your geographic location. MAC addresses are usually supposed to be globally unique, because you might plug your computer into any local network, and it’d be bad if two computers on a link had the same MAC address.

LAN Communication Types, Ethernet Packet Structure

There are different possible destinations in a Layer 2 packet. In **unicast**, the packet is intended to a single recipient. In **broadcast**, the packet is intended for all machines on the local network. In **multicast**, the

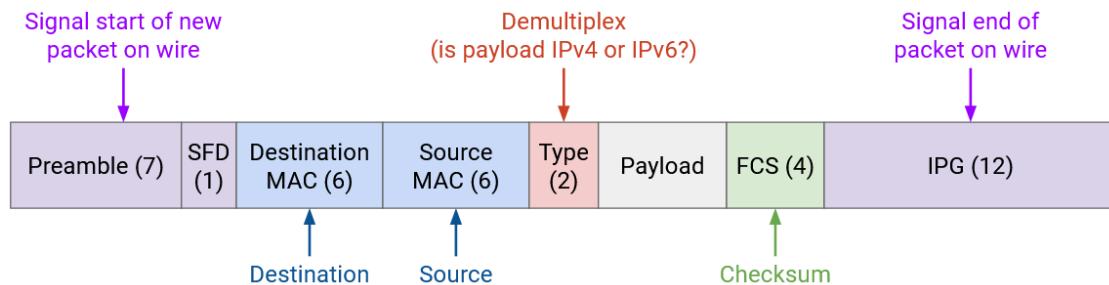
packet is intended for all machines in the local network that belong to a particular group. Machines can choose to join certain groups to receive packets meant for that group. Ethernet supports unicast, multicast, and broadcast.

Note that broadcast is sometimes thought of as a special case of multicast, where everybody is automatically part of the broadcast group.

This unicast/broadcast/multicast model extends to other layers too. For example, we saw anycast at Layer 3, where the goal was to send to any one member of a group (any of the servers with the same IP address).

Ethernet Packet Structure

A data packet in Ethernet is called a **frame**. Many fields look similar to the IP header fields, though there are some differences.



The Ethernet packet starts with a 7-byte preamble, which indicates the start of a packet. This helps separate packets as they're signalled across the wire.

Then, we have the destination and source MAC addresses, similar to the destination and source fields in the IP header. We have a 2-byte type field, which allows us to demultiplex between IPv4 or IPv6, and pass the packet payload to the correct next protocol. This is similar to the protocol field in the IP header, or the port field in the TCP/UDP headers. We also have a checksum, though unlike IP, the checksum is over the entire packet, so that we don't have to rely on higher layers (e.g. the packet might not be TCP/IP at all).

To unicast a message, we set the destination MAC address to a specific machine's MAC address. Everybody on the shared medium receives the packet, so everybody needs to check the destination MAC to see if the packet is meant for them. If the destination MAC address doesn't match your address, you should ignore the packet.

To broadcast a message, we set the destination MAC to the special address FF:FF:FF:FF:FF:FF (all ones). Just like in unicast, everybody on the shared medium receives the packet, but this time, because the destination MAC address is the broadcast address, everybody knows to read the packet. Note that this all-ones broadcast address is the same in every Ethernet network.

To multicast a message, we set the destination MAC to the address of that group. Recall that the first two bits of the MAC addresses are flags. Normal addresses allocated to machines always set the first bit to 0, and addresses for groups always set the first bit to 1. Just like in unicast and broadcast, everybody still gets the message. Anybody who's part of a group needs to make sure they're listening on that group's

address in order to receive packets multicast to that group. Additional protocols are necessary to control who belongs to which groups, and we won't discuss them further.

Layer 2 Networks with Ethernet

So far, we've shown Layer 2 protocols as operating on a single link with multiple computers attached to it, but we could introduce multiple links and build a network entirely using Layer 2. Packets could be forwarded, and machines could even run routing protocols, all exclusively using Layer 2 MAC addresses.

The routing protocols we ran at the IP layer could also work at Layer 2, though one downside is that we can't aggregate MAC addresses. IP addresses are allocated based on geography, but MAC addresses are allocated based on manufacturer, so there's no clear way to aggregate them. This downside is why we can't build the global Internet out of only Layer 2.

If there are multiple links in a single local network, we'd have to ensure that if someone broadcasts a message, any switches at Layer 2 forward the packet out of all outgoing ports.

Multicast gets more complicated in a Layer 2 network with multiple links. Additional protocols are needed, though we won't discuss further.

One example of multicast being useful on a LAN is Bonjour/mDNS, a protocol developed by Apple. In this protocol, all Apple devices (e.g. iPhone, iPad, Apple TV) are hard-coded to join a special group on the local network. If your iPhone wants to find nearby devices to play music (e.g. Apple TV, Apple speaker or HomePod or whatever they call it), the iPhone can multicast a message to the group, asking if anybody can play music. Devices in the group can also multicast responses, saying "I am an Apple TV and I can play music." Interestingly, this protocol actually also uses DNS in the multicast group to send SRV records, which maps each machine to its capabilities.

Historical note: In the modern Internet, we've said that the terms "router" and "switch" are interchangeable. Now that we have the notion of a Layer 2 network, we could say that a switch only operates at Layers 1 and 2, while a router operates at Layers 1, 2, and 3.

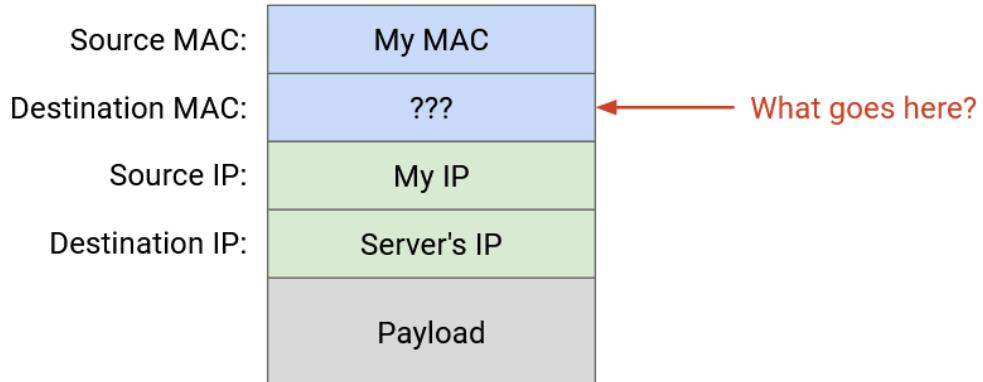
If you go back to our picture of wrapping and unwrapping headers, we've assumed that every router parses the packet up to Layer 3, and forwards the packet to the next router over IP. However, if we had a Layer 2 network with multiple links, a switch only needs to pass the packet up to Layer 2 and forward the packet to the next switch over Ethernet.

Today, pretty much all switches also implement Layer 3, which is why we use the terms interchangeably. Historically, Ethernet predates the Internet, which is why there was a distinction between switches and routers.

ARP: Connecting Layers 2 and 3

Connecting Layers 2 and 3

Recall that packets get additional headers wrapped around them as they move down the stack, to lower layers. To send an IP packet, we first fill in its destination IP at Layer 3. Then, we pass that packet down to Layer 2, where we have to add a MAC address to send the packet along the link. What MAC address do we add?



First, we need to check if the destination IP is somebody in our own local network, or somebody in a different local network. To determine this, the sender's forwarding table will have an entry indicating the range of local IP addresses, sometimes called our **subnet**. For example, the entry might say that 192.0.2.0/24 is direct, which means all addresses between 192.0.2.0 and 192.0.2.255 are on the same local network. The table also has a default route, saying that all other non-local destinations should be forwarded to the router.

If the destination IP is in our subnet, we need some way to translate between the destination IP address and that machine's corresponding MAC address. If the destination is outside our subnet, we need some way to translate the router's IP address (from the forwarding table) to its corresponding MAC address, so we can send the packet to the router.

One naive solution is to broadcast every packet, so that the destination or router will definitely receive and process it. However, this is inefficient. It forces everyone to parse every packet (e.g. read the Layer 3 headers) to check if the packet is meant for them. Also, if the Layer 2 network has more than one link, the switches at Layer 2 have to flood the packet across all the links.

A better approach would be to translate the destination IP address to its corresponding MAC address (if local) or the router's MAC address (if non-local), and unicast the packet at Layer 2.

ARP: Address Resolution Protocol

ARP (Address Resolution Protocol) allows machines to translate an IP address into its corresponding MAC address.

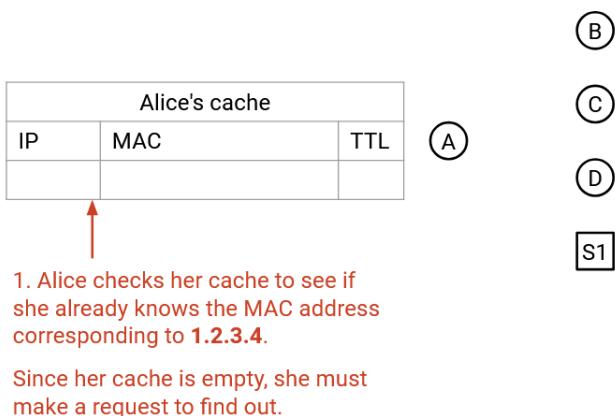
To request a translation, a machine can broadcast a solicitation message: "I have MAC address f8:ff:c2:2b:36:16. What is the MAC address of the machine with IP 192.0.2.1?"

All machines who are not this IP address ignore the message. The user who has this IP address unicasts a reply to the sender's MAC address, saying "I am 192.0.2.1, and my MAC address is a2:ff:28:02:f2:10."

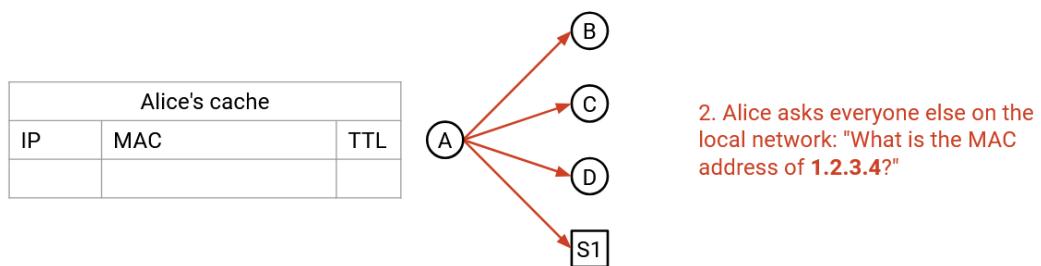
Machines can also broadcast their own IP-to-MAC mapping to everybody, even if nobody asks.

When you receive an IP-to-MAC mapping, you can add it to your local **ARP Table**, which caches these mappings for the future. The table also includes an expiry date for each entry, since IP addresses aren't permanently assigned to a computer. A different computer could get assigned the same IP address, or the same computer could change IP addresses. (TODO: interfaces?)

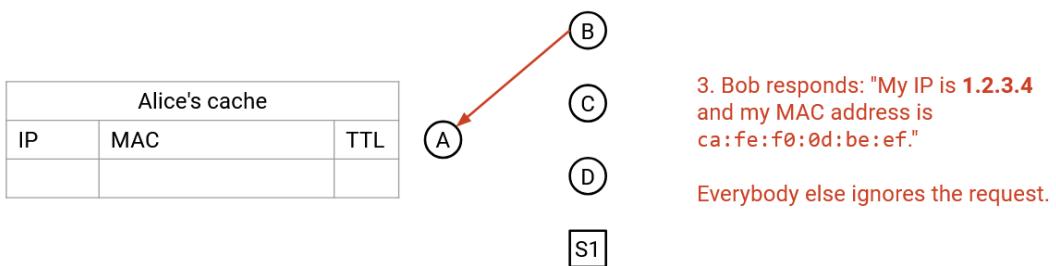
Step 1:



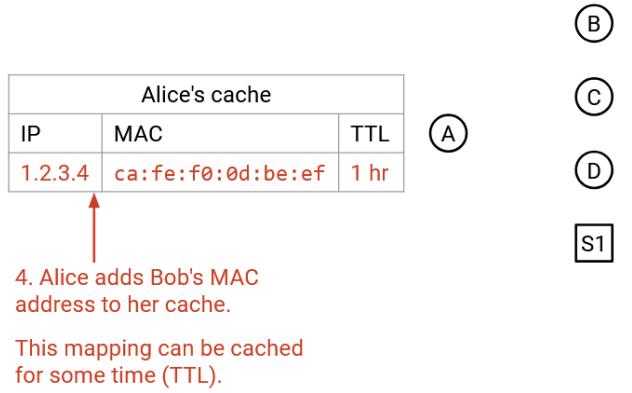
Step 2:



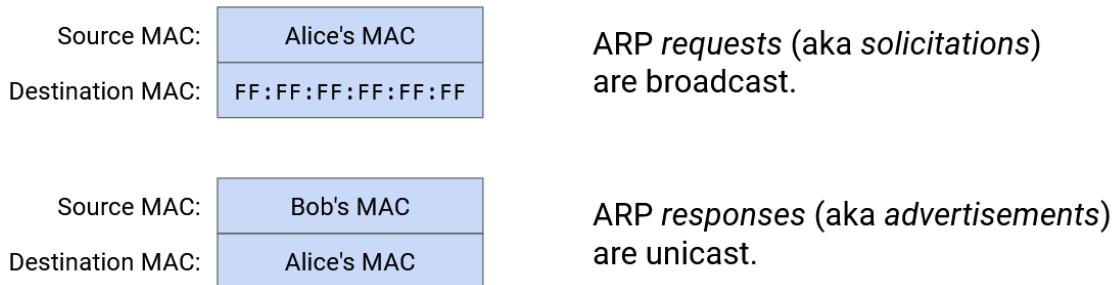
Step 3:



Step 4:



Note that ARP runs directly on Layer 2, so all packets are sent and received over Ethernet, not IP.



Connecting ARP and Forwarding Tables

Recall that in a router's forwarding table, we would sometimes include an entry indicating that a host is directly connected to the router.

In reality, the router's forwarding table contains a single entry, mapping the entire subnet's range of IP addresses to be direct. If the router receives a packet whose destination is in this local range, the router runs ARP to find the corresponding MAC address, and uses Layer 2 to send the packet to the correct host on the link.

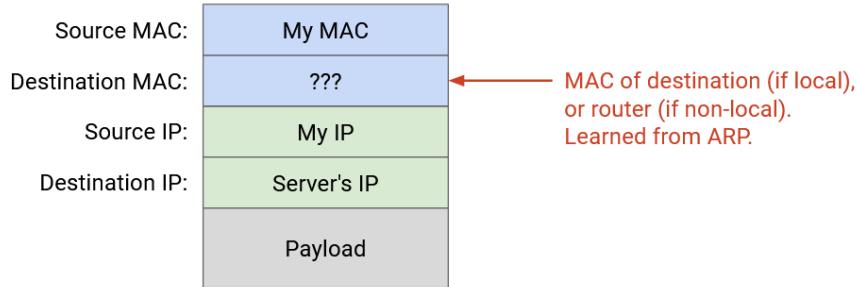


This also helps us in the case where multiple hosts are connected on the same link. In our conceptual picture, we'd say that Host A is directly connected on Port 1. Multiple hosts might be on that link, so by using ARP, we can create a Layer 2 packet that gets unicast to only Host A, and not other computers on the link.

Given a forwarding entry that maps a subnet like 192.0.2.0/24 as direct, how can we determine if a given IP address falls in that range? This is where it's useful to write ranges using a netmask instead of slash notation. Recall that to write this range as a netmask, we set all fixed bits to 1 and all unfixed bits to 0, to get 255.255.255.0. Then the range is expressed as 192.0.2.0 with netmask 255.255.255.0.

Now, to check if an address is in the range, we perform a bitwise AND of the address and the netmask. This causes all unfixed lower bits to get zeroed out, retaining only the fixed upper bits. Then, we check if the result matches 192.0.2.0 (the first address in the range, where all unfixed bits are 0).

Note that as packets get forwarded across hops, the Layer 2 destination will change to the MAC address of the next hop, so that packets can travel across links. However, the Layer 3 destination stays the same across each hop.



Neighbor Discovery in IPv6

ARP translates IPv4 addresses to MAC addresses. To translate IPv6 addresses to MAC addresses, we use a similar protocol called **neighbor discovery**.

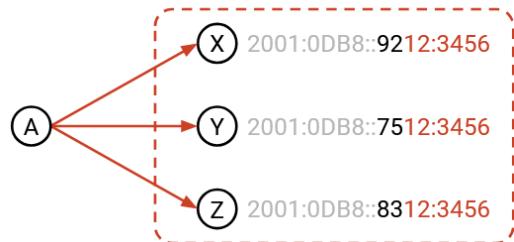
Instead of broadcasting the request for an IP-to-MAC translation, neighbor discovery instead multicasts the request to a specific group, and each computer listens on a specific group based on its IP address. For example, everyone with an IP address ending in 12:3456 might listen on the group MAC address 33:33:FF:12:34:56, while everyone with an IP address ending in 78:90AB might listen on the group MAC address 33:33:FF:78:90:AB.

If I want the MAC address corresponding to the user with an IPv6 address ending in 12:3456, I can plug those IPv6 bits into the group MAC address to get 33:33:FF:12:34:56, and I know that the user with that IP address must be listening to this group MAC address.

Everyone with IP ending in 78:90AB listens on multicast MAC address 33:33:FF:78:90:AB.



Everyone with IP ending in 12:3456 listens on multicast MAC address 33:33:FF:12:34:56.



If A wants the MAC matching 2001:0DB8::9212:3456,
A multicasts to only the 12:3456 group.

Some terminology: In the neighbor discovery protocol, the request for a mapping is called Neighbor Solicitation, and the reply containing the mapping is called Neighbor Advertisement.

DHCP: Joining Networks

Joining Networks

When a computer first joins the network, what information does it need to connect to the Internet?

We always know our own MAC address, because it's burned into the hardware.

We need to be allocated an IP address so we can send and receive packets. Recall, IP addresses are allocated geographically, so when we connect to a new network, someone has to give us an IP address to use.

We need to learn the subnet mask so that we can learn the range of local IP addresses. Given the mask (fixed bits all ones, unfixed bits all zeros), we can bitwise AND the mask with our own IP address to learn the local IP prefix.

We need to learn who the router on the local network is, so that we can send any non-local packets to the router. Sometimes we call this router the **default gateway**.

We also might need to learn where the DNS recursive resolver is located for this network.

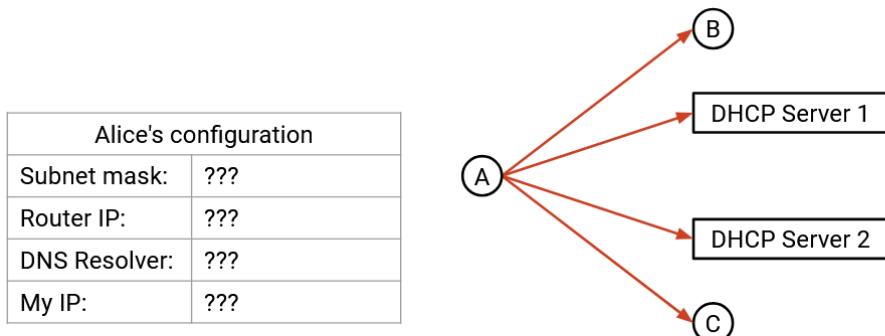
The user could manually configure these values when they first join the network. This is time-consuming, especially since we have to re-configure these values every time we join a different network. Also, the average Internet user probably has no idea how to configure these values manually. That said, manual configuration does sometimes work for machines like routers, which don't move around often.

We need a protocol that allows new hosts to automatically learn these values (and possibly other useful information).

DHCP: Dynamic Host Configuration Protocol

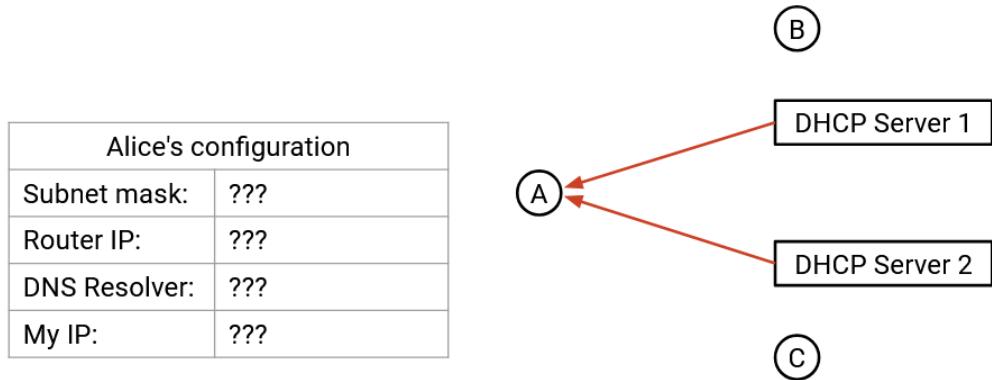
DHCP has four steps:

1. The new client broadcasts a **Discover** message, asking for configuration information.



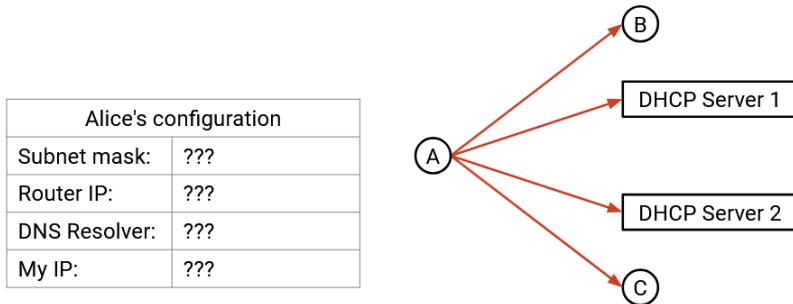
1. **Client discover:** Alice broadcasts a request:
"Can anyone give me a configuration?"

2. Any **DHCP server** who can help will unicast an **Offer** to the client, with a configuration that the client can use (e.g. IP address, gateway address, DNS address).



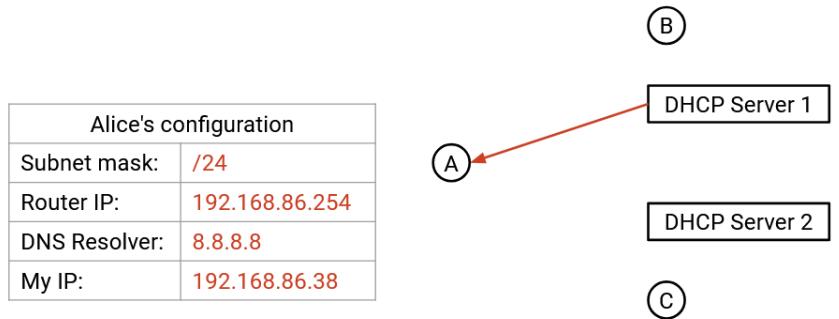
2. **DHCP Offer:** One or more DHCP servers reply with an offer for Alice.

3. The client will broadcast a **Request** message, indicating which offer they accepted. This message is broadcast because the client might get multiple offers. By telling everybody which offer it's accepting, the client allows the rejected offers to be freed up for future clients.



3. **Client Request:** Alice broadcasts which offer she chose:
"I'll use the offer from DHCP Server 1."

4. The server sends an acknowledgement to confirm that the request was granted.



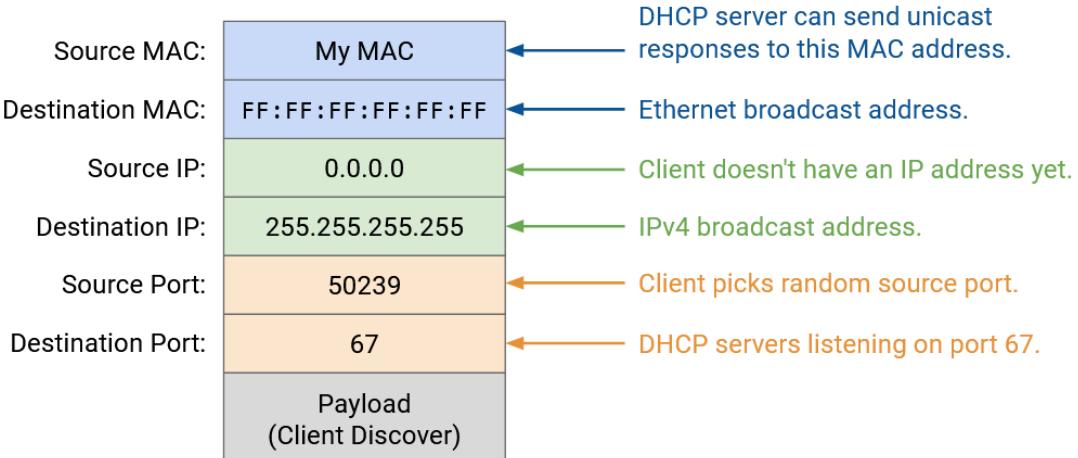
4. **DHCP Acknowledgment:** The chosen DHCP server confirms that the configuration has been set for Alice.

DHCP Servers

In step 2, who is able to offer configurations? DHCP servers are added to the network, and their goal is to offer this information to new hosts. On smaller networks like your home network, the home router itself often acts as the DHCP server. In larger networks, there could be a separate machine that acts as the DHCP server.

DHCP servers need to be in the same local network as the client, since the protocol operates inside the local network. In larger networks, we might not want to run DHCP server code inside every router, so local routers can relay requests to a central remote DHCP server that actually runs the protocol.

DHCP servers listen on a fixed port, UDP port 67, for requests from new machines. The servers are configured with all the necessary information: They know about the gateway and DNS servers, and they have a pool of usable IP addresses that they can allocate to new users.



Note that IP addresses are only temporarily leased to hosts. The lease is only valid for a limited amount of time (e.g. order of hours or days). If the host wants to keep using the address, it must renew the lease. If an IP address is currently leased to a host, the DHCP server cannot offer the same address to other clients.

DHCP Implementation

Note that DHCP is a Layer 7 application protocol, and it runs on top of UDP, which itself runs on top of IP.

In step 1, how does the client broadcast a message over IP? It sends a packet with destination IP of 255.255.255.255 (all ones), which is the IPv4 broadcast address. When this packet is passed down to Layer 2, instead of translating this IP address using ARP, the IPv4 broadcast address is mapped to the Ethernet broadcast address of FF:FF:FF:FF:FF:FF (all ones). Then, the packet can get broadcast across the network at Layer 2.

What about the source IP? The client doesn't have one at the start of the protocol, so it sets the source IP to be 0.0.0.0.

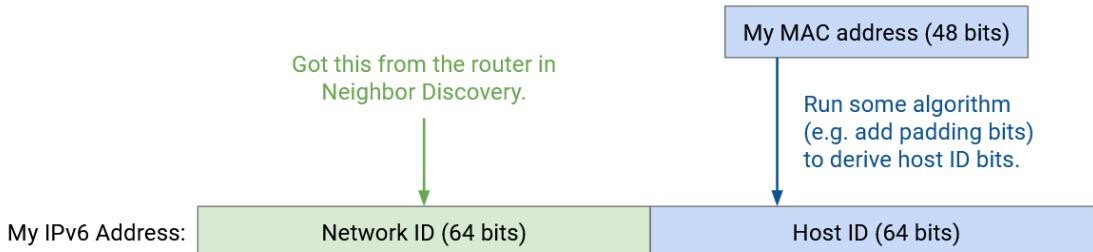
With the hard-coded source IP of 0.0.0.0 and destination IP of 255.255.255.255, the client doesn't need to know anything about the local network to start running this protocol.

If there's no source IP, how do the DHCP servers know how to unicast the offers? The DHCP servers could either broadcast the offers, or use the client's MAC address to unicast the offers.

Autoconfiguration in IPv6

DHCP also exists in IPv6 networks. However, because IPv6 addresses are longer, it turns out that we can give ourselves a guaranteed unique IPv6 address without anybody else managing a pool of addresses and leasing them. This protocol is called **Stateless Address Autoconfiguration (SLAAC)**.

The trick is to use the MAC address, which we know is unique to each machine. As before, we ask for the local network information, which includes the gateway address, DNS address, and notably, the prefix for the local network. This prefix is usually 64 bits long. Then, we copy our own MAC address bits into the host bits of the IPv6 address. We can be confident that nobody else has this IPv6 address: users in other networks will have a different prefix, and no one else in the network (or anywhere else) will have the same MAC address bits.



To get the local network information, we can extend the Neighbor Discovery protocol (IPv6 version of ARP). The Router Solicitation message lets the user broadcast a request for the local network information, and the Router Advertisement message lets routers reply with that information.

SLAAC has additional mechanisms to detect duplicate addresses, just in case.

NAT: Network Address Translation

Motivation: IPv4 Address Exhaustion

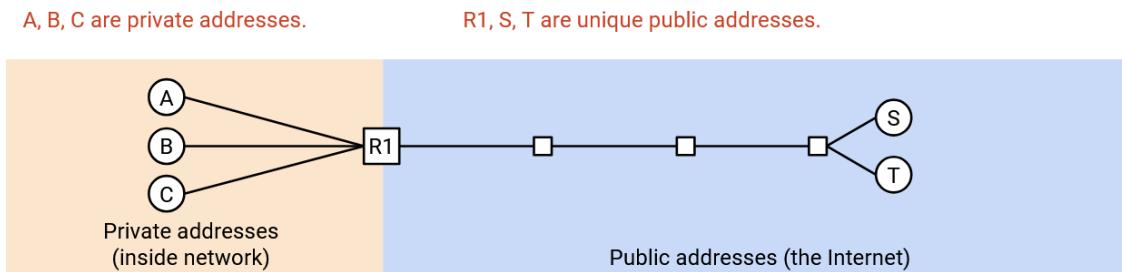
Recall that we only have 2^{32} different IPv4 addresses, which is not enough to address every host on the Internet. We've already seen that IPv6 is a robust solution to IPv4 address exhaustion, but IPv6 adoption has been fairly slow.

In the meantime, to conserve addresses, recall that IANA allocated special RFC 1918 ranges of private IP addresses, which can be used by any networks that don't require Internet addresses: 192.168.0.0/16, 10.0.0.0/8, and 172.16.0.0/12. It turns out that these addresses are often used in your home network as well, so that your own personal device doesn't need a unique IP address. But, you do need Internet access, so how can you use a private IP address?

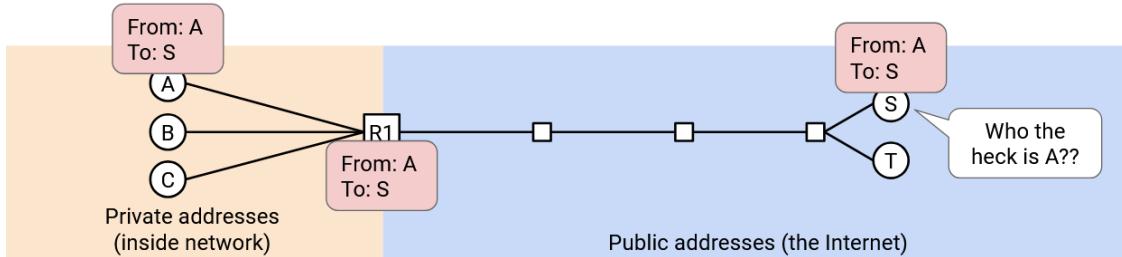
NAT: Conceptual

In NAT, the goal is to use a single public IP address to represent many hosts in the local network. The trick is to have the gateway router convert private IP addresses into the single public address before sending out messages. Then, the router converts the public address back into a private address for incoming replies.

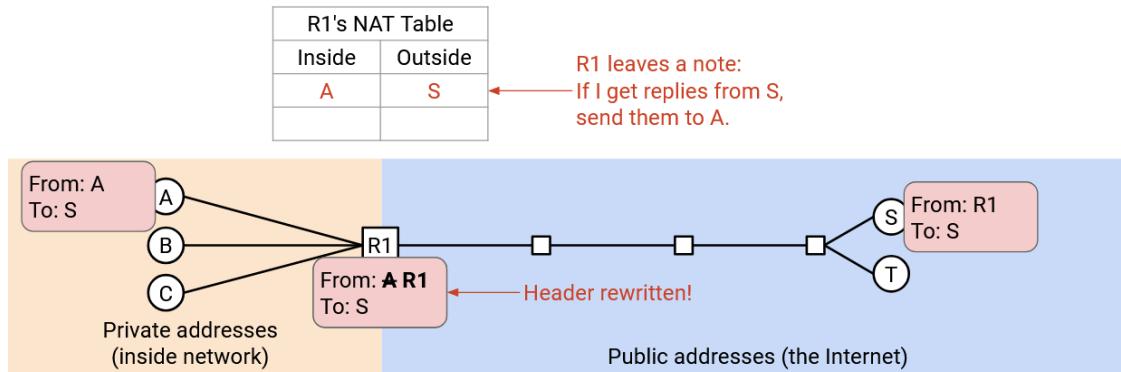
Alice, Bob, and Chuck are all working for Joe's Tire Shop. They have private IP addresses A, B, and C, which cannot be used on the wide Internet, because they're not unique. Instead, everyone in Joe's Tire Shop must share a single public IP address, which is the only unique, publicly-understandable IP address they have.



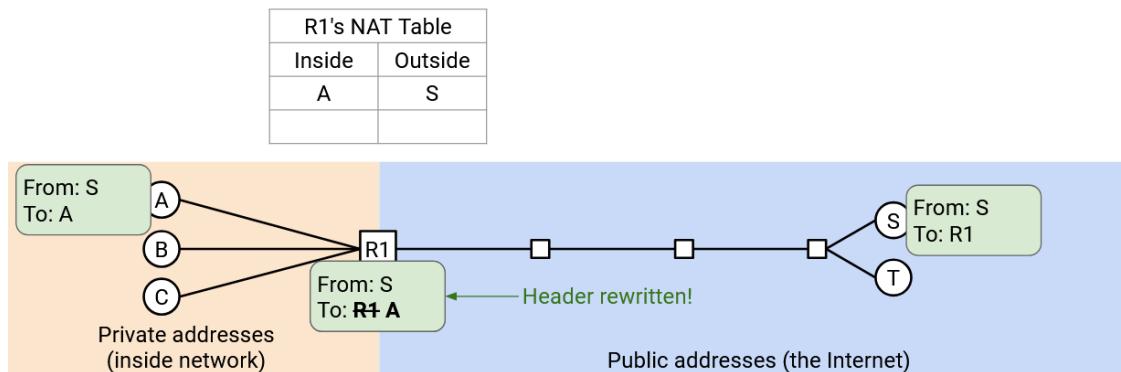
Alice wants to send a message to an external public server with public IP address S. She sends a packet that says "From: A, To: S." If we sent this packet naively, S would be unable to send replies, because A is a private IP address.



Instead, when the packet reaches the gateway router, it rewrites the header to say “From: R1, To: S.” The router also makes a note: If I get any replies from S, they should go to A.

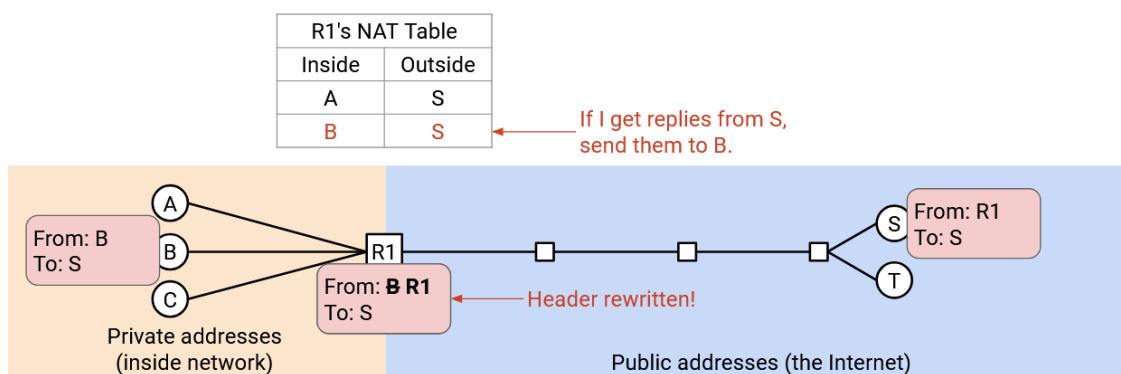


Now, when S gets a packet, it can send replies to the public address R1: “From: S, To: R1.” When R1, the gateway router, receives the reply, it checks its note, and rewrites the header to say “From: S, To: A.” Then, the packet gets sent back to A.

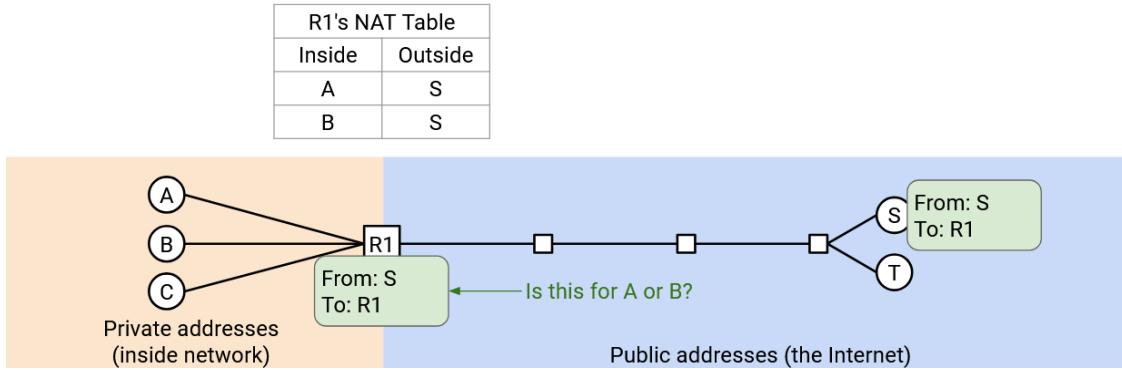


Now, Alice, Bob, and Chuck can all send outgoing packets. When the router receives a packet, it must remember a mapping between the external destination and the internal sender. (“B just sent a packet to N, so any replies from N should be sent back to B.”)

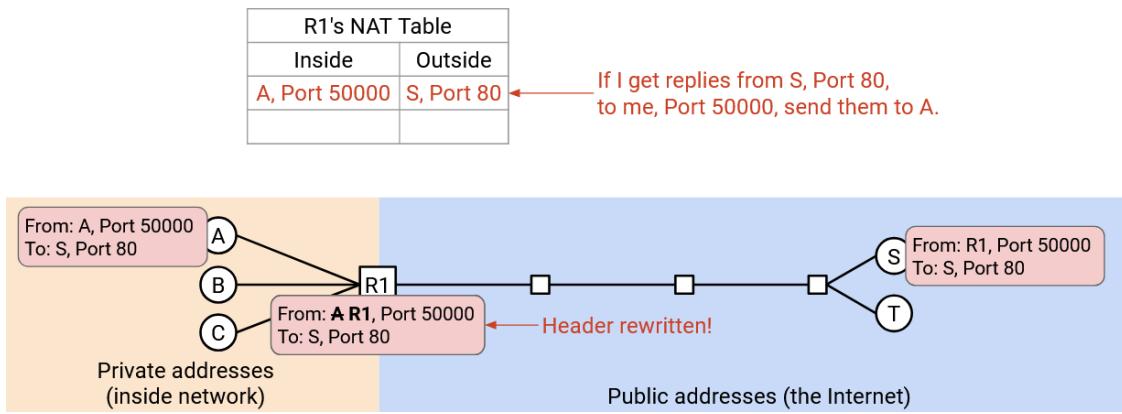
One problem arises if Alice and Bob both want to talk to S.



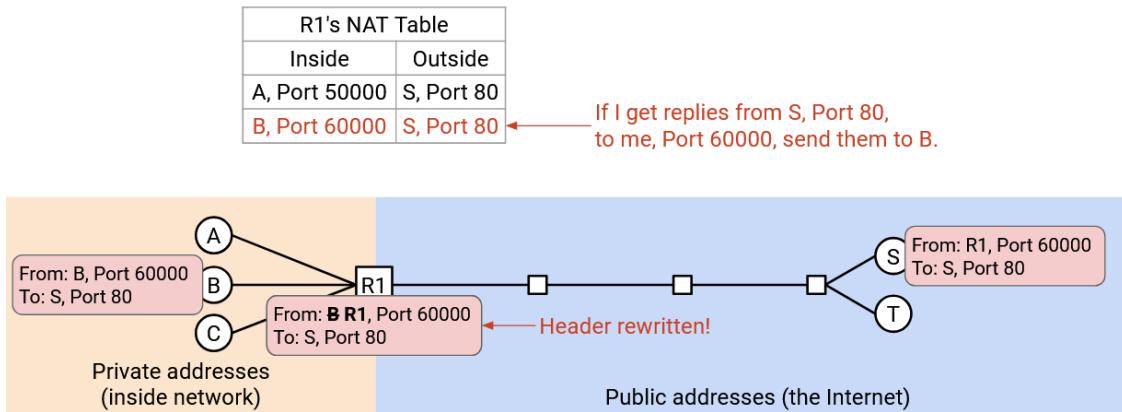
We now have ambiguity if a reply arrives from S. Should the router send this reply to A or B?



We can solve this problem by using logical ports, from Layer 4. Alice's connection says: "From: A, Port 50000, To: S, Port 80." The router rewrites this to say "From: R1, Port 50000, To: S, Port 80," just like before. The note now says, if I get any replies from S, Port 80, to R1, Port 50000, it should go to A.

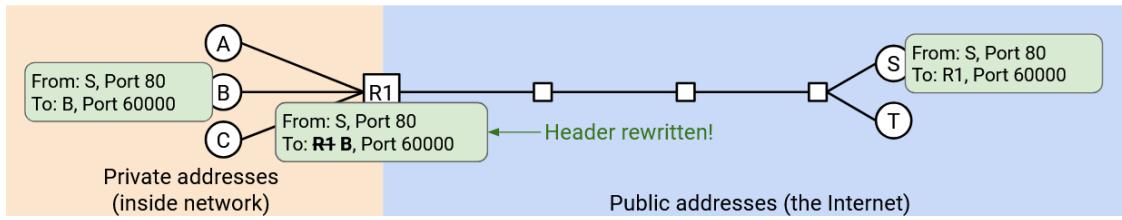


Bob could create a separate connection that says: "From B, Port 60000, To: S, Port 80." The router rewrites this to say "From: R1, Port 60000, To: S, Port 80," just like before. The note for this connection says, if I get any replies from S, Port 80 to R1, Port 60000, it should go B.



More generally, the router is now keeping track of connections using the 5-tuple of source IP, destination IP, protocol, source port, and destination port. When the router receives an outgoing packet, it changes the private source IP to the public source IP, and makes a note of the 5-tuple. Then, when the router receives an incoming packet, it looks up which connection the packet belongs to, and sends the packet to the appropriate client (with their private IP).

R1's NAT Table	
Inside	Outside
A, Port 50000	S, Port 80
B, Port 60000	S, Port 80

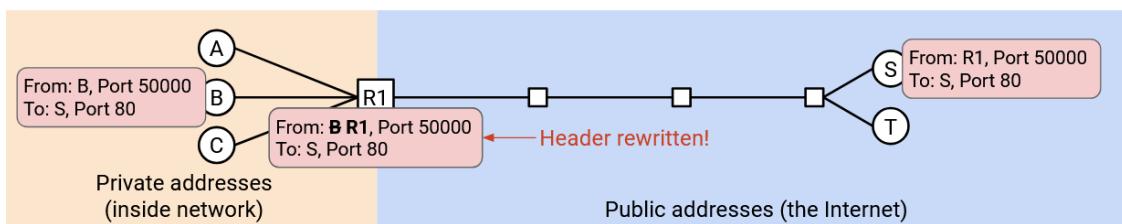


Rewriting Client Port Numbers

We have one last issue: What if, instead of Port 50000 and Port 60000, Alice and Bob both chose the same port number (e.g. Port 50000)?

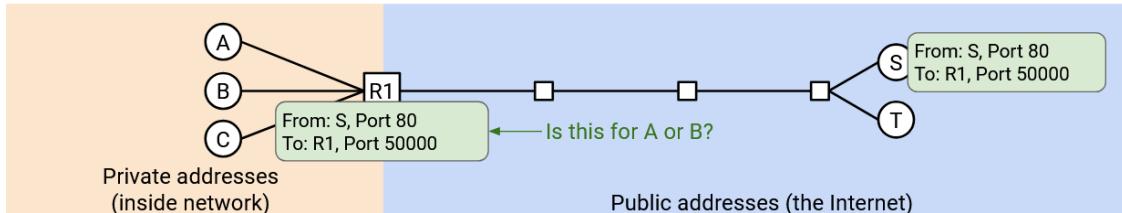
R1's NAT Table	
Inside	Outside
A, Port 50000	S, Port 80
B, Port 50000	S, Port 80

If I get replies from S, Port 80, to me, Port 50000, send them to B.



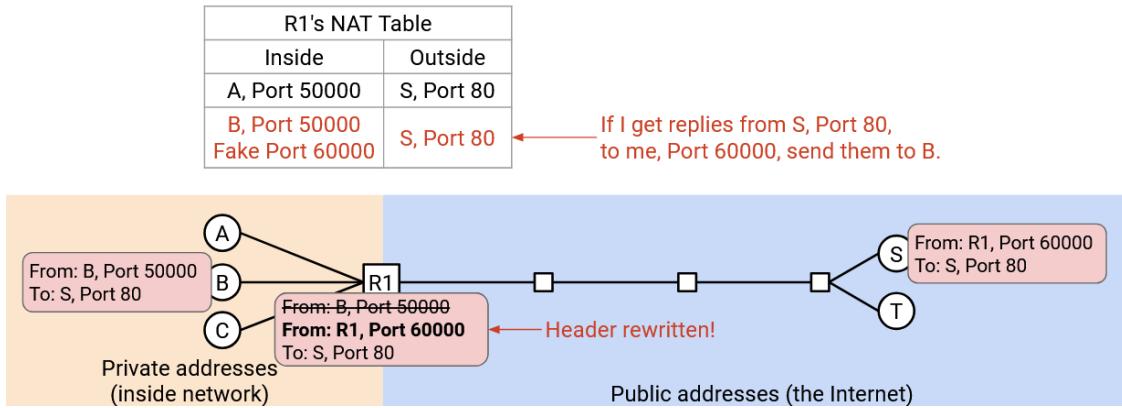
Now, the router remembers two connections: (A Port 50000 to S Port 80), and (B Port 50000 to S Port 80). If the router receives an incoming packet "From: S, Port 80, To: R1 Port 50000," it's ambiguous whether this packet was from A or B's connection.

R1's NAT Table	
Inside	Outside
A, Port 50000	S, Port 80
B, Port 50000	S, Port 80



The last fix we have to make is to also allow the router to rewrite the port number. When Bob sends “From: B, Port 50000, To: S, Port 80,” the router realizes that someone else already has a connection using Port 50000, to S Port 80. Therefore, the router makes up a “fake” port number for Bob (let’s use 60000) and rewrites both the source IP and source port to get: “From: R1, Port 60000, To: S, Port 80.”

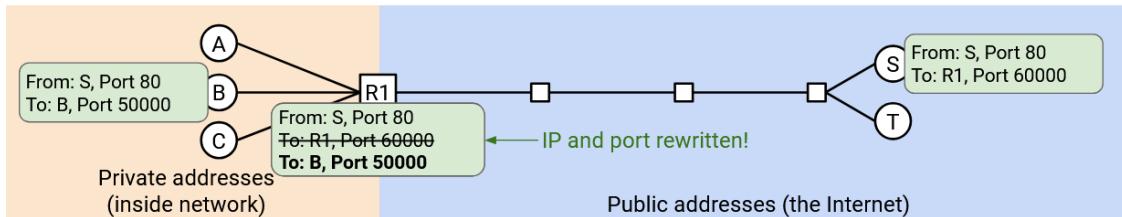
As before, the router remembers the active connection (A Port 50000 to S Port 80), but for Bob, the router additionally notes the fake port number: (B Port 50000, faked as 60000, to S Port 80).



Now, if the router receives an incoming packet “From: S, Port 80, To: R1, Port 50000,” this must be for Alice. By contrast, an incoming packet like “From: S, Port 80, To: R1, Port 60000,” with the fake port number, this must be for Bob.

R1's NAT Table	
Inside	Outside
A, Port 50000	S, Port 80
B, Port 50000	S, Port 80
Fake Port 60000	

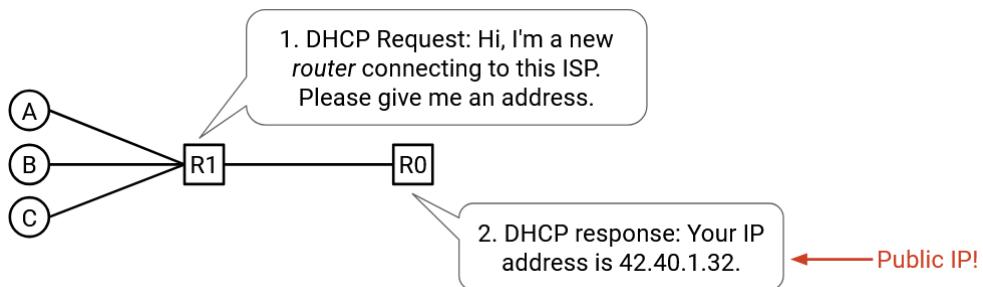
Incoming packets are S, Port 80 in both cases.
 If the inside port is 50000, packet is for A.
 If the inside port is 60000 (fake port), packet is for B.



Note that Bob has no idea that the router is changing his port number. When the router forwards this packet back to Bob, the fake port number must be changed back to the original port number. “From: S, Port 80, To: R1, Port 60000” must be rewritten as “From: S, Port 80, To: R1, Port 60000.” More generally, none of the private clients should need to know or care about their packets getting rewritten. The router should be giving all of them the illusion that they’re sending and receiving packets from their private IP address and whatever ports they choose.

NAT: Implementation

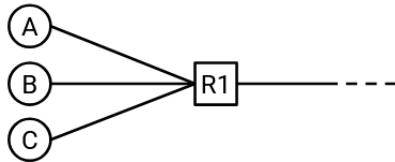
When a home router connects to the ISP for the first time, it can run DHCP to receive an IP address. (Earlier, we talked about hosts running DHCP, but routers can also run DHCP.) The ISP’s DHCP server replies and allocates a single IP address to the home router. This is the single public address that all the hosts in this router’s home network will be sharing.



There are several different modes of NAT. The one we just saw is called **Port Address Translation (PAT)**, and it gives us the ability to introduce the fake port numbers that we saw. The PAT mode requires routers to be aware of Layer 4 protocols, so that they can parse the packets, keep track of connections, and rewrite headers.

PAT is the most complex and widely-used mode of NAT, but simpler modes of NAT also exist for one-to-one address translation. If every host actually had their own IP address, but they sent packets from private addresses, the router could just do a one-to-one translation, mapping 10.0.0.1 (private) to 42.0.2.1 (public), and 10.0.0.2 (private) to 42.0.2.2 (public), and so on. This simpler mode wouldn’t let us conserve IP addresses by hiding multiple hosts behind a single public address, but it can still be useful in other situations.

R1's NAT Table		
Host	Private	Public
A	10.0.0.1	42.0.2.1
B	10.0.0.2	42.0.2.2
C	10.0.0.3	42.0.2.3



Where is NAT Used?

NAT increases the complexity of packet forwarding for the router. The router must now be able to parse the Layer 4 header, in addition to the Layer 3 header. Also, the router must be able to rewrite the Layer 3 and Layer 4 headers. Finally, the router must maintain a connection state table to keep track of all the flows passing through the router. All this functionality increases the number of CPU cycles needed to forward each packet, and also increases the amount of memory needed on the router per flow.

Because NAT increases router complexity, it is performed as close to the edge of the network as possible, in order to limit the number of flows passing through the router. Running NAT on your home router is a good idea, since there aren't too many devices in your home that will send connections through the home router. By contrast, running NAT on a high-performance datacenter router would be a bad idea.

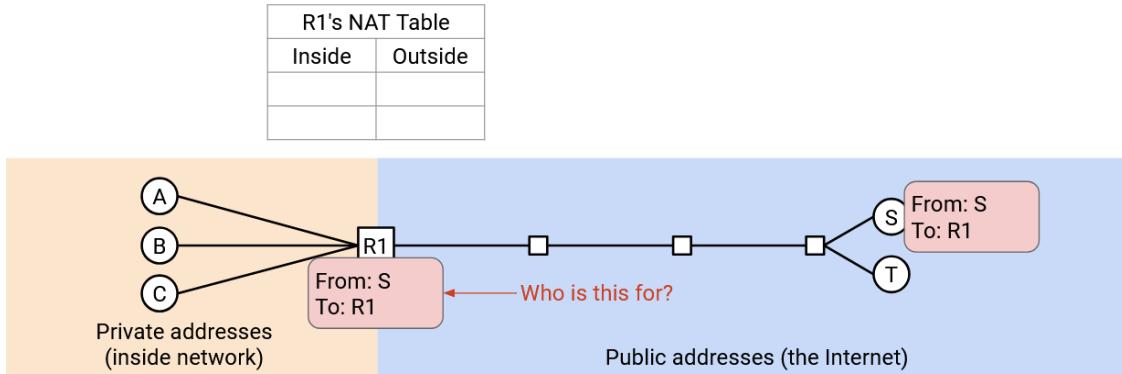
In practice, small-scale NAT is used in almost every personal (home/office) network for IPv4, even today. As IPv4 addresses ran out, ISPs were unable to give one public address to each customer (i.e. each home router). As a result, the ISP network itself also had to run a more complex version of NAT called Carrier Grade NAT (CGNAT). This version of NAT is deployed deeper in the network, and requires routers to keep track of many more connections.

Note that we generally don't use NAT for IPv6, because there are enough IPv6 addresses to assign a unique public one to every computer in the world.

Inbound Connections

So far, we've assumed that connections are always initiated by the client with the private IP address. In other words, the first packet is always outgoing, from client to server. This is consistent with how most home networks operate. When you load a website in your browser, you're the client initiating the connection. It's generally not the case that others are trying to connect to you.

But, what if you were running a server, and you did want people from the outside world to be able to initiate connections to this server? Users from the outside can't send packets to a private IP address. They could try to send packets to the router's IP address, but if the router gets a packet like "From: outside user, To: R1, Port 28," the router has no idea which of the private clients to forward this packet to. This is the very first packet of a new connection, so the router's table has no information about this connection yet.



To allow inbound connections, routers performing NAT need a **port mapping table**. Alice, who is inside the network and only has a private IP address, tells the router: I'm going to run a new server, and listen for requests on Port 28. Now, if the router sees some packet from an outside user to R1, Port 28, the router knows to forward this packet to Alice.

Entries in this port mapping table may need to be specified manually (e.g. Alice manually configuring the router). Dynamic protocols such as UPnP (Universal Plug-n-Play) and NAT-PMP (NAT Port Mapping Protocol) allow for dynamic configuration of open ports. These protocols are sometimes used by applications like online gaming, where inbound connections are needed.

Security Implications of NAT

NAT breaks the end-to-end principle. So far, we've said that with Layer 3, anybody on the Internet can reach anyone else. However, because NAT doesn't allow inbound connections by default, users in a home network, who only have a private IP address and are sharing a public IP address, cannot be reached automatically. They'd need to configure the router before they can accept inbound packets.

NAT has the property that it doesn't allow inbound connections by default. This could be viewed as a security feature, though it's more of a side effect than an intentional design feature. NAT causes inbound connections to be blocked by default, which might be useful for stopping attackers from trying to connect to hosts inside the network. This behavior is actually pretty similar to firewalls (see UC Berkeley CS 161 notes for more information), which also often block inbound connections by default. That said, this is mostly a coincidence, so NAT isn't really implementing a principled security policy, and shouldn't be thought of as a bulletproof defense.

NAT also has the side effect that it can help preserve client privacy. Again, this isn't really an intentional security feature. Because the router rewrites the client's IP address, when the server receives a packet, it doesn't know the original sender's identity (could be Alice, Bob, or Chuck).

By contrast, if we didn't use NAT, the server can learn the exact identity of the sender. Also, if we didn't use NAT and we used IPv6, the server might be able to learn the exact computer the sender is using, since IPv6 addresses are sometimes auto-configured using the MAC address (copying MAC address bits into the IP address). If we were using IPv6 and still wanted client privacy, some alternate solutions like IPv6 temporary/privacy addresses do exist.

TLS: Secure Bytestreams

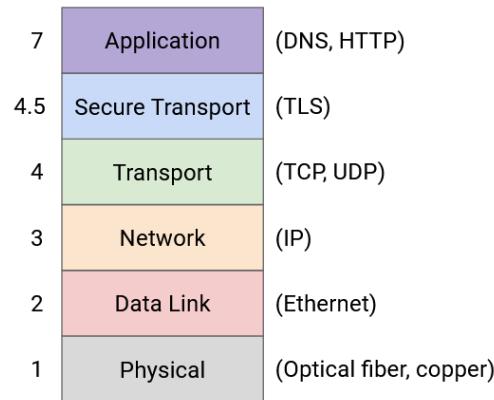
Secure Bytestreams

TCP by itself is insecure against network attackers. Someone on the network (e.g. a malicious router, an attacker sniffing packets on a wire) could read or even modify your TCP packets while they're in transit.

Also, with TCP, you might connect to an attacker instead of the real server. Suppose you want to connect to a bank website, and you do a DNS lookup for `www.bank.com`. The attacker (e.g. someone who hacked into the resolver or a router) changes the DNS response so that it maps `www.bank.com` to the attacker's IP address, 6.6.6.6. Now, when you form a TCP connection to the bank website, you're talking to the attacker. You might end up sending your bank password to the attacker!

To address these security issues, we add a new protocol, **Transport Layer Security (TLS)**, on top of TCP.

TLS can be thought of as a Layer 4.5 protocol, sitting in between TCP and application protocols like HTTP. (We use a weird number like 4.5 because the obsolete Layers 5 and 6 have nothing to do with security.) TLS relies on the bytestream abstraction of TCP, so it doesn't think about individual packets or packet loss/reordering. TLS provides the exact same bytestream abstraction to applications as TCP does, but the bytestream is now secure against network attackers. This is why HTTP and HTTPS are semantically identical protocols. The only difference is that HTTPS runs over the secure bytestream of TLS-over-TCP, while HTTP runs over raw TCP with no TLS.



To distinguish between HTTPS and HTTP, we use Port 80 for HTTP connections, and Port 443 for HTTPS connections. Servers can force users to use HTTPS by replying to all Port 80 requests with a redirect to use Port 443 instead.

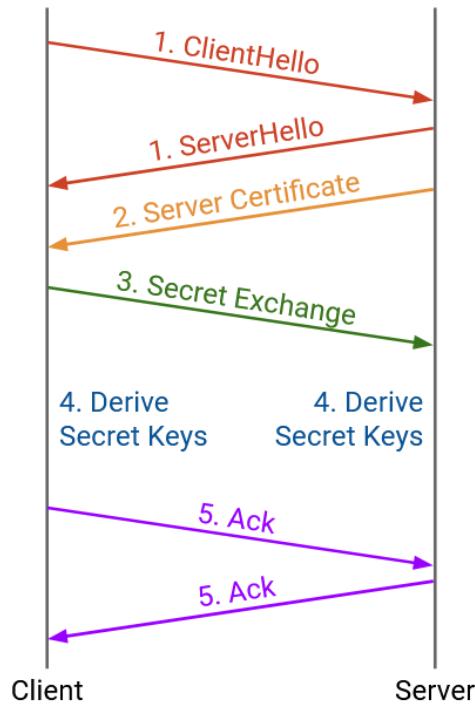
TLS Handshake

At a high level, TLS uses cryptography to encrypt messages sent over the bytestream. TLS also uses other cryptographic protocols (message authentication codes) to prevent attackers from changing messages as they're sent over the network.

In order to encrypt traffic, TLS must start with an additional handshake to exchange keys and verify the identity of the server (e.g. real bank, not someone impersonating the bank).

Because TLS is built on top of TCP, the TCP three-way handshake first proceeds as normal. This creates an (insecure) bytestream, allowing all future messages, including the TLS handshake, to proceed without thinking about individual packets.

The TLS handshake can now proceed:



1. The client and server exchange hellos. The hellos contain random numbers, which ensures that every handshake results in different secret keys. (It would be bad if we used the same key every time, and attackers hacked us and learned that key.) The hellos also allow the client and server to agree on specific cryptographic protocols to use. The client's hello lists all cryptographic schemes the client supports, and the server's hello picks one to use.
2. The server sends a certificate of authenticity. This will allow the client to verify that it's talking to the real server, and not an impersonator. There's a bit of complexity in how the client actually verifies this certificate, which we won't discuss here.
3. The client and server derive a secret that only the two of them know. Since the bytestream is still insecure at this point, they'll need a cryptographic protocol that enables sharing a secret over an insecure channel. We won't discuss the details here, but if you're familiar with RSA public-key encryption (e.g. from CS 70 at UC Berkeley), that's one possible cryptographic scheme to use here. The client encrypts a secret with the server's public key and sends it to the server. Only the server knows the corresponding private key and is able to decrypt the message and learn the secret.
4. The client and server derive secret keys based on the shared secret and the random values from the hellos. Using the secret ensures that attackers can't learn the secret keys. Using the random values ensures

that we derive a different key every time. This derivation is done locally and independently by both the client and server. The secret keys are never actually sent across the network, so the attacker has no chance to learn them.

5. The client and server exchange some acknowledgements to confirm that they derived the same secrets, and nobody tampered with the messages sent over the network so far (since the bytestream is still insecure).

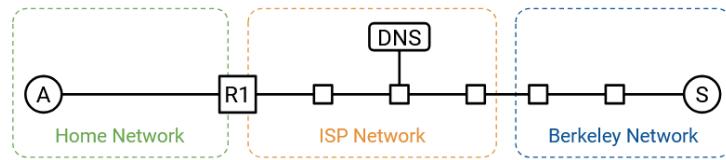
At this point, the handshake is complete, and all future messages are encrypted with the secret key (message authentication codes are also used to prevent tampering). We have now established a secure bytestream on top of the TCP connection, and applications can exchange data on top of our secure bytestream.

End-to-End Connectivity

Motivation

In this section, we'll do a step-by-step walkthrough of what happens when we turn on our computer, plug it into an Ethernet network, and type `www.berkeley.edu` in our web browser. In the process of doing so, we'll see how all the different pieces of the network work together to process the user's request.

We'll assume that we don't need to turn on the Internet from scratch. For example, routers are already actively running routing protocols and have populated their forwarding tables accordingly.

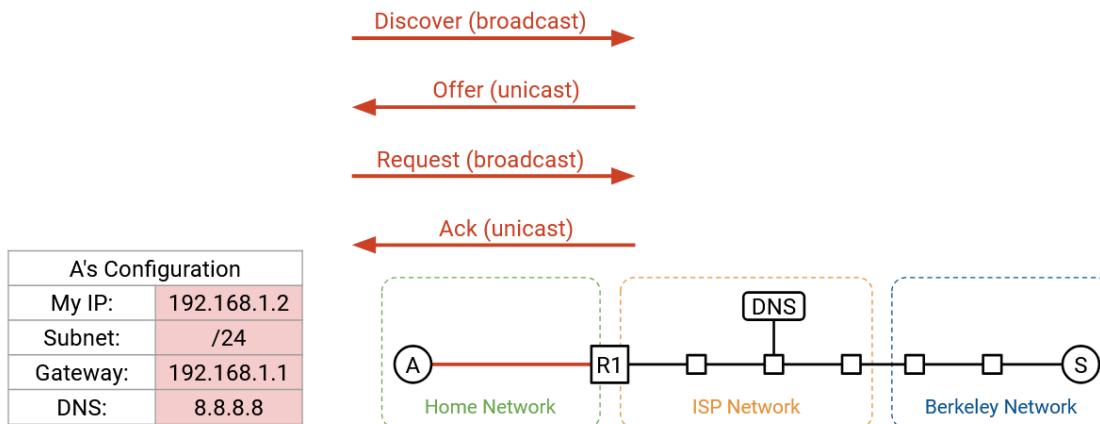


Step 1: DHCP

We turn on our computer and plug it into an Ethernet network. We don't have any information about the network yet, so we broadcast a DHCP request.

We'll assume the home router is the DHCP server, which is common in home networks. The router/server unicasts an offer back to us. The offer contains information about the network: the subnet mask, the IP address of the default gateway, and the IP address of the DNS server. The offer also gives us an IP address we can use.

To complete the DHCP protocol, we send the request message confirming we'd like to use the offered configuration, and the router/server responds with an acknowledgement.



Step 2: Find Router at Layer 2

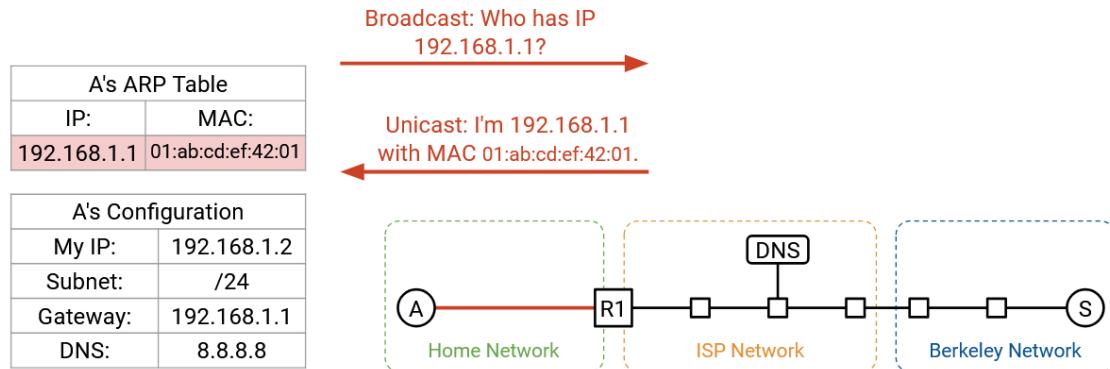
From DHCP, we learned about the IP address of the router, and our forwarding table now says that all non-local packets should be forwarded to this router. We're about to send some packets to the DNS server (to look up the IP address of `www.berkeley.edu`), and to the Berkeley server itself, both of which may be non-local.

Before we can forward IP packets to the router, though, we need to figure out the router's Layer 2 MAC address, so that we can send the packet to the router inside the local network.

First, we can verify that the router's IP address, 192.168.1.1, belongs to the local subnet, 192.168.1.2/24. This tells us that the router is in the local network, and by sending an Ethernet packet to the router's MAC address, we'll reach the router.

To find the router's MAC address, we broadcast an ARP request, asking for the MAC address of 192.168.1.1 (router's IP address). The router hears this request and replies, saying, "I'm 192.168.1.1, and my MAC address is 01:ab:cd:ef:42:01."

We can now cache this IP-to-MAC mapping, and we now know the router's MAC address. As long as this entry stays in the cache, we won't have to make the same ARP request again. All future requests to the outside Internet can be forwarded to the router's MAC address.



Step 3: DNS Lookup

Next, we need to look up IP address of `www.berkeley.edu`. This is all done in the operating system, after the browser code calls something like `getaddrinfo` to trigger the DNS lookup.

From DHCP, we learned the IP address of the DNS server, 8.8.8.8. We also learned that we're in the subnet 192.168.1.2/24. The DNS server isn't in our local network, so we need to forward the DNS packet to the router.

We can now build up our DNS request packet, from the top down.

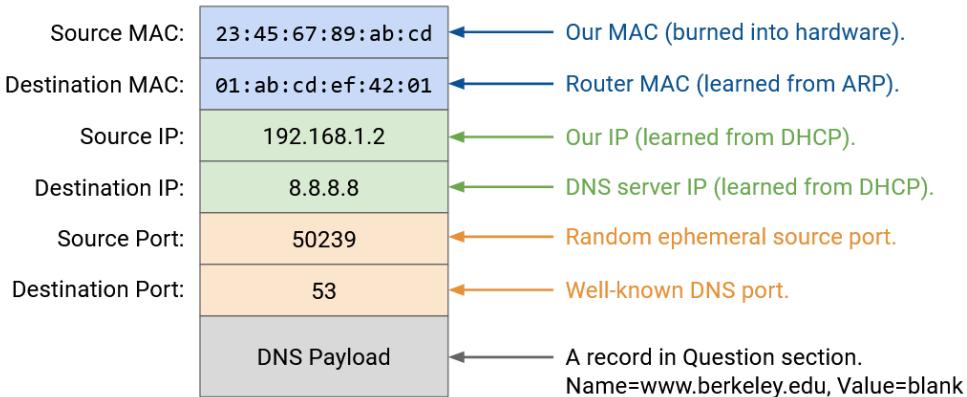
Layer 7: In the Question section, we add a DNS record requesting the A record with `www.berkeley.edu`'s IP address. We add the DNS header with the ID, number of records, and so on.

Layer 4: DNS runs on top of UDP. We pick any random source port, since we're the client. We pick Port 53 for the destination port, since this is where resolvers and name servers listen for DNS queries.

Layer 3: The source IP is our own IP, as assigned by DHCP. The destination IP is 8.8.8.8, the IP address of the DNS server, which we learned from DHCP.

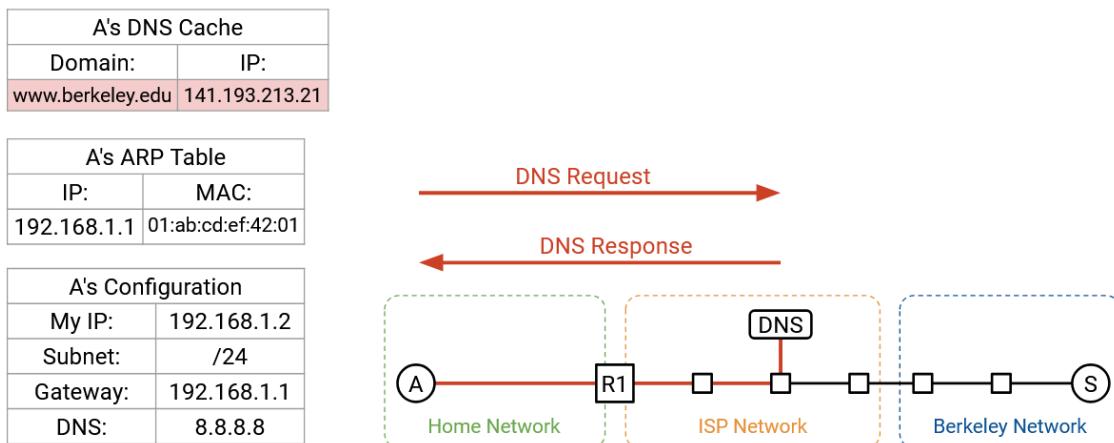
Layer 2: The source MAC is our own MAC address, which is burned into our hardware. The destination MAC is the MAC address of the router (the next hop), which we learned from ARP.

With the packet fully built, we can send the bits along the wire (Layer 1).



When the packet reaches the router, if the network is using NAT, the router might rewrite the UDP/IP headers to translate our private IP address into a public IP address. However, as the end host, we don't have to worry about NAT. The router should be doing all the translation for us, giving us the illusion that we can use our own IP address (from DHCP).

When our packet reaches the recursive resolver at 8.8.8.8, if the resolver doesn't have our answer cached already, it might need to perform some additional lookups and ask the authoritative name servers for the records. Eventually, the recursive resolver finds the answer, and sends the A record back to us. We now have `www.berkeley.edu`'s IP address.

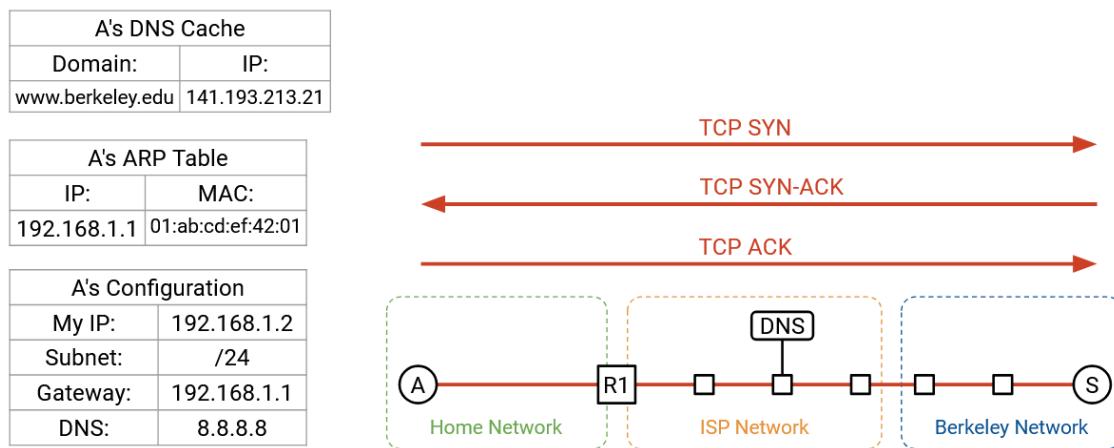


Step 4: Connect to Website

Now that we have `www.berkeley.edu`'s IP address, we can send packets to Berkeley. We're using a web browser, so our goal is to make an HTTP request to this server.

HTTP runs on top of TCP, so we first have to make a TCP handshake to open a connection with the Berkeley server. The browser will call something like `connect` on a particular socket to open this connection, and the operating system (where TCP is running) will perform the handshake and pass packets to and from the browser.

The TCP handshake is performed: We send a SYN, Berkeley sends a SYN-ACK, and we send an ACK. We now have a bytestream between our computer and the Berkeley server.



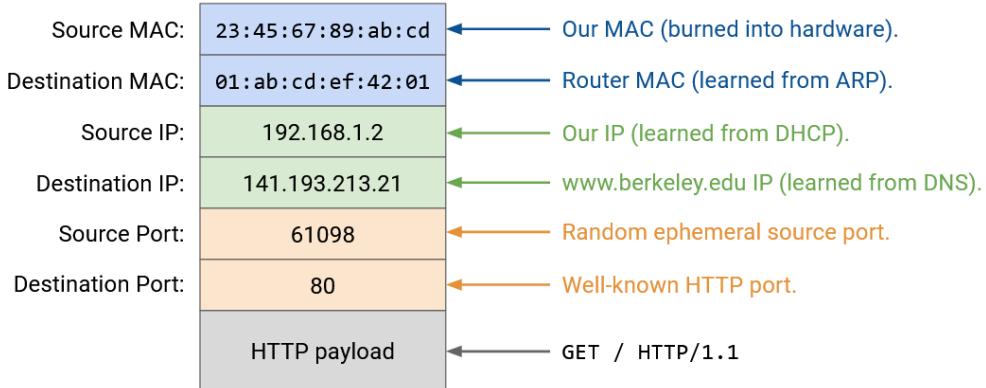
Now, we can build up our HTTP packet, from the top down.

Layer 7: The HTTP method is GET. The resource we want is `/` (the homepage). The version is HTTP/1.1.

Layer 4: HTTP runs on top of TCP. The browser can pick any source port, since it's the client. In general, this port could be manually specified by the application, or the application could specify "Port 0," which is shorthand for asking the operating system to pick a random ephemeral port that's currently unused. (As an aside, thinking back to NAT, allowing applications to manually specify ports is why two users might choose the same source port.) The destination port is 80, the fixed port number for HTTP.

Layer 3: The source IP is our own IP, as assigned by DHCP. The destination IP is 141.193.213.21, the IP address of `www.berkeley.edu` that was returned from our DNS query earlier.

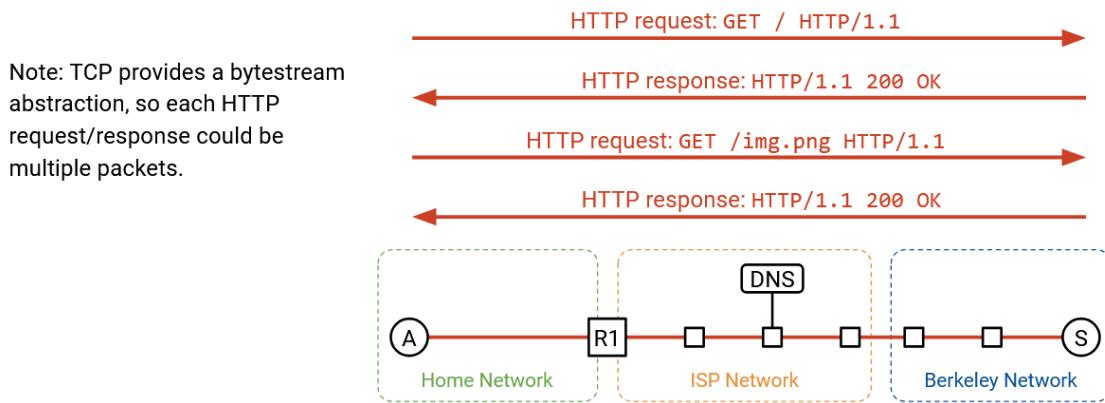
Layer 2: This is the same as our DNS packet earlier. The source MAC is our own (burned into hardware), and the destination MAC is the router's (discovered and cached from ARP).



The HTTP response comes back with status code 200 OK, and the content of the response has the HTML code of the website. The browser calls `read` on the socket to fetch the bytes of the HTTP payload, with the status code and the response, and processes them accordingly.

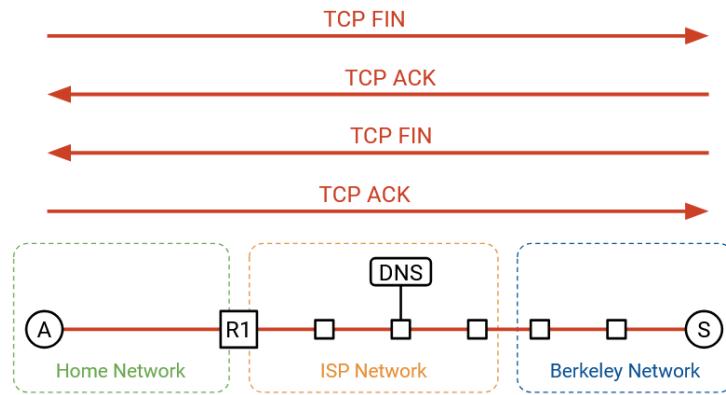
Within the bytestream, HTTP can add some delimiter like a newline character to denote the end of a request or response. Also, HTTP headers like `Content-Length` can specify the length of the payload. This also allows the browser to allocate enough memory to receive the response.

The HTTP response that comes back might trigger further requests. If the HTML in the response has some syntax like ``, this tells the browser to make another HTTP request to fetch the `/logo.png` resource. Or, the user might click a link on the website like `www.berkeley.edu/about.html`, which would also trigger another HTTP request to the same server.



Recall that multiple HTTP requests to the same server can be pipelined across the same TCP connection for efficiency, so we can keep the TCP connection open and keep using it for subsequent HTTP requests and responses.

Eventually, after some pipelining, the client or server chooses to close the connection. The normal teardown handshake occurs, where each side sends a FIN, and both FIN packets are acked. We're all done!



Note that the HTTP requests/responses are not necessarily contained in a single packet. HTTP is built on top of the TCP bytestream, so a single HTTP request or response could get split up across multiple TCP/IP packets, where each packet has the same headers at Layers 1-3, and the Layer 4 headers differ in sequence number. There's only a single header for the entire HTTP request/response, even if the request/response is split across packets. With HTTP, there's no longer a one-to-one correlation from one request/response to one packet.

Sockets

If you're a user visiting a website in your browser, you don't need to write any code to run the application (HTTP) over the Internet. However, if you were a programmer writing your own application, you probably need to write some code to interact with the network.

The **socket** abstraction gives programmers a convenient way to interact with the network. The socket abstraction exists entirely in software, and there are five basic operations that programmers can run:

We can **create** a new socket, corresponding to a new connection. In an object-oriented language like Java, this could be a constructor call.

We can call **connect**, which initiates a TCP connection to some remote machine. This is useful if we're the client in a client-server connection.

We can call **listen** on a specific port. This does not start a connection, but allows others to initiate a connection with us on the specified port.

Once the connection is open, we can call **write** to send some bytes on the connection. We can also call **read**, which takes one argument N, to read N bytes from the connection.

This socket abstraction gives programmers a way to write applications without thinking about lower-level abstractions like TCP, IP, or Ethernet.

From the operating system perspective, each socket is associated with a Layer 4 port number. All packets to and from a single socket have the same port number, and the operating system can use the port number to de-multiplex and send packets to the correct socket.

Layers in the OS

In hardware, Layers 1 and 2 are implemented on your computer's hardware Network Interface Card (NIC). Layers 3 and 4 are implemented in the networking stack in the operating system. The Layer 7 applications are implemented in software. The benefit of putting Layers 3 and 4 in the OS is, the applications don't have to worry about re-implementing them every time.

With this division of labor, the application just needs to think about data. The NIC just needs to think about packets. The network stack in the OS translates between connections and packets.

Viewing Packets

Tools like tshark and wireshark exist if you want to look at packets being sent across the network. These tools are useful when debugging the networking part of your code.

In your browser, you can also use the Network tab of the inspect element console to view data being sent and received.

If you actually looked the raw packets being sent across the network, you'll see some real-world complexities that we didn't cover in our end-to-end walkthrough. For example, packets might be encrypted and sent over TLS. Also, if we're using HTTP/3.0, packets might be sent over QUIC (the UDP variant optimized for HTTP) instead of TCP.

Revisiting Layering

The full end-to-end picture lets us see why layering is a useful principle for building the network. We were able to solve specific problems at a single layer, without thinking about all the layers at the same time.

In fact, we haven't discussed Layer 1 at all in this class. We didn't talk about the electrical engineering or physics required to send signals across a wire. However, we were still able to build the other layers on top of Layer 1, without knowing exactly how Layer 1 works.

In this class, we've discussed HTTP as the predominant Layer 7 protocol, but HTTP is a relatively simple protocol. It's possible that multiple applications want to build the same complicated functionality on top of HTTP, but they don't want to each write the code for that functionality independently. To support this, we can actually build further protocols on top of HTTP, so that programmers don't always have to start from scratch with HTTP.

One example of a protocol above Layer 7 is a remote procedure call (RPC) library. This allows a programmer to write some code, where some of the functions actually execute on a different computer elsewhere in the network. It would be annoying if everyone had to write RPC on top of HTTP from scratch, so instead, libraries like Apache Thrift and gRPC exist to abstract even more details away from the programmer.

```

func main() {
    flag.Parse()
    // Set up a connection to the server.
    conn, err := grpc.Dial(*addr, grpc.WithTransportCredentials(insecure.NewCredentials()))
    if err != nil { log.Fatalf("did not connect: %v", err) }
    defer conn.Close()
    c := pb.NewGreeterClient(conn)

    // Contact the server and print out its response.
    ctx, cancel := context.WithTimeout(context.Background(), time.Second)
    defer cancel()
    r, err := c.SayHello(ctx, &pb>HelloRequest{Name: *name})
    if err != nil { log.Fatalf("could not greet: %v", err) }
    log.Printf("Greeting: %s", r.GetMessage())
}

```

Programmer can ignore everything at lower layers, and focus on their own application logic.

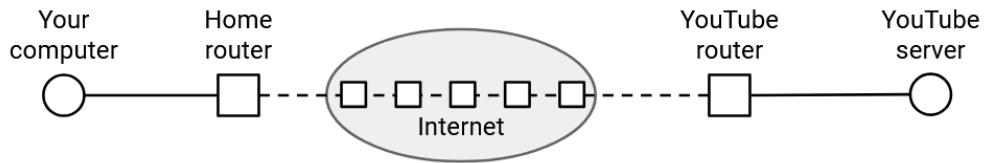
Here's an example of some network code that a programmer might write. It programs a client to say hello to some remote server.

Notice that all of the network protocols we discussed are completely hidden behind the two lines of calls to networking libraries. The programmer didn't have to think about HTTP, TCP, IP, Ethernet, ARP, DHCP, or any other lower-level protocol. It's still useful to know about these protocols if they go wrong, and understanding the protocols can help you optimize your code for specific protocols, but ultimately, layering is a very powerful tool for abstraction.

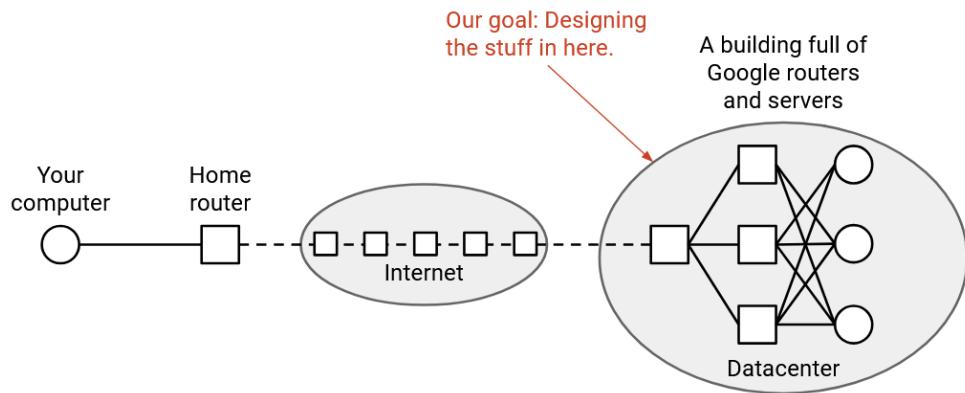
Datacenter Topology

What is a Datacenter?

So far, in our model of the Internet, we've shown end hosts sending packets to each other. The end host might be a client machine (e.g. your local computer), or a server (e.g. YouTube). But, is YouTube really a single machine on the Internet serving videos to the entire world?



In reality, YouTube is an entire building of interconnected machines, working together to serve videos to clients. All these machines are in the same local network, and can communicate with each other to fulfill requests (e.g. if the video you requested is stored across different machines).



Recall that in the network-of-network model of the Internet, each operator is free to manage their local network however they want. In this section, we'll focus on local networks dedicated to connecting servers inside a datacenter (as opposed to users like your personal computer). We'll talk about challenges unique to these local networks, and specialized solutions to networking problems (e.g. congestion control and routing) that are specifically designed to work well in datacenter contexts.

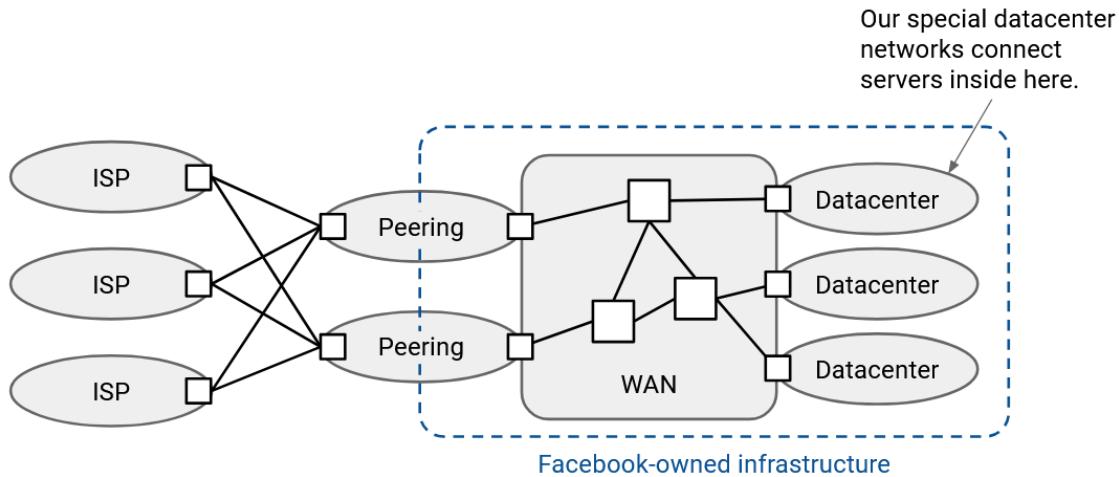
In real life, a datacenter is housed in one physical location, often on dedicated properties. In addition to computing infrastructure (e.g. servers), datacenters also need supporting infrastructure like cooling systems and power supplies, though we'll be focusing on the local network that connects the servers.

Datacenters serve applications (e.g. YouTube videos, Google search results, etc.). This is the infrastructure for the end hosts that you might want to talk to. Note that this is different from Internet infrastructure we've seen so far. Previously, we saw carrier hotels, buildings where lots of networks (owned by different

companies) connect to each other with heavy-duty routers. This is the infrastructure for routers forwarding your packets to various destinations, but applications are usually not hosted in carrier hotels.

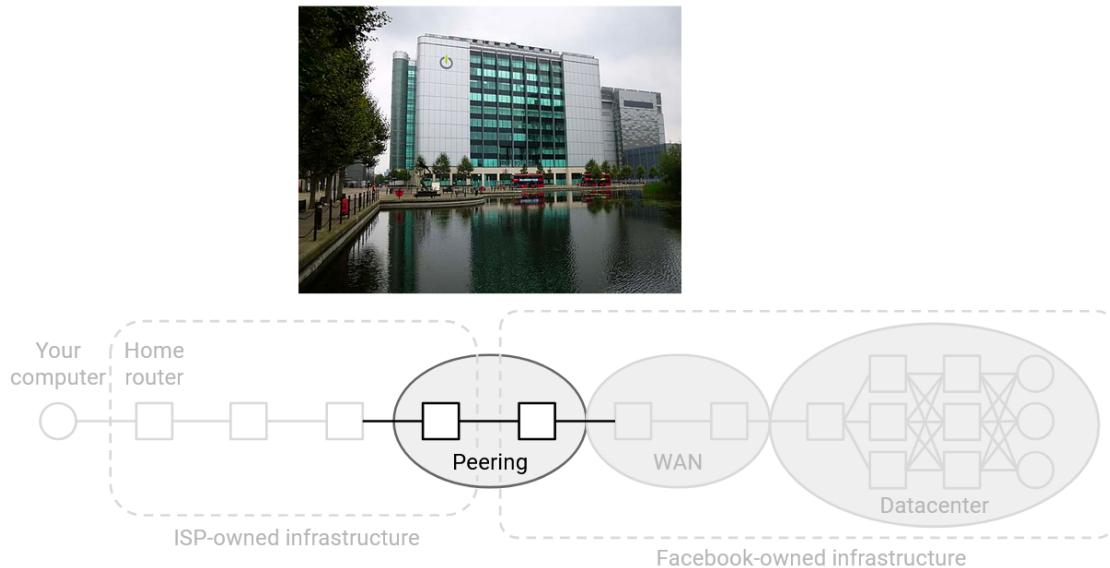
A datacenter is usually owned by a single organization (e.g. Google, Amazon), and that organization could host many different applications (e.g. Gmail, YouTube, etc.) in a single datacenter. This means that the organization has control over all the network infrastructure inside the datacenter's local network.

Our focus is on modern hyperscale datacenters, operated by tech giants like Google and Amazon. The large scale introduces some unique challenges, but the concepts we'll see also work at smaller scales.



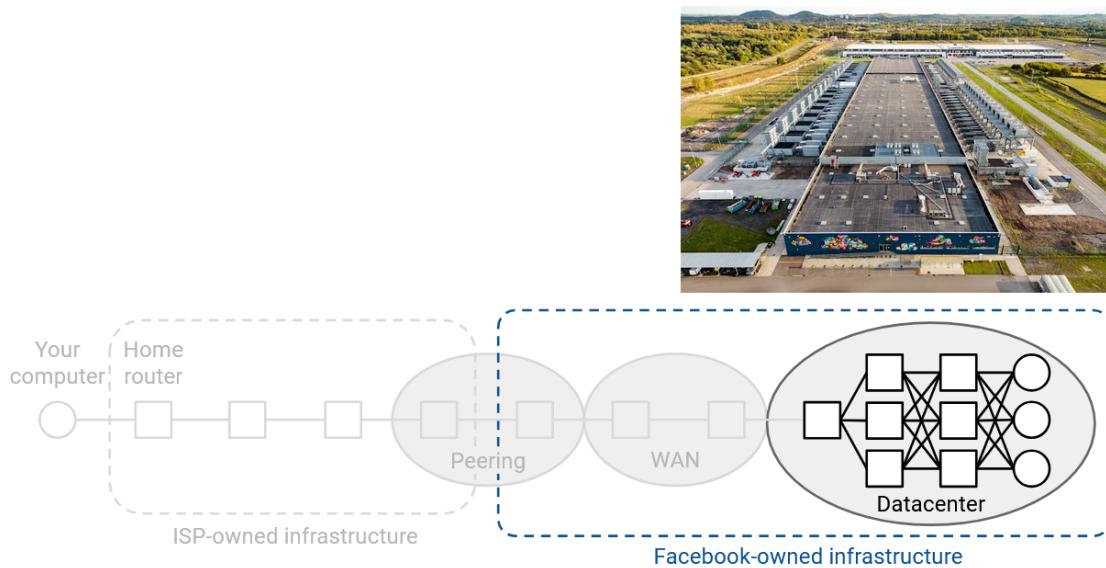
This map shows the wide area network (WAN) of all the networks owned by a tech giant like Google.

The peering locations connect Google to the rest of the Internet. These mainly consist of Google-operated routers that connect to other autonomous systems.



In addition to peering locations, Google also operates many datacenters. Applications in datacenters can

communicate with the rest of the Internet via the peering locations. The datacenters and peering locations are all connected through Google-managed routers and links in Google's wide area network.



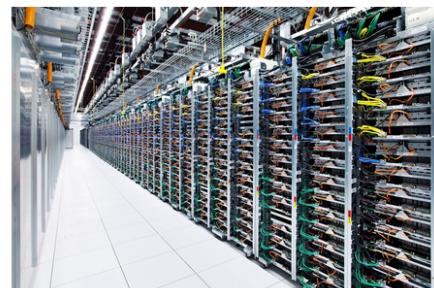
Datacenters and peering locations optimize for different performance goals, so they're often physically located in different places.

Peering locations care about being physically close to other companies and networks. As a result, carrier hotels are often located in cities to be physically closer to customers and other companies.

By contrast, datacenters care less about being close to other companies, and instead prioritize requirements like physical space, power, and cooling. As a result, datacenters are often located in less-populated areas, sometimes with a nearby river (for cooling) or power station (datacenters might need hundreds of times more power than peering locations).



Infrastructure for cooling inside a Google datacenter.



Ultimately, a datacenter is just a building with a ton of servers. Our job is to build the network that connects the servers.

Why is the Datacenter Different?

What makes a datacenter's local network different from general-purpose (wide area) networks on the rest of the Internet?

The datacenter network is run by a single organization, which gives us more control over the network and hosts. Unlike in the general-purpose Internet, we can run our own custom hardware or software, and we can enforce that every machine follows the same custom protocol.

Datacenters are often homogeneous, where every server and switch is built and operated exactly the same. Unlike in the general-purpose Internet, we don't have to consider some links being wireless, and others being wired. In the general-purpose Internet, some computers might be newer than others, but in a datacenter, every computer is usually part of the same generation, and the entire datacenter is upgraded at the same time.

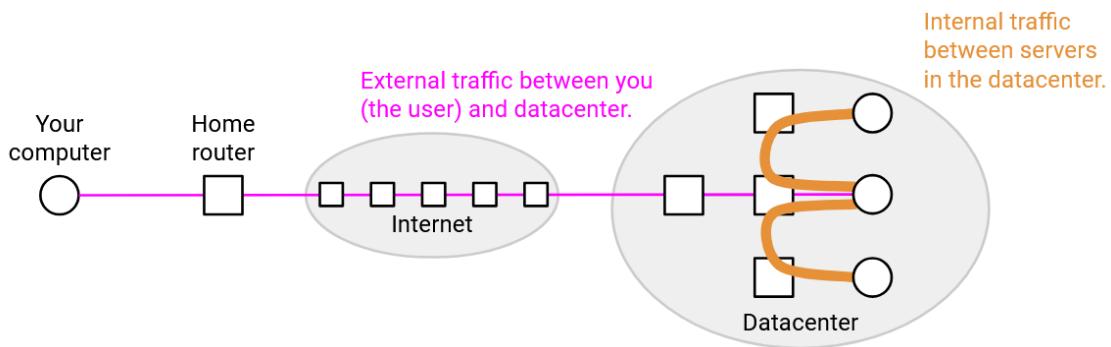
The datacenter network exists in a single physical location, so we don't have to think about long-distance links like undersea cables. Within that single location, we have to support extremely high bandwidth.

Datacenter Traffic Patterns

When you make a request to a datacenter application, your packet travels across routers in the general-purpose Internet, eventually reaching Google-operated router. That router forwards your packet to one of the datacenter's edge routers, which then forwards your packet to some individual server in the datacenter.

This one server probably doesn't have all the information to process your request. For example, if you requested a Facebook feed, different servers might need to work together to combine ads, photos, posts, etc. It wouldn't be practical if every server had to know everything about Facebook to process your request by itself.

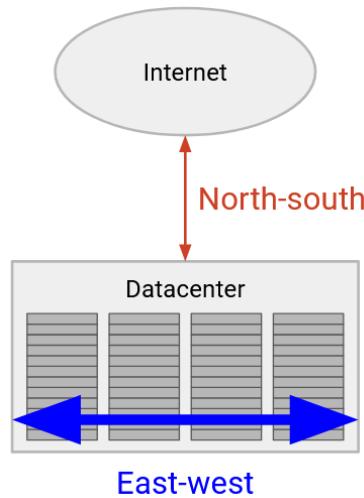
In order for the different servers to coordinate, the first server triggers many backend requests to collect all the information needed in your request. A single user request could trigger hundreds of backend requests (521 on average, per a 2013 Facebook paper) before the response can be sent back to the user. In general, there's significantly more backend traffic between servers, and the external traffic with the user is very small in comparison.



Most modern applications are dominated by internal traffic between machines. For example, if you run a distributed program like mapreduce, the different servers need to communicate to each other to collectively

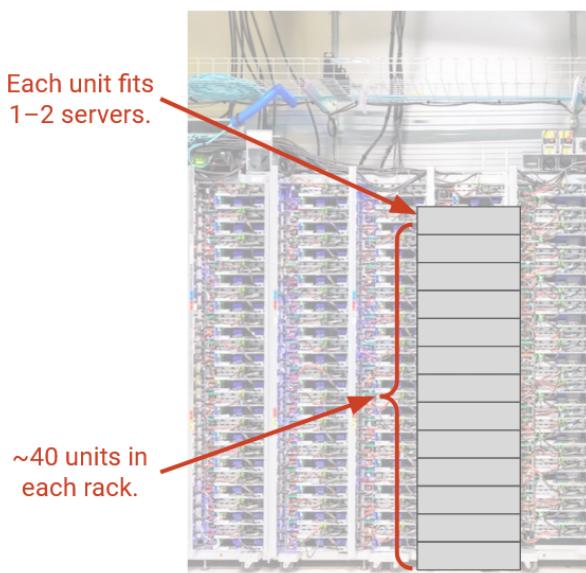
solve your large query. Some applications might even have no user-facing network traffic at all. For example, Google might run periodic backups, which requires servers communicating, but produces no visible result for the end user.

Connections that go outside the network (e.g. to end users or other datacenters) are described as **north-south** traffic. By contrast, connections between machines inside the network are described as **east-west** traffic. East-west traffic is several orders of magnitude larger than north-south traffic, and the volume of east-west traffic is increasing in recent years (e.g. with the growth of machine learning).



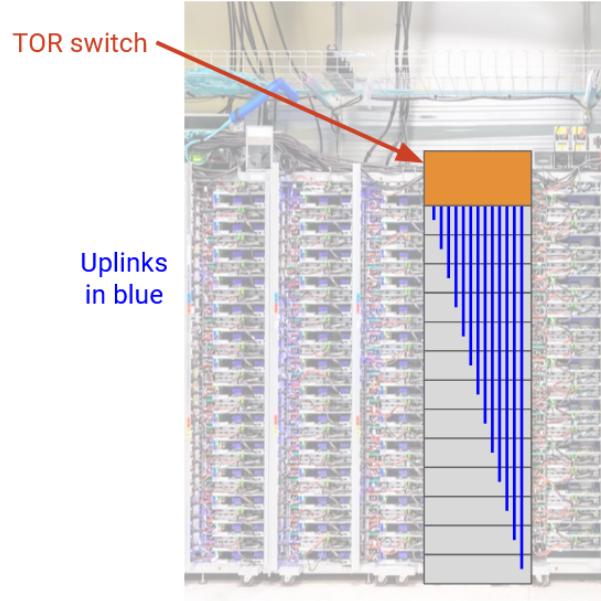
Racks

A datacenter fundamentally consists of many servers. The servers are organized in physical racks, where each rack has 40-48 rack units (slots), and each rack unit can fit 1-2 servers.

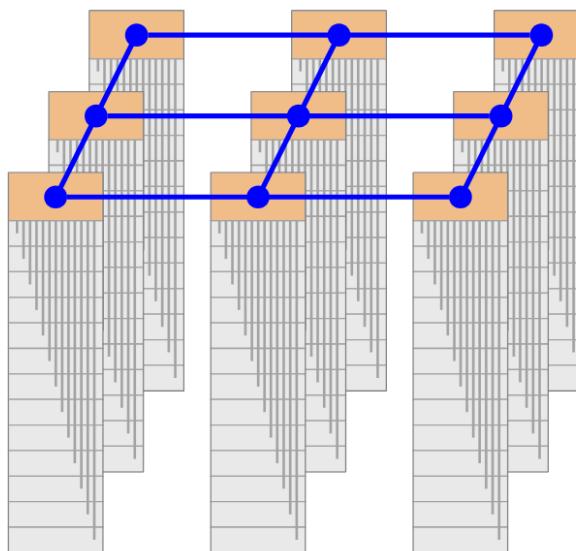


We'd like all the servers in the datacenter to be able to communicate with each other, so we need to build a network to connect them all. What does this network look like? How do we efficiently install links and switches to meet our requirements?

First, we can connect all the servers within a single rack. Each rack has a single switch called a **top-of-rack (TOR) switch**, and every server in the rack has a link (called an **access link** or **uplink**) connecting to that switch. The TOR is a relatively small router, with a single forwarding chip, and physical ports connecting to all the servers on the rack. Each server uplink typically has a capacity of around 100 Gbps.

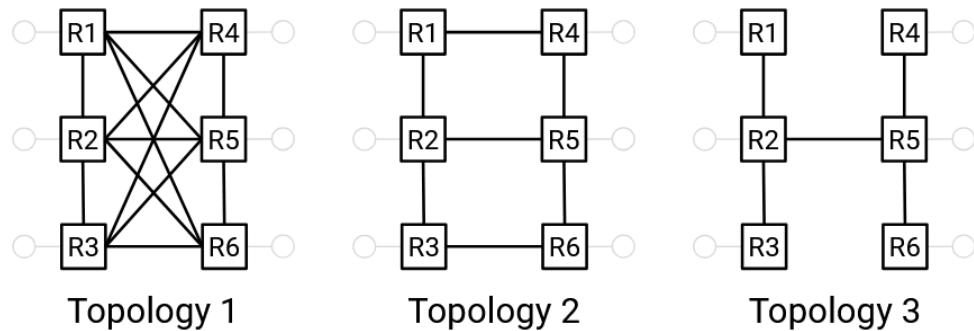


Next, we have to think about how to connect the racks together. Ideally, we'd like every server to talk to every other server at their full line rate (i.e. using the entire uplink bandwidth).



Bisection Bandwidth

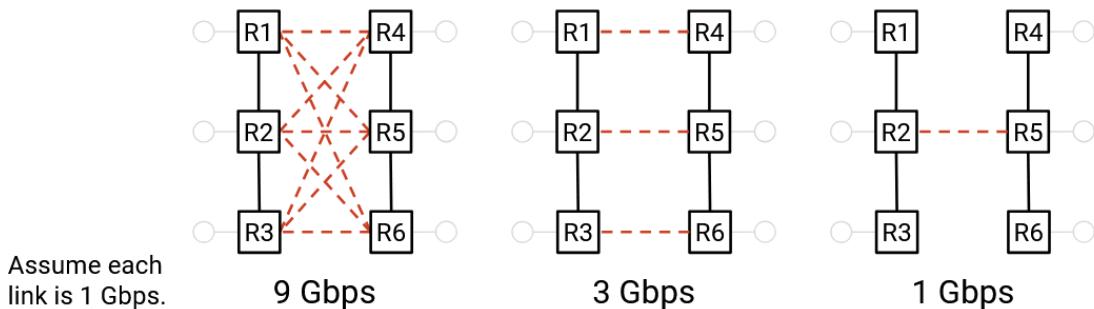
Before thinking about how to connect racks, let's develop a metric for how connected a set of computers are.



Intuitively, even though all three networks are fully connected, the left network is the most connected, the middle network is less connected, and the right network is the least connected. For example, the left and middle networks could support 1-4 and 3-6 simultaneously communicating at full line rate, while the right network cannot.

One way to argue that the left network is more connected is to say: We have to cut more links to disconnect the network. This indicates that there are lots of redundant links, which allows us to run many simultaneous high-bandwidth connections. Similarly, one way to argue that the right network is less connected is to say: We only have to cut the 2-5 link to connect the network, which indicates the existence of a bottleneck that prevents simultaneous high-bandwidth connections.

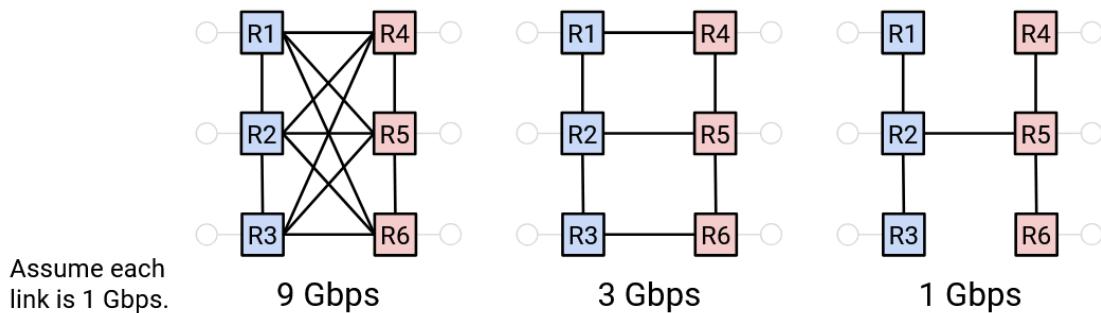
Bisection bandwidth is a way to quantify how connected a network is. To compute bisection bandwidth, we compute the number of links we need to remove in order to partition the network into two disconnected halves of equal size. The bisection bandwidth is the sum of the bandwidths on the links that we cut.



In the rightmost structure, we only need to remove one link to partition the network, so the bisection bandwidth is just that one link. By contrast, in the leftmost structure, we need to remove 9 links to partition the network, so the bisection bandwidth is the combined bandwidth of all 9 links.

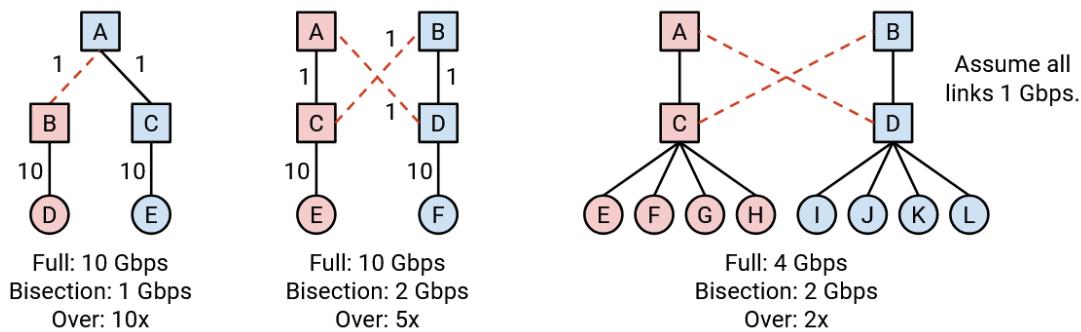
An equivalent way of defining bisection bandwidth is: We divide the network into two halves, and each node in one half wants to simultaneously send data to a corresponding node in the other half. Among

all possible partitions of nodes, what is the minimum bandwidth that the nodes can collectively send at? Considering the worst case (minimum bandwidth) forces us to think about bottlenecks.



The most-connected network has full bisection bandwidth. This means that there are no bottlenecks, and no matter how you assign nodes to partitions, all nodes in one partition can communicate simultaneously with all nodes in the other partition at full rate. If there are N nodes, and all $N/2$ nodes in the left partition are sending data at full rate R , then the full bisection bandwidth is $N/2$ times R .

Oversubscription is a measure of how far from the full bisection bandwidth we are, or equivalently, how overloaded the bottleneck part of the network is. It's a ratio of the bisection bandwidth to the full bisection bandwidth (the bandwidth if all hosts sent at full rate).

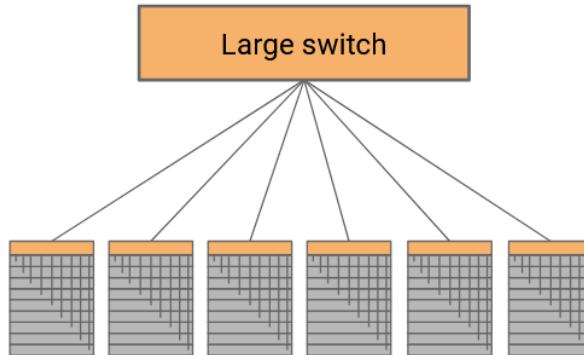


In the rightmost example, assuming all links are 1 Gbps, then the bisection bandwidth is 2 Gbps (to split the left four hosts with the right four hosts). The full bisection bandwidth, achieved when all four left hosts were simultaneously sending data, is 4 Gbps. Therefore, the ratio 2/4 tells us that the hosts can only send at 50% of their full rate. In other words, our network is 2x oversubscribed, because if the hosts all sent at full rate, the bottleneck links would be 2x overloaded (4 Gbps on 2 Gbps of links).

Datacenter Topology

We've now defined bisection bandwidth, a measure of connectedness that's a function of the network topology. In a datacenter, we can choose our topology (e.g. choose where to install cables). What topology should we build to maximize bisection bandwidth?

One possible approach is to connect every rack to a giant cross-bar switch. All the racks on the left side can simultaneously send data at full rate into the switch, which forwards all that data to the right side at full rate. This would allow us to achieve full bisection bandwidth.

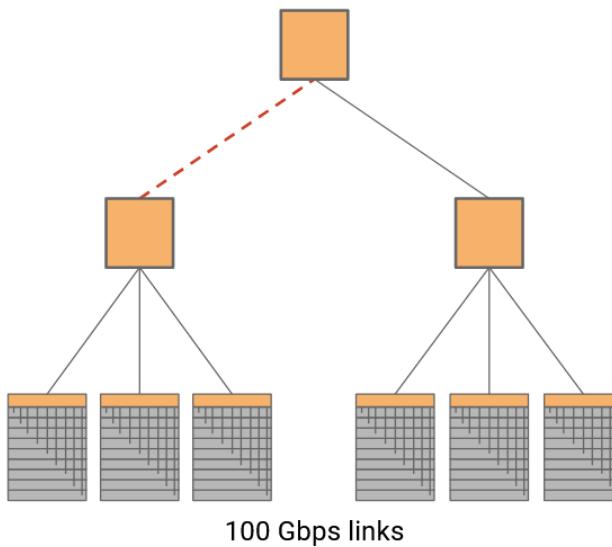


What are some problems with this approach? The switch will need one physical port for every rack (potentially up to 2500 ports). We sometimes refer to the number of external ports as the **radix** of the switch, so this switch would need a large radix. Also, this switch would need to have enormous capacity (potentially petabits per second) to support all the racks. Unsurprisingly, this switch is impractical to build (even if we could, it would be prohibitively expensive).

Fun fact: In the 2000s, Google tried asking switch vendors to build a 10,000-port switch. The vendors declined, saying it's not possible to build this, and even if we could, nobody is asking for this except you (so there's no profit to be made in building it).

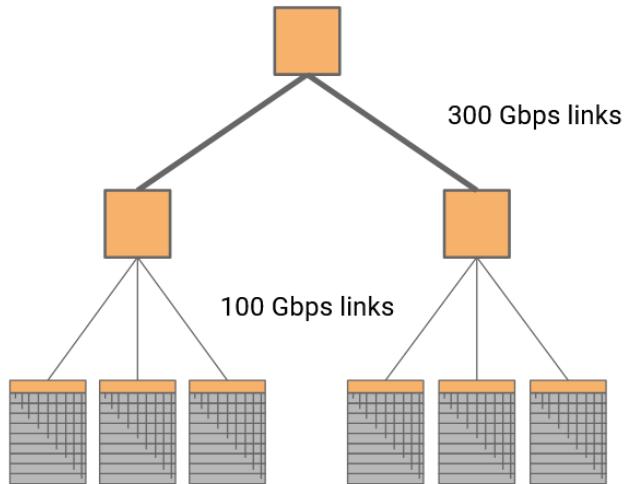
Another problem is that this switch is a single point of failure, and the entire datacenter network stops working if this switch breaks.

Another possible approach is to arrange switches in a tree topology. This can help us reduce the radix and the bandwidth of each link.



What are some problems with this approach? The bisection bandwidth is lower. A single link is the bottleneck between the two halves of the tree.

To increase bisection bandwidth, we could install higher-bandwidth links at higher layers.

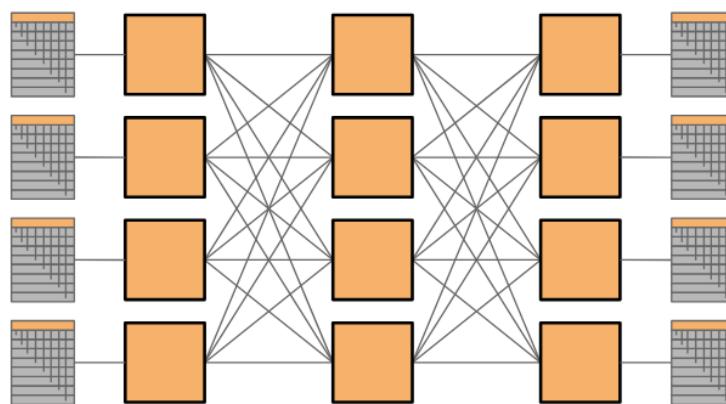


In this case, if the four lower links are 100 Gbps, and the two higher links are 300 Gbps, then we've removed the bottleneck and restored full bisection bandwidth.

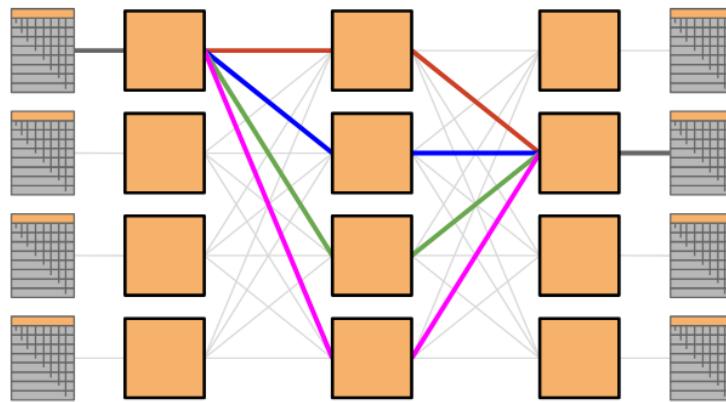
This topology can be used, although we still haven't solved the problem where the top switch is expensive and scales poorly.

Clos Networks

So far, we've tried building networks using custom-built switches, potentially with very high bandwidth or radix. These switches are still expensive to build. Could we instead design a topology that gives high bisection bandwidth, using cheap commodity elements? In particular, we'd like to use a large number of cheap off-the-shelf switches, where all the switches have the same number of ports, each switch has a low number of ports, and all link speeds are the same.



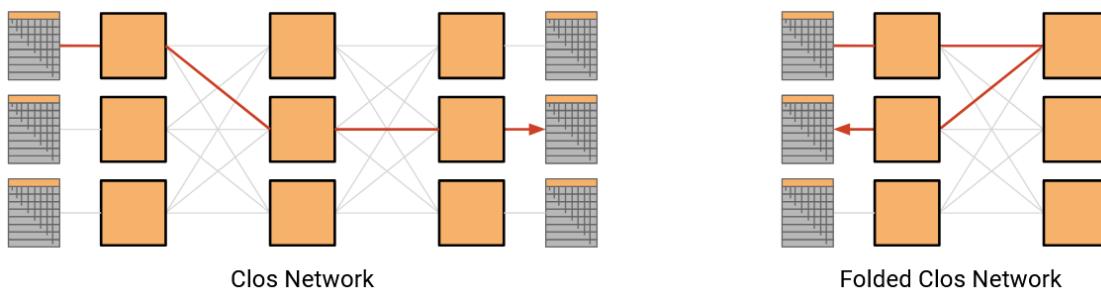
A **Clos network** achieves high bandwidth with commodity parts by introducing a huge number of paths between nodes in the network. Because there are so many links and paths through the network, we can achieve high bisection bandwidth by having each node send data along a different path.



Unlike custom-built switches, where we scaled the network by building a bigger switch, we can scale Clos networks by simply adding more of the same switches. This solution is cost-effective and scalable!

Clos networks have been used in other applications too, and are named for their inventor (Charles Clos, 1952).

In a classic Clos network, we'd have all the racks on the left send data to the racks on the right. In datacenters, racks can both send and receive data, so instead of having a separate layer of senders and recipients, we can have a single layer with all the racks (acting as either sender or recipient). Then, data travels along one of the many paths deeper into the network, and then back out to reach the recipient. This result is called a **folded Clos network**, because we've “folded” the sender and recipient layers into one.



Fat-Tree Clos Topology

The fat-tree topology has low radix per switch, and achieves full bisection bandwidth. However, the switch at the top of the tree is expensive, scales poorly, and still represents a single point of failure.

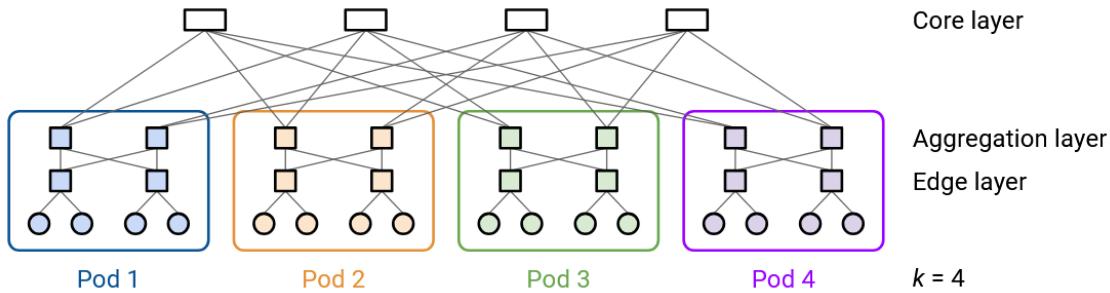
The Clos topology allows us to use commodity switches to scale up our network. If we combine the Clos topology with the fat-tree topology, we can build a scalable topology out of commodity switches!

The topology presented here was introduced in a 2008 SIGCOMM paper titled “A Scalable, Commodity Data Center Network Architecture” (Mohammad Al-Fares, Alexander Loukissas, Amin Vahdat).

In a k -ary fat tree, we create k pods. Each pod has k switches.

Within a pod, $k/2$ switches are in the upper aggregation layer, and the other $k/2$ switches are in the lower edge layer.

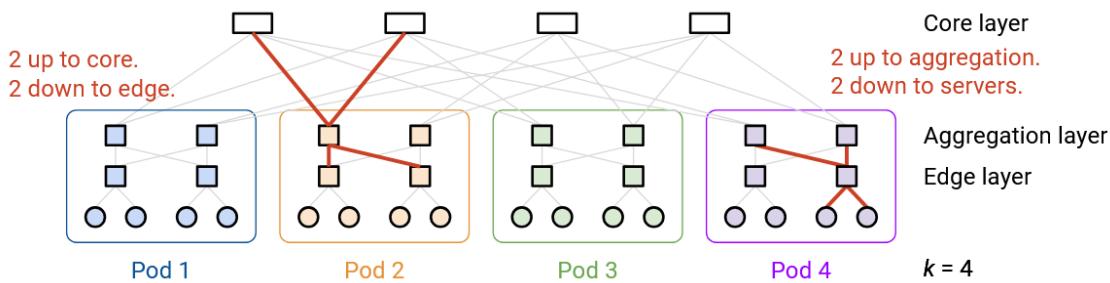
(Note: This topology is defined for even k , so that we can split up the switches evenly between the aggregation layer and edge layer).



Each switch in the pod has k links. Half of the links ($k/2$) connect upwards, and the other half ($k/2$) connect downwards.

Consider a switch in the upper aggregation layer. Half ($k/2$) of its links connect up to the core layer (which connects the pods, discussed more below). The other half ($k/2$) of its links connect downwards to the $k/2$ switches in the edge layer.

Similarly, consider a switch in the lower edge layer. Half ($k/2$) of its links connect upwards to the $k/2$ switches in the aggregation layer. The other half ($k/2$) of its links connect downwards to $k/2$ hosts in this pod.

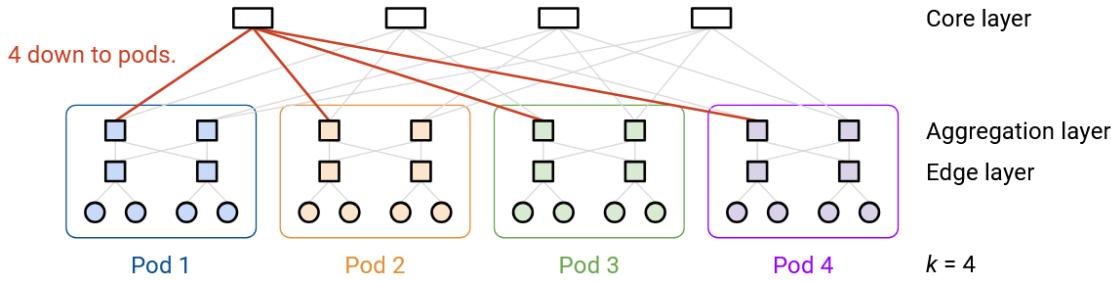


Next, let's look at the core layer, which connects the pods together. Each core switch has k links, connecting to each of the k pods.

There are $(k/2)^2$ core switches. How did we derive this number? There are k pods, and each pod has $k/2$ switches in the upper aggregation layer, for a total of $k^2/2$ switches in the aggregation layer. Each aggregation-layer switch has $k/2$ links pointing upwards, for a total of $k^2/2 \times k/2 = k^3/4$ links pointing upwards. This means that the core layer will need to have a total of $k^3/4$ links pointing downwards, to match the number of upwards links from the aggregation layer.

Each core layer switch has k links pointing downwards, so we need $k^2/4$ core layer switches (each with k links) to create $k^3/4$ links pointing towards. This allows the number of links up from the aggregation layer to match the number of links down from the core layer.

We can also compute that there are $(k/2)^2$ hosts per pod in this topology. How did we derive this number? There are $k/2$ switches at the edge layer of each pod. Each edge-layer switch has $k/2$ downwards links to hosts, for a total of $k/2 \times k/2 = (k/2)^2$ hosts per pod. Note that each host is only connected to one edge-layer switch (a host is not connected to multiple switches in this topology). Since there are k pods in total, we can also deduce that there are $(k/2)^2 \times k$ hosts in total in this topology.



$k = 4$, the smallest example, is unfortunately a little confusing because some of the numbers coincidentally end up the same (e.g. $(k/2)^2 = k = 4$). For a clearer example, we can look at $k = 6$.

Each pod has $k = 6$ switches. $k/2 = 3$ switches are in the upper aggregation layer, and $k/2 = 3$ switches are in the lower edge layer.

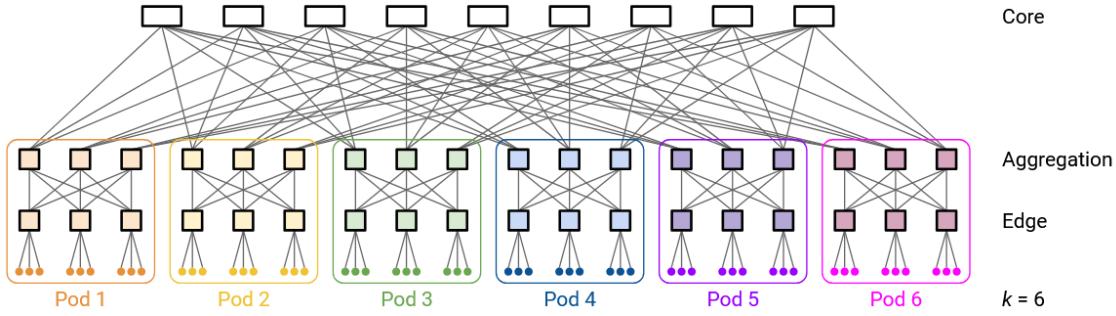
An edge layer switch has $k/2 = 3$ links downwards to 3 hosts, and $k/2 = 3$ links upwards to the 3 aggregation switches in the same pod.

An aggregation layer switch has $k/2 = 3$ links upwards to the core layer (specifically, to 3 different core layer switches), and $k/2 = 3$ links downwards to the 3 edge layer switches in the same pod.

Each pod has $k/2 = 3$ edge switches, each connected to $k/2 = 3$ hosts, so each pod has a total of $(k/2)^2 = 9$ hosts. The topology has k pods in total, for a total of $k \times (k/2)^2 = 54$ hosts.

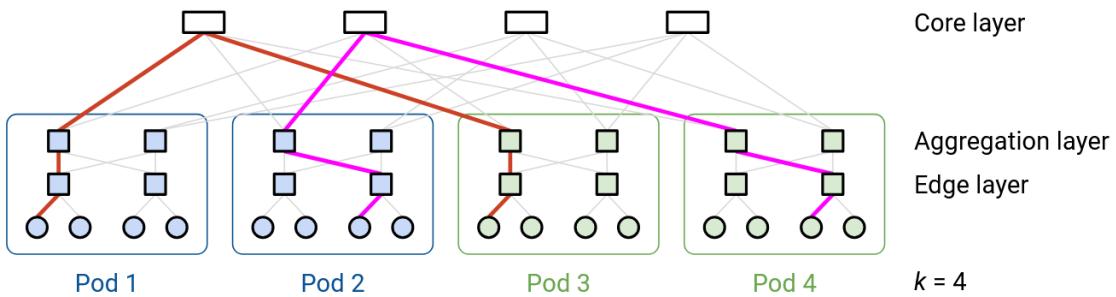
At the core layer, we have $(k/2)^2 = 9$ core switches. Each switch has $k = 6$ links, connecting downwards to each of the $k = 6$ pods.

In total, the core layer has $(k/2)^2 \times k$ links pointing downwards (number of core switches, times number of links per switch). The aggregation layer has $k \times (k/2) \times (k/2)$ links pointing upwards (number of pods, times number of aggregation switches per pod, times number of upwards links per aggregation switch). These two expressions match (and evaluate to 54 for $k = 6$), allowing the core layer to be fully-connected to the aggregation layer.



This topology achieves full bisection bandwidth. If you split the pods into two halves (e.g. left half and right half), then every host in the left half has a dedicated path to a corresponding host in the right half. This allows all the hosts to pair up (one in left half, one in right half), and for each pair to communicate along a dedicated path, with no bottlenecks.

Also, notice that this topology can be built out of commodity switches. Every switch has a radix of k links, regardless of which layer the switch is in. Also, every link can have the same bandwidth (e.g. 1 Gbps), and the scalability comes from the fact that we've created a dedicated path between any pair of hosts.

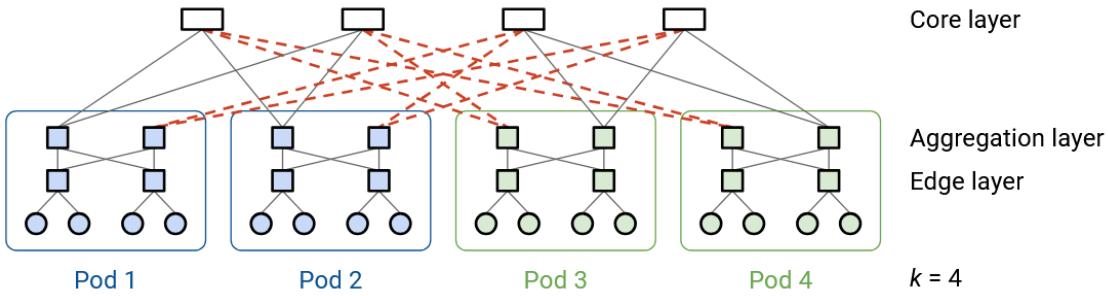


Another way to see the full bisection bandwidth is to delete links until the network is partitioned into two halves (pods in the left half, and pods in the right half).

Each core layer switch has k links, one to each of the pods. This also means that each core layer switch has $k/2$ links to the left side, and $k/2$ links to the right side.

In order to fully isolate one side (e.g. fully isolate the left side), then for each core switch, we'd have to cut $k/2$ links to the left side. There are $(k/2)^2$ core switches, and we have to cut $k/2$ links per switch, for a total of $(k/2)^3$ links cut. This means our bisection bandwidth is $(k/2)^3$ links (assuming every link has identical bandwidth).

There are $(k/2)^2$ hosts per pod, and $k/2$ pods in the left side, for a total of $(k/2)^3$ links in the left side. Similarly, there are $(k/2)^3$ links in the right side. If every host in the left side wanted to communicate with every host in the right side, then $(k/2)^3$ links' worth of bandwidth would be needed. Our bisection bandwidth matches this number, which means that full bisection bandwidth is achieved.



How does this Clos fat-tree topology relate to the idea of racks and top-of-rack switches from earlier?

For specific nice values of k , we can arrange the hosts and switches inside a pod into separate racks, and connect the racks to each other.

For example, consider $k = 48$, the example value used in the original paper. This means that inside a pod, there are $k/2 = 24$ aggregation layer switches, $k/2 = 24$ edge layer switches, and $(k/2)^2 = 576$ hosts per pod.

We can arrange the switches and hosts such that all 48 switches live in a rack that we place in the middle. Then, we can surround that rack of switches with 12 racks, each holding 48 hosts. This helps us fit all switches and hosts into identically-sized racks (48 machines per rack). Placing the switches in the middle rack also reduces the amount of physical wiring needed to build this topology.

The middle rack has $k = 48$ switches. Each switch has $k = 48$ ports, for a total of $48^2 = 2304$ ports in this rack.

Of these $k^2 = 2304$ ports, half of them ($k^2/2 = 1152$) connect switches inside the rack to each other. How did we derive $k^2/2$? It might help to look at some of the conceptual diagrams from earlier. Each of the $k/2$ aggregation layer switches has $k/2$ downward links, for a total of $(k/2)^2$ ports used. Similarly, each of the $k/2$ edge layer switches has $k/2$ upward links, for a total of $(k/2)^2$ ports used. This gives a total of $2 \times (k/2)^2 = k^2/2$ ports used.

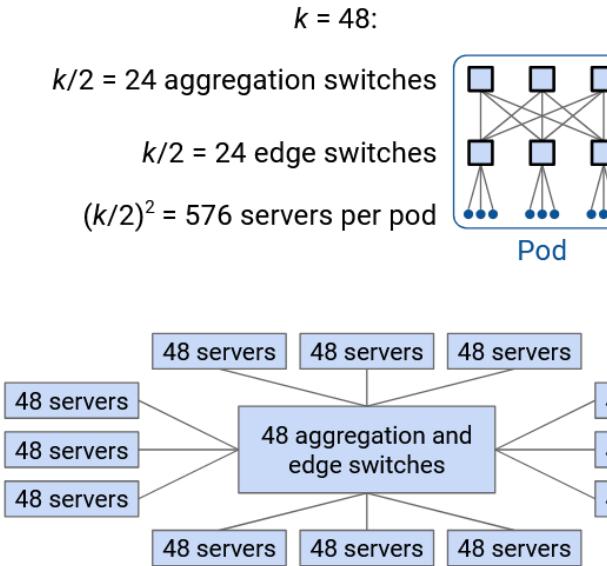
Note that the links between aggregation and edge switches are connecting switches inside the same rack. Therefore, two ports are needed for each link (one from an aggregation switch, and one from an edge switch), and that's why we doubled the $(k/2)^2$ value (or equivalently, accounted for that value twice at both the aggregation and edge layers).

Of the $k^2 = 2304$ ports, another quarter of them ($k^2/4 = 576$) connect switches to hosts inside the same pod. How did we derive this number? Remember that there are $(k/2)^2$ hosts within a pod, and each host is connected to exactly one switch. Therefore, we need $(k/2)^2 = k^2/4$ ports on the switches to connect to hosts.

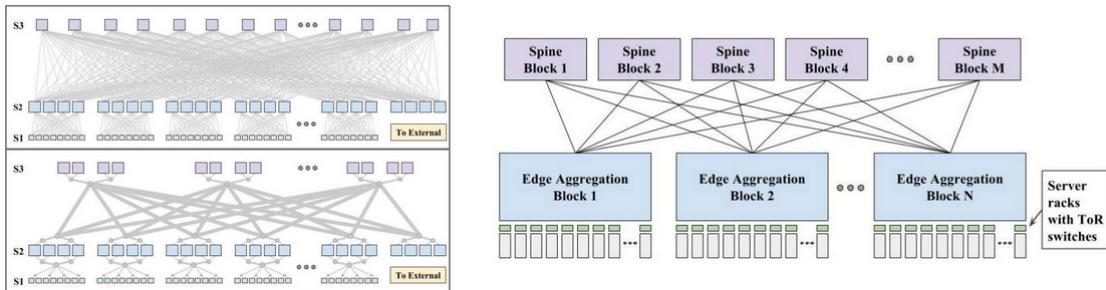
Finally, of the $k^2 = 2304$ ports, the remaining quarter ($k^2/4 = 576$) connect the pod to the core layer. How did we derive this number? Remember that there are $(k/2)^2$ core switches, and each core switch has a link to each pod. In other words, a pod has a single link to each of the $(k/2)^2$ core switches. Therefore, we need $(k/2)^2 = k^2/4$ ports on the switches to connect to the core switches.

In summary: Out of k^2 total ports, half of them are used to interconnect aggregation/edge switches in the same layer (connections happen entirely within the middle rack). Another quarter of them are used to connect edge switches to hosts in the pod (connections between the middle rack and the 12 surrounding racks with hosts). The last quarter of them are used to connect aggregation switches to the core layer

(connections between the middle rack and other core-layer racks).



Real-World Topologies

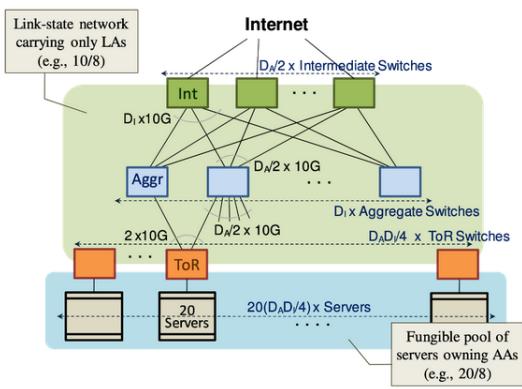


Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network

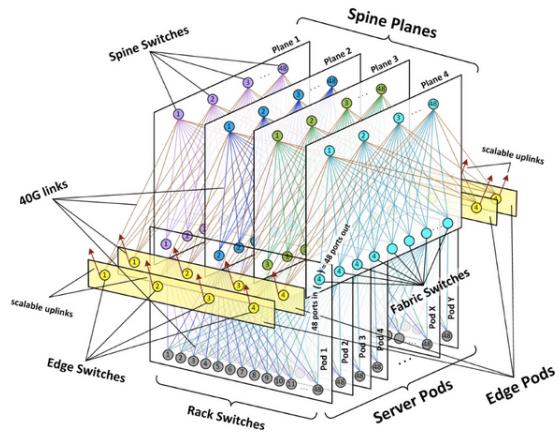
Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagal, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hözle, Stephen Stuart, and Amin Vahdat
Google, Inc.
jupiter-sigcomm@google.com

ACM SIGCOMM 2015

In this example (2008), there are many different paths between any two end hosts.



VL2 @ Microsoft, ACM SIGCOMM'09
Greenburg, Hamilton, Jain, Kandula, Kim, Lahiri, Maltz, Patel, Sengupta



"Introducing data fabric, the next-generation Facebook data center network", Alexey Andreyev, 2015

In this paper (2015), various topologies were explored.

Many specifics variants exist (2009, 2015), but they all share the same goal of achieving high bandwidth between any two servers.

Congestion Control in Datacenters

Why are Datacenters Different?

We've seen that datacenter networks have additional constraints (e.g. physically in one building, owned by one operator) compared to general-purpose networks. This can lead to special protocols that exploit these special characteristics of the network. In this section, we'll explore TCP congestion control algorithms that may not work on the general Internet, but are effective in datacenter contexts. This is an active area of research and development.

First, we should answer: What makes the congestion control different in a datacenter?

Recall that packet delay consists of transmission delay (time to signal the bits on the wire, determined by bandwidth), propagation delay (time for bits to travel across wire), and queuing delay.

In datacenters, transmission delay is usually relatively small (remember, we have high-capacity 10 Gbps links). Propagation delay is also relatively small in datacenters (remember, all the servers are in the same building). As a result, in datacenters, queuing delay is often the dominant source of delay. By contrast, in the wide-area Internet, the propagation delay could be orders of magnitude larger (e.g. packets could have to travel across the country), and is a more common dominating source of delay.

Recall that TCP congestion control deliberately fills up queues until packets get lost (we detect congestion by checking for loss). The TCP designers had not considered datacenter contexts, where queuing delay can have a much larger impact on performance.

The problem of large queues is exacerbated in datacenters, because unlike in the wide-area Internet, most datacenter connections fall into one of two categories. Most connections are **mice**, which are short and latency-sensitive. For example, a web search query and the results page contain a very small amount of data, but we want to return the results to the user very quickly. By contrast, some connections are **elephants**, which are large and throughput-sensitive. For example, backing up data from one server to another server requires a long-running connection that sends a lot of data at high throughput.

If we run TCP congestion control with these two types of connections, the elephants will increase their rates until the queues are all filled up. Now, any subsequent mice will be stuck in the queues, causing the mice to be delayed.

In order to maximize performance for these specific types of connections, datacenter congestion control algorithms must avoid filling up queues. Many datacenter-specific solutions have been developed in recent years.

For example, BBR was released by Google in 2016. In this approach, instead of detecting congestion by checking for loss (which requires queues to be full), we instead detect congestion by checking for packet delay.

DCTCP: Feedback from Routers

DCTCP (Datacenter TCP) was released in 2010 by Microsoft, and is now widely used (e.g. implemented in the Linux kernel).

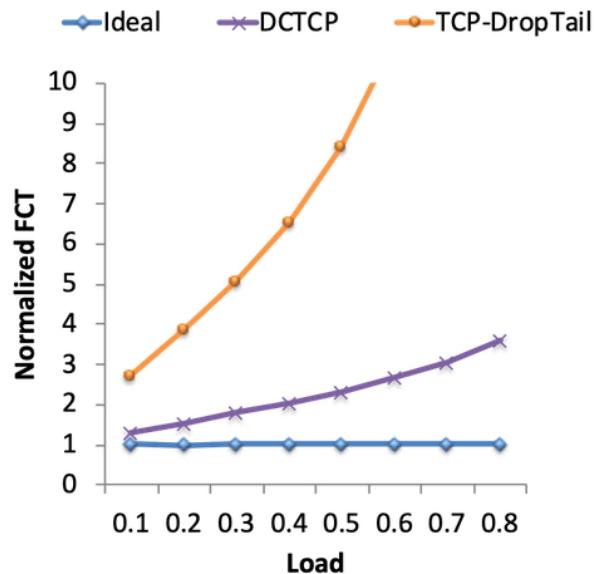
Recall that the IP header has an ECN bit, and the router can enable this bit to indicate that it's congested. When the packet reaches the destination, the ack will also have the ECN bit set, and this tells the sender to slow down.

In DCTCP, routers will enable the ECN bit when the queue length exceeds some threshold. This allows senders to detect and adapt to congestion earlier (as the queue is filling up, before the queue gets totally full).

In response to congestion, the sender reduces the rate in proportion to the number of packets with ECN markings. This allows the sender to adapt to congestion more gently. Instead of binary decision (congestion or no congestion), the sender can detect that some congestion might be happening, and slightly decrease the rate to compensate.

The ECN bit is not very effective in the wide-area Internet because not all routers support it. However, in a datacenter, the operator controls all the switches and can have them all toggle the ECN in a consistent way. In practice, implementing DCTCP at hosts and routers is a relatively small change.

To measure how DCTCP performs, we can measure **flow completion time (FCT)**, which measures the time between the first byte being sent and the last byte being received. As a benchmark, the ideal FCT is the completion time if we used an omniscient scheduler that had global knowledge of the entire network and all connections. The scheduler could then use that knowledge to optimally schedule flows and allocate bandwidth to flows.



This graph shows the normalized FCT, which is a ratio of actual FCT to ideal FCT. This tells us how much worse we're doing, compared to the ideal congestion control algorithm. We can see that standard TCP congestion control performs 3x worse than ideal, and up to 10x worse than ideal if the load on the network is higher. By contrast, DCTCP performs significantly better than standard TCP congestion control. DCTCP connections are finishing much faster, with less queuing delay.

pFabric: Packet Priorities

We saw that the issue in datacenters is that mice can be trapped in queues behind elephants. What if we gave the mice some way to skip to the front of the queue so they could complete faster?

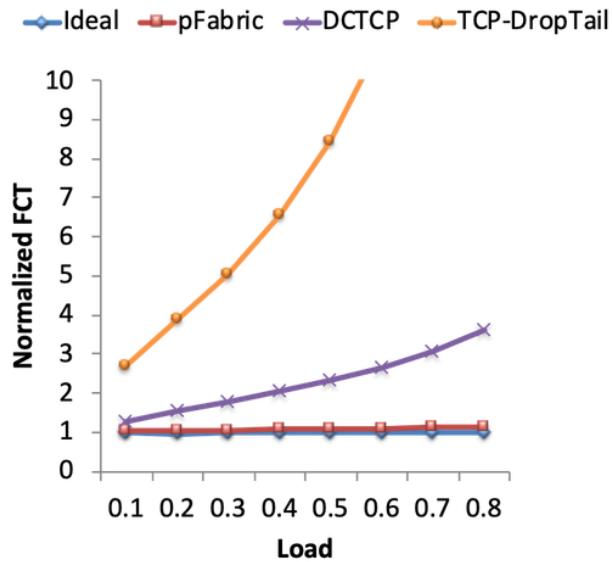
To prioritize mice, we will assign a priority number to every packet. The priority is computed as the remaining flow size (number of unacknowledged bytes). Lower numbers have higher priority.

With this system, mice packets will be high-priority (flow sizes are very low). Elephants will be low-priority, though the last few bytes in an elephant connection will be higher-priority. This has the effect of prioritizing connections that are almost-done (even if they're larger elephant connections).

To implement this idea, recall that IP packet headers have fields to indicate the priority of a packet. In pFabric, each packet carries a single priority number, and switches are modified so that they send the highest-priority packet. If the queue is full, the switch will drop the lowest-priority packet.

With the priority system in place, senders can now safely transmit and retransmit packets at full line rate, without needing to adjust their rate for congestion control. Senders only need to reduce their rate in cases of extreme loss (e.g. timeout).

If we look at the graph of FCTs again, we see that pFabric performance is even better than DCTCP, and is very close to ideal.



Why does pFabric work so well? Elephants and mice travel together, and everybody is sending at full line rate, which ensures full utilization of the available bandwidth. We don't have to waste time on slow start. Also, we can avoid collapses because most of the packets in large elephants are low-priority. The priority system ensures that mice packets still get through the queue with low latency.

Implementing this system requires non-trivial changes at both switches and end hosts, and requires full control of both switches and end hosts. Switches need to implement a priority system, and senders need to replace their TCP implementation to send at full line rate. Still, pFabric is a good example of networks (switches) and end hosts cooperating to achieve good performance.

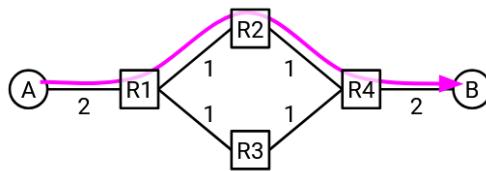
Datacenter Routing

Why are Datacenters Different?

In the previous section, we designed Clos networks, which created many paths between servers. Servers can communicate simultaneously at high bandwidth by using different paths through the network.

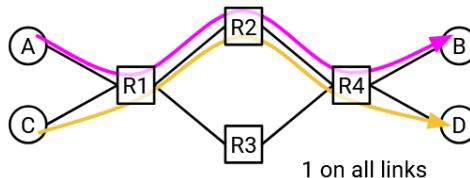
What problems occur if we apply our standard routing algorithms to these network topologies?

So far, our routing protocols pick a single path between a source and destination. If all our traffic uses the same path, we aren't taking advantage of all the extra links in the Clos network. Ideally, we'd like to modify our routing protocols so that a packet can use multiple paths between the same endpoints.



Suppose that A and B have 200 Gbps uplink bandwidth, and the switch-to-switch links have 100 Gbps bandwidth. If all traffic between A and B is forced to take the green path, we're leaving the red path unused. We could have sent data at full rate, if we allowed packets to take different paths.

Also, if there are multiple simultaneous connections, we'd like those connections to use different paths in order to maximize bandwidth.

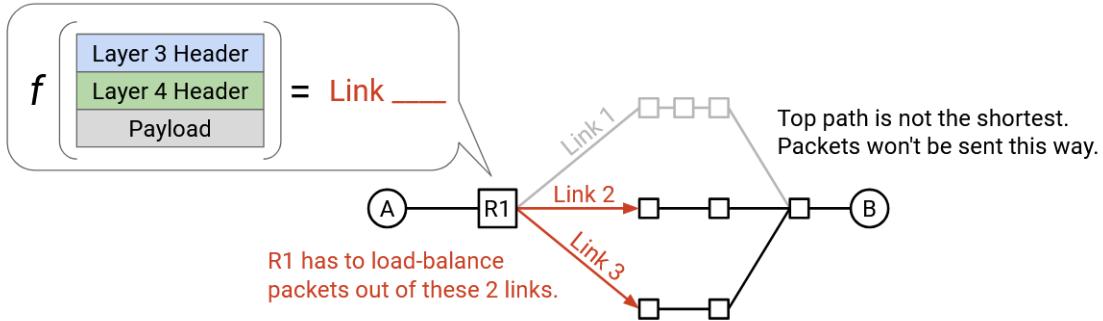


Suppose that all links have 100 Gbps bandwidth. In this example, multiple connections are competing for bandwidth. If the A-B and C-D connections both pick the same path, the R1-R2 and R2-R4 links are overused (200 Gbps on 100 Gbps capacity). We could have sent data at full rate, if A-B and C-D used different paths.

Equal Cost Multi-Path (ECMP) Routing

In **equal cost multi-path** routing, our goal is to find all of the shortest paths (with equal cost), and load-balance packets across those paths.

If a packet arrives at a router, but there are multiple outgoing links that are all valid shortest paths, which link should the router choose? The router needs some function (think of it like a piece of code) that takes a packet, and outputs a choice of link. The function should properly load-balance traffic across the equal-cost paths.

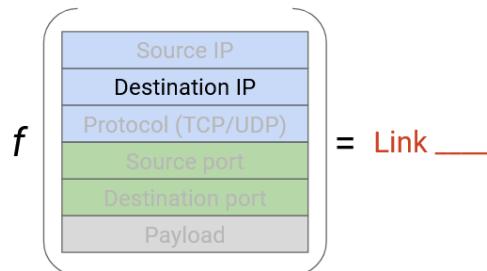


One possible strategy is round-robin. If there are two shortest-path outgoing links, our function could say: send all odd packets along Link 1 and all even packets along Link 2.

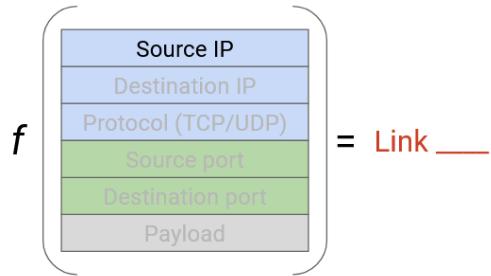
What are some problems with this approach? Equal-cost paths doesn't necessarily mean all paths have the same latency. (Remember, the costs are defined by the operator using whatever metric they like.) If we send all odd packets along a slow link, and all even packets along a fast link, then the TCP recipient might end up receiving all the even packets before the odd packets. TCP cares about reordering packets, so the recipient would be forced to buffer the even packets until the missing odd packets arrive, resulting in poor performance.

A smarter strategy would involve looking at some of the packet header fields, and using those fields to make some deterministic choice of link. What fields could we look at?

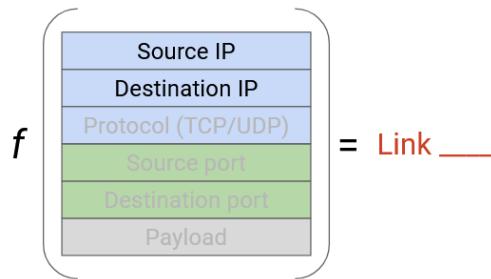
We could use the destination IP to select between shortest links. (We're already using the destination IP in routing anyway.) But, what if lots of sources send packets to the same destination? All the packets have the same destination IP, so they all get mapped to the same shortest link. We aren't load-balancing packets across the various shortest links.



What if we used the source IP to select between shortest links? We have a similar problem, if one source is sending packets to lots of destinations. All the packets have the same source, so they all get mapped to the same shortest link.

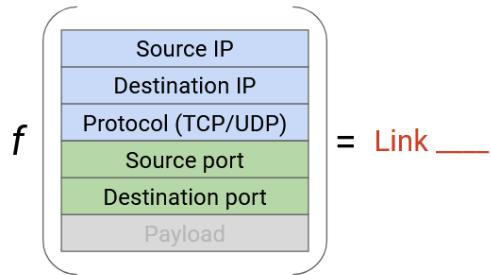


Instead of looking at only one field, we could look at both the source and destination IP. To load-balance between shortest links, we could hash the source and destination IP and map the resulting hash to a link (similar to how hash tables work). The source and destination IP together contain enough entropy to avoid our problems from earlier, where many connections with the same source or the same destination get mapped to the same link.



We still have one more problem: What if there are multiple large connections between the same source and destination? We don't want all these connections to map to the same link. To solve this, we can additionally look at the source and destination ports in the TCP or UDP header.

More generally, all of the problems we've described (reordering in a TCP connection, too many connections on one link) can be solved if we place each connection on a separate link. To uniquely identify a connection, we need a 5-tuple of: (source IP, destination IP, protocol, source port, destination port). Note that we need the protocol to distinguish between TCP and UDP connections using the same IPs/ports. Two packets are part of the same connection if and only if they have the same 5-tuple.



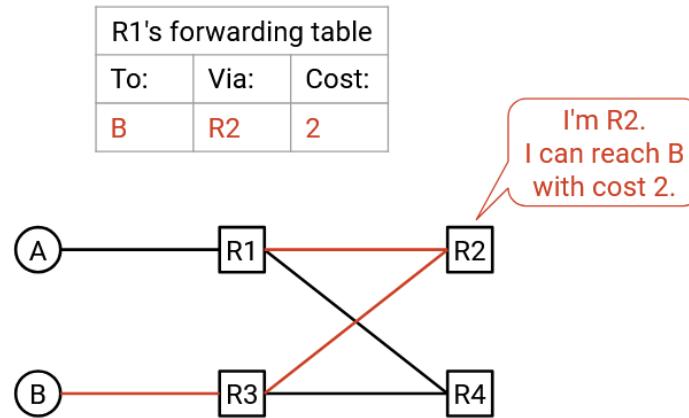
By hashing all 5 values, we can ensure packets in the same connection use the same path (avoiding reordering problems), and we can load-balance connections across different paths. This approach is sometimes called **per-flow load balancing**. Modern commodity routers usually have built-in support to read these 5 values.

Per-flow load balancing ensures that each link is being used by roughly the same number of connections, though it doesn't account for connections being different sizes. Accounting for connection size is technically possible, though it's more expensive (routers would have to do more work) without a lot of benefit (per-flow does a pretty good job balancing different-sized connections), so this is not done in practice.

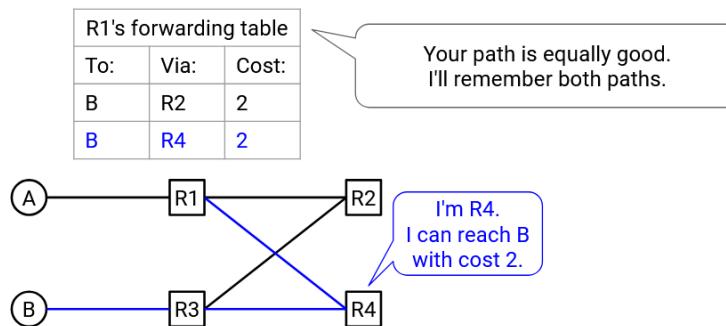
Multi-Path Distance-Vector Protocols

To maximize bandwidth, we should send packets along different paths, even if they're going to the same destination (e.g. if the packets are part of different connections). This means we have to modify our routing protocols so that routers learn about all the shortest paths, not just one.

In standard distance-vector protocols, if we receive an advertisement for a new path with cost equal to the best-known cost, we don't accept that new path. But, in order to remember all least-cost paths, we should actually accept that equal-cost path, and store both paths in the forwarding table. In the forwarding table, a destination can now be mapped to multiple next hops, as long as they all have the same minimal cost.



In this example, R1 receives advertisements from both R4 and R3, both advertising that they can reach B in 2 hops. Our forwarding table stores both R4 and R3 as possible next hops, both with equal minimal cost of 3.



When forwarding packets, the router hashes the 5-tuple to forward roughly half the connections to R3, and the other half to R2.

Multi-Path Link-State Protocols

In link-state protocols, we flood advertisements so that everybody has a full picture of the network. Normally, each node calculates a shortest path to each destination to populate the forwarding table. To support multiple paths, we need each node to instead compute all of the shortest paths for each destination.

As in the modified distance-vector protocol, the forwarding table can now contain multiple next-hops for a given destination.

Datacenter Addressing

Why are Datacenters Different?

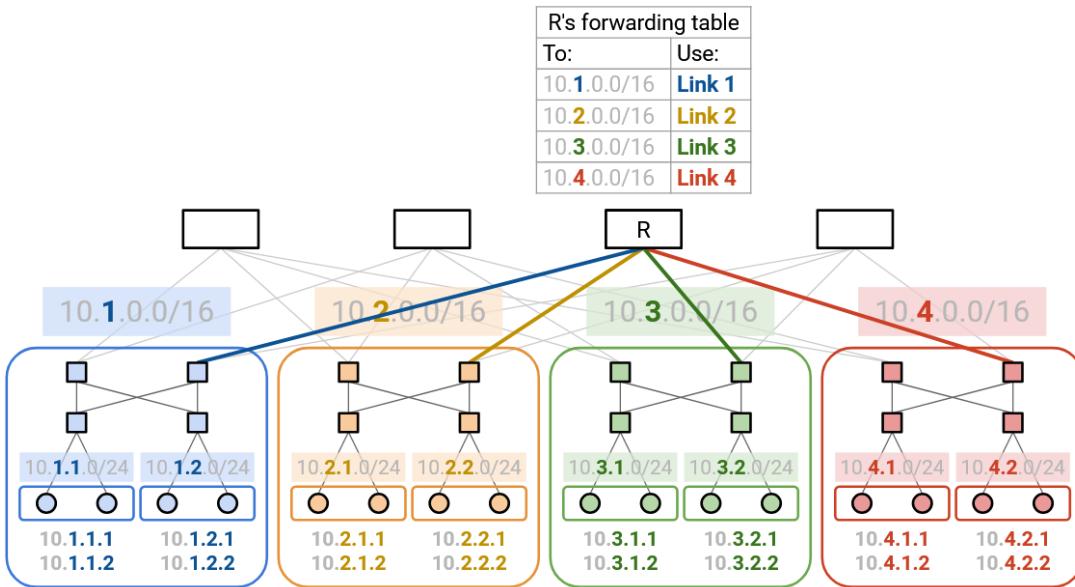
In the previous section, we saw that we can modify distance-vector and link-state routing protocols to compute all paths through the datacenter network.

However, these protocols might scale poorly in datacenters. In distance-vector protocols, we have to make an announcement for every destination, which means that 100,000+ destinations have to be advertised. In link-state protocols, we have to flood advertisements along every link, which scales poorly in Clos networks with a huge number of links. Also, recall that datacenter topologies often use cheap commodity switches, which have limited memory and CPU resources (e.g. the forwarding table can't be too large).

In general-purpose networks, we solved these scaling problems by introducing hierarchical IP addressing. Higher-level organizations (e.g. country-level) could allocate ranges of addresses to smaller organizations (e.g. universities). Datacenters don't have geographic and organizational hierarchies that we can use to organize addresses.

However, in datacenters, we can exploit the fact that the operator controls the physical topology of the network, and assign addresses to servers based on where they're located in the building. We can also exploit the fact that the topology has some regular structure (e.g. we're probably organizing servers in rows, instead of randomly stuffing them in the building).

Topology-Aware Addressing



In this particular topology, the racks are physically organized into separate pods in the building. One natural approach would be to allocate a range of addresses to each pod. Then, each pod can allocate sub-ranges to each rack in the pod. Finally, each rack can allocate an individual IP address to each server.

The operator knows how many servers are in each rack, and how many racks are in each pod, so we can use that information to allocate ranges of the appropriate size. For example, a rack could receive a /24 range, which gives that rack 256 addresses for its servers.

This allocation approach lets aggregate routes and store fewer entries in our forwarding table. For example, consider one of the spine routers at the top of the diagram. This router doesn't need to remember a path for every single server. Instead, the forwarding table only needs four entries, one for each pod. When a packet arrives, the router checks the first 16 bits to forward the packet to the appropriate pod.

Route aggregation also results in more stability. If a host is added or removed inside a specific rack, the spine router doesn't need to know. As long as we maintain the same addressing scheme, the existing forwarding table is still correct without any changes. As a result, routing updates usually occur when links and switches fail, but not when hosts fail.

Assigning addresses based on datacenter topology is good for scaling, but there are some limitations. In particular, if we move a server to a different location, we'd have to change its address.

Virtualization and Encapsulation

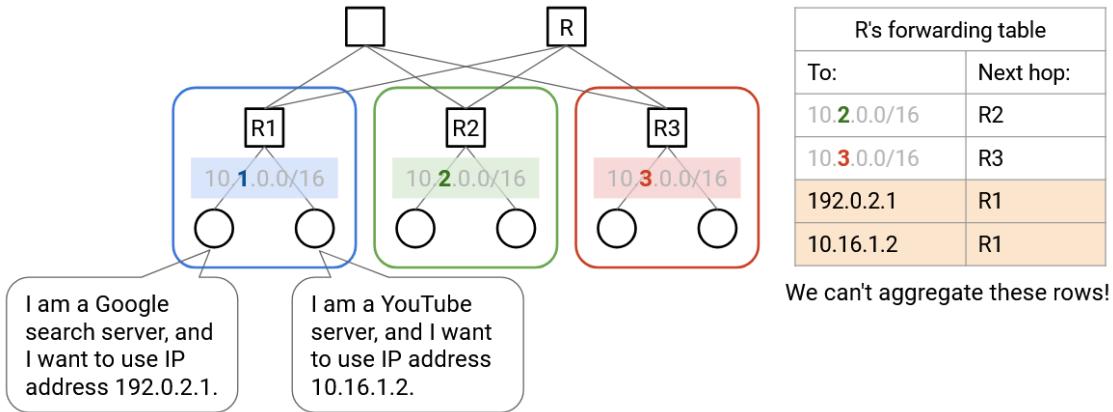
Physical Datacenter Limitations

Datacenters are organized in a fixed and structured way. Identical servers are organized into racks, and racks arranged in some fixed topology. This approach has some benefits. For example, it gives us a natural way to assign hierarchical addresses.

However, when we consider how applications are hosted on datacenters, the fixed organization of datacenters has some downsides. Suppose Google introduced a new service that they want to host in an existing datacenter. If we placed that application directly on a physical server, someone would have to physically install a new server, with its own IP address, for this application. If the service expands, more servers might need to be installed. If the server goes down, we'd have to wait for somebody to fix it. The key problem here is that changing physical infrastructure is hard, but we often want to add new hosts, scale up existing hosts, and move hosts quickly and frequently.

Placing applications on physical servers also introduces scaling issues. Suppose Google's new service is very lightweight, but needs a dedicated server (e.g. for security reasons). We'd have to assign an entire physical server to this lightweight service, and most of the server's computing capacity would be unused.

This approach also has routing issues. Suppose we wanted to move the service to a different part of the datacenter building (e.g. because part of the building is undergoing maintenance). First, someone would have to physically move the server in the building. Also, in our hierarchical address model, we would need to assign this service a new IP address corresponding to its new physical location. Ideally, the application would prefer to keep the same address, regardless of its datacenter location.



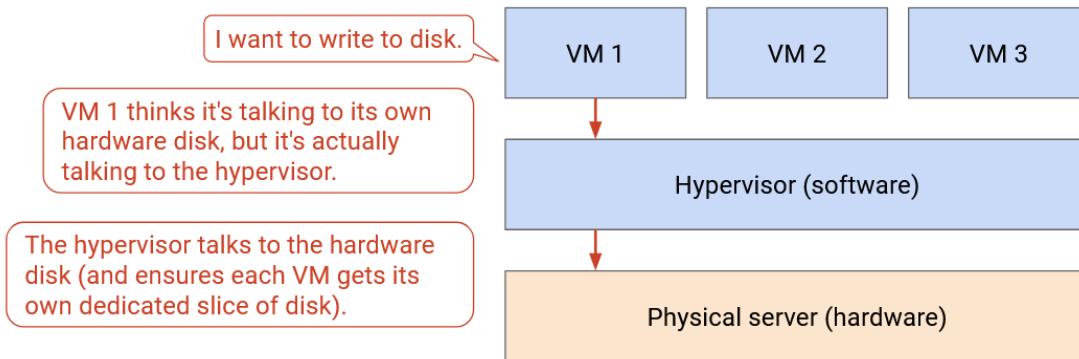
Virtualization

We can use virtualization to solve these problems and give applications more flexibility, while maintaining the rigid physical structure of the datacenter. **Virtualization** allows us to run one or more virtual servers inside a physical server.

The virtual server gives applications the illusion that they are running on a dedicated physical machine.

However, in reality, multiple virtual servers might be running on the same machine. When the application tries to interact with hardware (e.g. disk, network card), it is actually interacting with a **hypervisor** in software. The hypervisor presents each virtual application with the same interface that real hardware would. The hypervisor itself runs on actual physical hardware, and can forward application requests (e.g. disk write, network packet send) to the hardware level.

With virtualization, if we have a new application, we can ask a hypervisor to start up a new virtual machine for this application. The hypervisor runs in software, so there's no need to install any new server in the physical datacenter. Similarly, we can move hosts to a different physical machine, entirely in software.

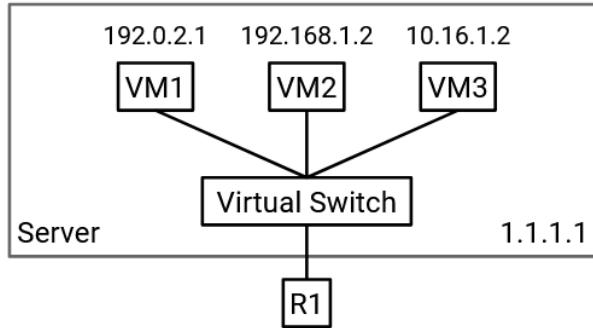


Virtualization allows multiple applications to share a physical server. The applications can be separated from each other, and can be managed by different people. This lets us use the compute resources in the datacenter more efficiently. This also allows us to have more hosts in the datacenter. For example, a single rack with 40 servers could have more than 40 end hosts.

Virtual Switches

The physical server has a single network card and a single IP address, but we need to give each virtual machine the illusion that it has its own dedicated network card and address. Also, switches might now have multiple virtual machines connected to a single physical port.

In order to manage multiple network connections on the same physical machine, the server needs a **virtual switch**. This virtual switch runs in software on the server (it's not a physical router), and performs the same operations as a real switch (e.g. forwarding packets). Each virtual machine is connected to the virtual switch, and the virtual switch is connected to the rest of the network.



Note: Switches usually run on dedicated hardware to maximize efficiency. Virtual switches can be run in software on a general-purpose CPU because they only need to support a few virtual machines (lower capacity than what switches usually handle).

Underlay and Overlay Network

With virtualization, we now have virtual hosts running on top of physical servers. Unlike physical servers, virtual hosts can be created, shut down, and changed rapidly.

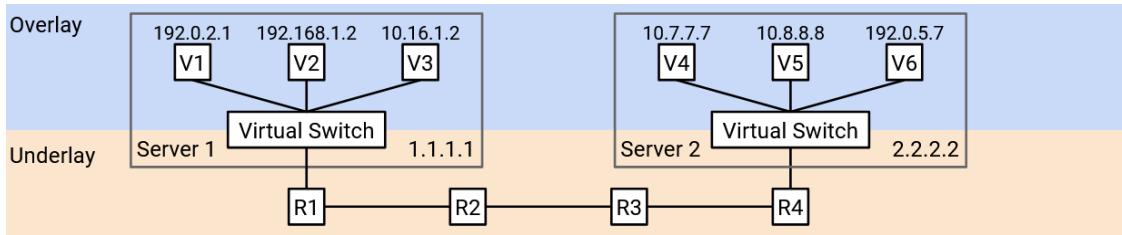
Virtual machines don't necessarily use the same addressing scheme as the physical servers. Physical server IP addresses are defined by the physical datacenter topology (e.g. pods, racks). By contrast, virtual machine IP addresses are usually defined by some real-life hierarchy (e.g. countries, organizations). In particular, the virtual hosts on a single physical server don't necessarily all have the same IP prefixes, so we can't use the same aggregation tricks to scale up.

If we tried to naively extend our routing schemes to support virtual machines, our forwarding tables would become very large, very quickly. Previously, we could aggregate by saying: "all servers in the blue pod have the same IP prefix, and they all have a next hop of R2." Now, the servers in that blue pod could contain hundreds of virtual hosts, all with different IP addresses (no common prefix). We would need a separate forwarding entry for every virtual host. Also, if a virtual host moves to a different physical machine (keeping the same IP address), the routing protocol would have to re-discover paths to this virtual host. Can we find a way to avoid scaling the datacenter to support every VM address?

The key problem here is that we now have two different addressing systems, one for virtual hosts, and one for physical hosts. Both addressing schemes work at the IP layer, but within the IP layer, there are now two sub-layers of abstraction that we need to think about.

The **underlay network** handles routing between physical machines. The underlay network contains datacenter infrastructure like top-of-rack switches and spine switches. The underlay network scales well because we define hierarchical addresses using the physical datacenter topology.

The **overlay network** exists on top of the physical topology (underlay), and it only thinks about routing between virtual machines. In practice, each virtual machine usually only needs to communicate with a few other virtual machines in the network. As a result, the overlay network scales well because a virtual machine does not need to know about every single other virtual machine.



Ideally, we'd like the two layers to think about addressing separately. The underlay network should not need to know about virtual host addresses (otherwise, it would scale poorly). Similarly, the overlay network should not need to know about every physical server in the datacenter (each VM only needs to know about a few other VMs).

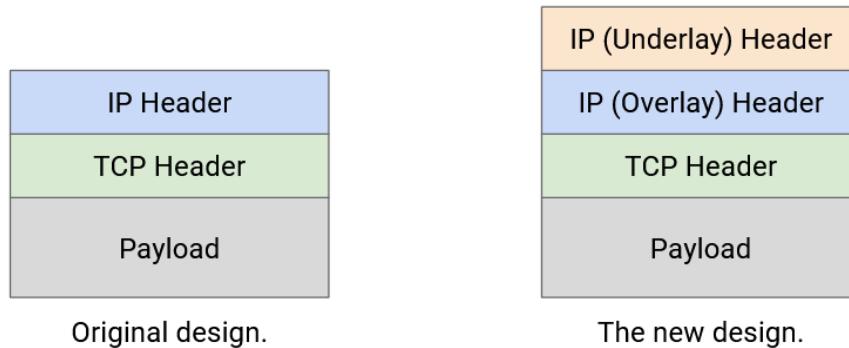
If we didn't tell the underlay network about virtual host addresses, then if a datacenter switch gets a packet with a virtual IP as the destination, it would look in its forwarding table, not find any virtual IPs, and drop this packet. We need some way to bridge the gap between the overlay (thinking virtually) and the underlay (thinking physically).

Encapsulation

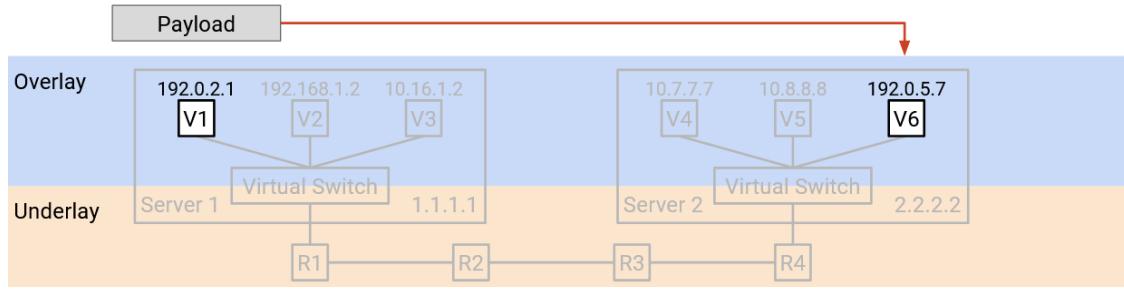
To unify the overlay and underlay layers, we can use the same strategies with layering and headers that we used when we designed the Internet!

So far, we've treated IP as a single layer, and every packet has a single IP header, which understands the IP addressing system.

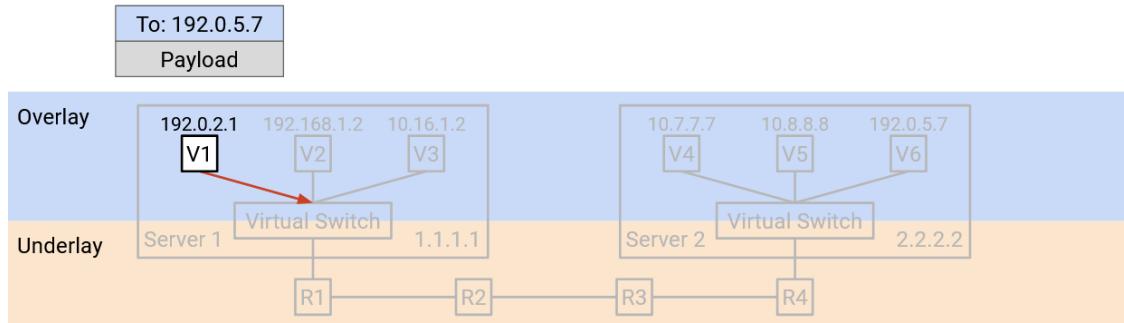
Now that we have two IP sub-layers with two different IP addressing systems, we could introduce an additional header into the packet. For example, we could have two IP headers, where one header understands the overlay network, and the other header understands the underlay network. Or, we could use the original IP header for the underlay network, and introduce a new type of header (different from IP) for the overlay network.



Now, our strategy for routing packets can combine the overlay and underlay networks. Suppose VM A wants to send a packet to VM B.



- VM A creates a packet with a single IP header, which contains the virtual IP address of B. (Remember, A is thinking in terms of overlay, and does not know about underlay physical IP addresses.) VM A forwards this packet to the virtual switch (on A's physical server).

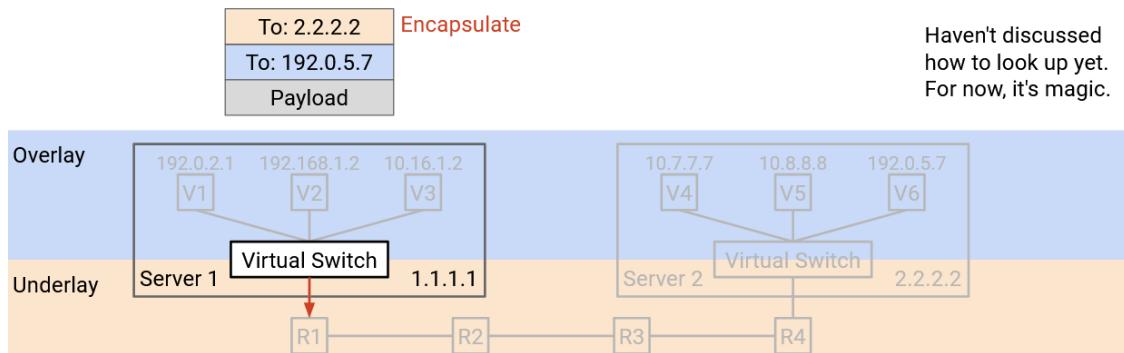


- The virtual switch reads the header to learn B's virtual IP address. Then, the virtual switch looks up the physical server address corresponding to B's virtual IP address. (We haven't described how to do this yet.)

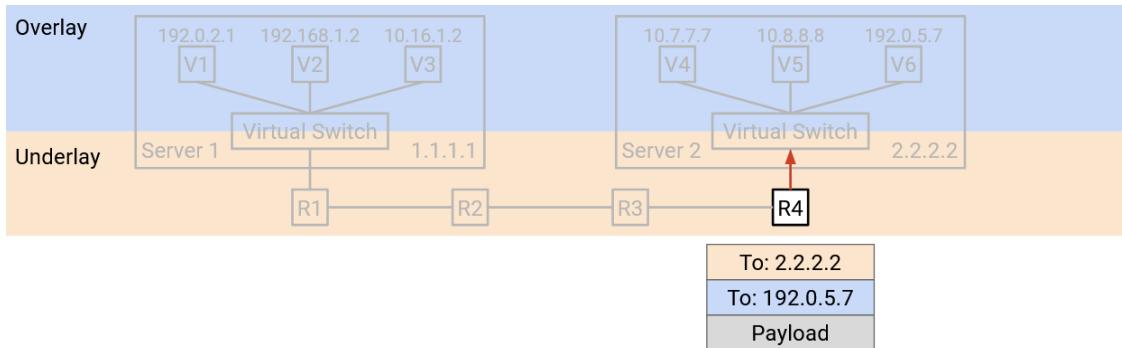
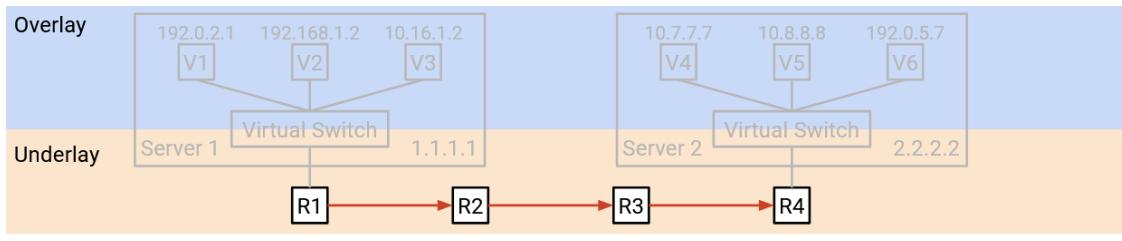
The virtual switch adds an additional outer header containing B's physical server address. Adding the header is sometimes called **encapsulation**.

At this point, the packet has two headers. The inner header (higher layer, overlay, added by VM A) contains B's virtual IP address, and the outer header (lower layer, underlay, added by virtual switch) contains B's physical server address.

The virtual switch forwards this packet to the next hop switch, based on the physical server address.

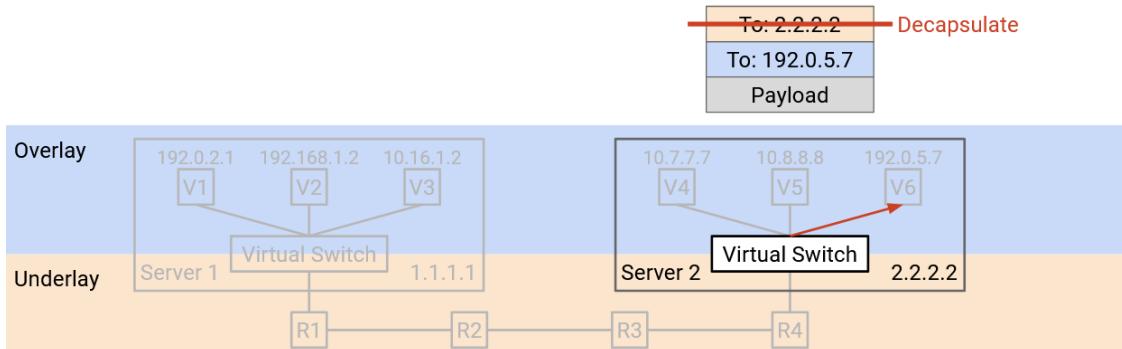


3. The packet is sent through the underlay network. Each switch in the datacenter only looks at the outer header (underlay, physical server address) to decide how to forward the packet. (Remember, the datacenter switches think in terms of underlay, and do not know about the overlay virtual IP address.)

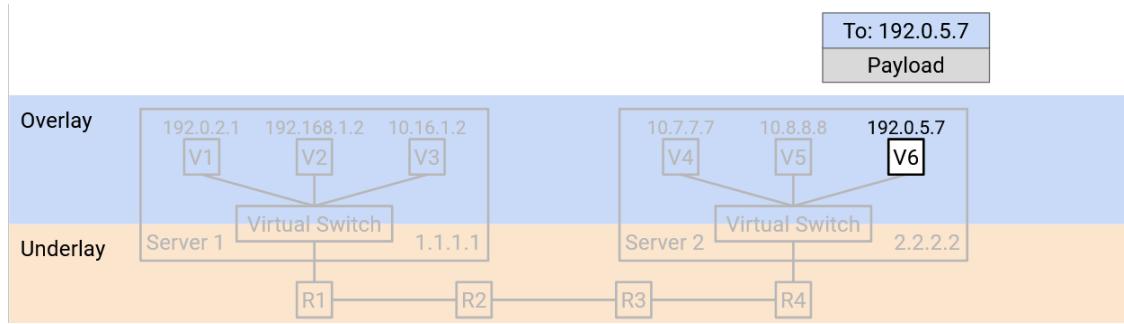


4. Eventually, the packet reaches the destination physical server's virtual switch. The virtual switch looks at the outer header (underlay) and notices that the destination physical server address is itself.

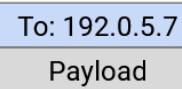
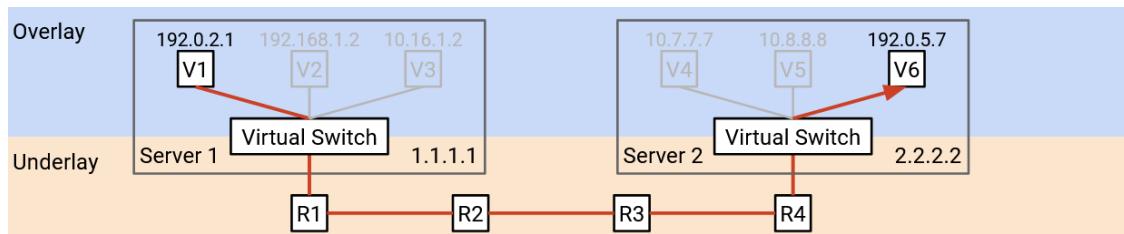
The virtual switch removes the outer header, exposing the inner header inside. Removing the outer header is sometimes called **decapsulation**.



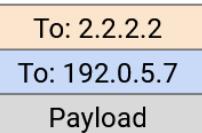
Finally, the virtual switch reads the inner header (overlay). This tells the virtual switch which of the VMs on the physical server the packet should be forwarded to.



In this process, **encapsulation** allowed us to think about routing at two different layers. The underlay was able to route packets using physical server addresses, without thinking about the overlay. Similarly, the VM in the overlay was able to send and receive packets without thinking about how to forward packets in the underlay. The virtual switches bridged the two layers by translating the virtual machine address into a physical server address, and adding and removing the extra underlay header.

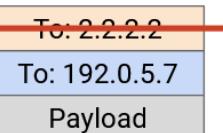


Original packet only has *virtual* (overlay) address.



We add the *physical* (underlay) address.

The extra header helps the packet travel through the underlay network.



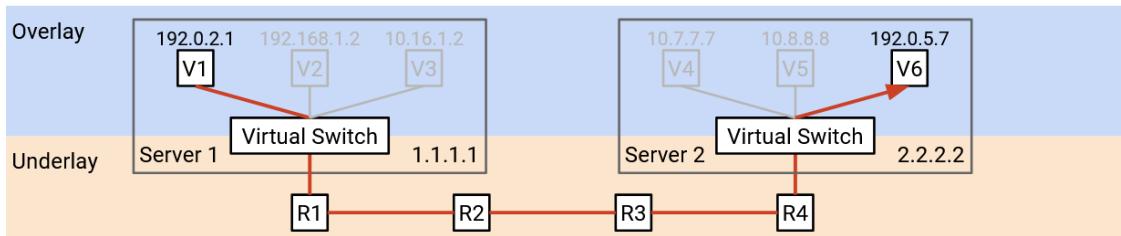
Eventually, we remove the extra header.

The packet travels based on the *virtual* (overlay) address the rest of the way.

VM1's forwarding table		Virtual switch's forwarding table		R2's forwarding table	
To:	Next hop:	To:	Next hop:	To:	Next hop:
Anywhere	Virtual switch	192.0.5.7	Add header: 2.2.2.2 Then, send to R1	1.1.1.1	R1
				2.2.2.2	R3

Haven't discussed how to map
192.0.5.7 → 2.2.2.2 yet.
For now, it's magic.

Only includes physical addresses
(which can be aggregated!)



Forwarding Tables with Encapsulation

What entries should we install in the forwarding tables to support routing with encapsulation?

The virtual machines should install a default route that forwards every packet to the virtual switch on the physical machine.

The virtual switches need to implement some extra functionality to bridge the two layers. In particular, when you see a virtual address, you should apply encapsulation (add an outer layer) with the corresponding physical address. The forwarding table has entries for every destination VM that any of the VMs on this server might want to talk to. We can support this scale because we assume the VMs won't need to talk to every other VM in the datacenter. Unlike standard routing algorithms, we don't need any-to-any routing (we don't need paths to every other VM).

Virtual switches also need an extra rule for decapsulating packets. If the outer (underlay) packet destination is the switch itself, you should decapsulate (remove the outer header) and pass the packet to the VM address in the inner header. This rule scales with the number of VMs on the server, which is usually small enough to be manageable.

Is it hard to add this functionality? Fortunately, virtual switches are implemented in software, so adding this functionality just requires writing code (no extra hardware needed). In practice, though, encapsulation is so common that it's sometimes implemented in hardware anyway.

The switches in the datacenter work exactly the same as they did before we introduced virtualization. The forwarding tables only contain physical server addresses, and we know that these can be scaled with aggregation tricks based on physical topology.

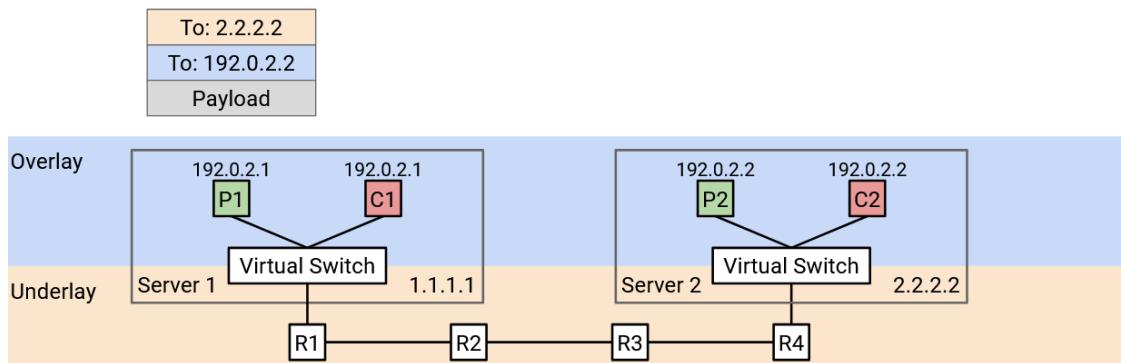
Multi-Tenancy and Private Networks

Datacenters are managed by a single operator, but different organizations might be running applications inside that datacenter. For example, a datacenter run by Google might have some virtual servers run by Gmail, and others run by Google Maps. This approach of hosting multiple services in one datacenter is called **multi-tenancy**.

Cloud providers also use datacenters to supply virtual machines for customers. For example, Amazon Web Services (AWS) and Google Cloud Platform (GCP) allow users to start up a virtual machine in a datacenter, do whatever they want, and destroy the virtual machine when they're done.

One problem with multi-tenancy is, we don't always want the different tenants to be able to communicate with each other. For example, if a customer requests a VM, they probably shouldn't be able to connect to every other VM in the datacenter.

Another problem is, tenants in a datacenter don't coordinate with each other when choosing addresses. For example, suppose our datacenter had two tenants, Pepsi and Coke. Each tenant creates their own private network, where they assign internal IP addresses to virtual machines. The private network is only for hosts inside the datacenter to communicate with each other, and these hosts will never be contacted from the public Internet. Because the networks are private, the two tenants can both use addresses in the same specially-allocated private ranges (RFC 1918 addresses). Pepsi's private network might have a VM with IP address 192.0.2.2, and Coke's private network might have a different VM with IP address 192.0.2.2. (In practice, we use private ranges in order to reuse IPv4 addresses, since we're running out of them.)

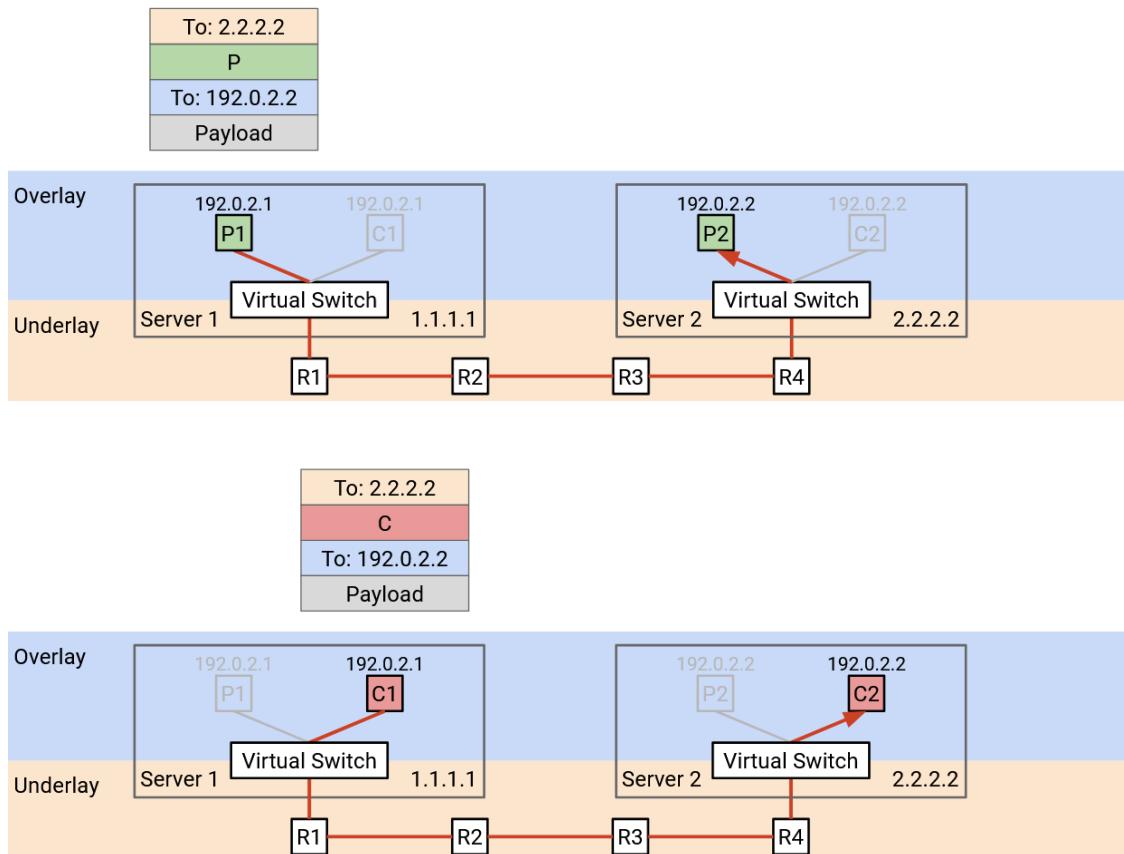


From the perspective of each tenant, this is not a problem. Pepsi's 192.0.2.2 will never communicate with Coke's 192.0.2.2, and neither host is accessible to the global Internet. However, this is a problem for the datacenter. If we use destination-based forwarding, and we see a packet with destination 192.0.2.2, we have no idea which VM this address is referring to.

Duplicate IP addresses occur in practice for two reasons. First, datacenters usually don't have control over what addresses the tenants are assigning to their VMs. Second, in IP, it's standard practice to use specific ranges for private networks, which often leads to duplicate addresses.

Encapsulation For Multi-Tenancy

We can use the idea of encapsulation again to solve this problem. We can add a new header that contains a **virtual network ID** for identifying a specific tenant (e.g. Pepsi has ID 1, Coke has ID 2). This new header doesn't contain information for forwarding and routing, but it provides additional context. Now, if a physical server has VMs for multiple tenants, it can pass the packet up to the correct virtual network.



When a virtual switch receives a packet and unwraps the outer (underlay) header, it looks at our new header to decide which tenant the packet is meant for. Then, it looks at the overlay header to forward the packet to a specific VM belonging to the correct tenant.

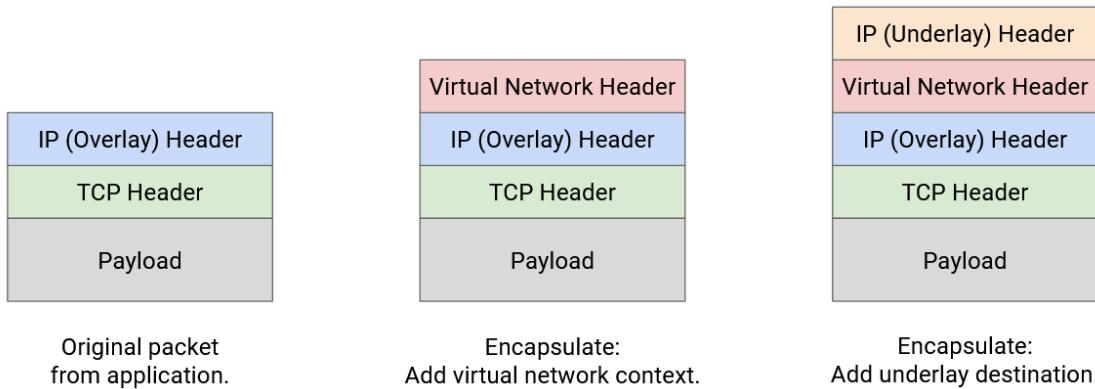
Stacking Encapsulations

We can use the idea of encapsulation multiple times, adding multiple new headers to support both virtualization and multi-tenancy.

To start, the virtual machine creates a standard TCP/IP packet, with a virtual IP destination.

In the first encapsulation step, we add a virtual network header, which tells us which tenant sent this packet. This helps us disambiguate two tenants using the same address, and also prevents packets from being sent to a different tenant.

In the second encapsulation step, we add an underlay network header, which tells us the physical server address corresponding to the virtual IP destination.



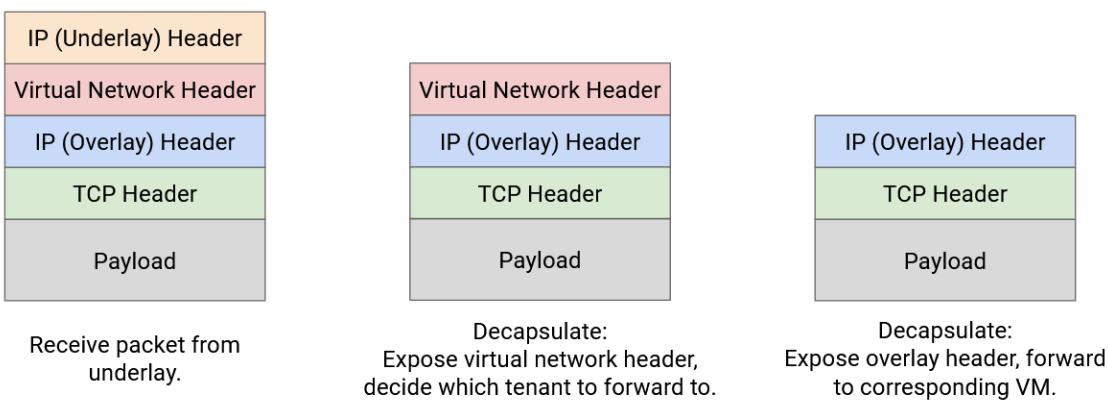
The layers of abstraction still hold when we stack encapsulations. The underlay network doesn't need to know that multiple tenants are in the same datacenter. The underlay network just looks at the outermost header for a physical server address, and forwards the packet accordingly.

The decapsulation step works in reverse order. The virtual switch on the destination server receives a packet with two extra headers.

In the first decapsulation step, we remove the outer underlay header. This is no longer necessary since the packet has reached the destination physical server.

In the second decapsulation step, we use the virtual network header to decide which set of VMs we should think about. The physical server might have VMs for multiple tenants, and this helps narrows down to a single tenant.

Finally, we use the innermost IP header to send the packet to the correct VM in the correct virtual network.



Note: With encapsulation, we have to be careful when reading the 5-tuple (IPs, ports, and protocol) for load-balancing packets across multiple paths. Fortunately, modern router hardware is good at parsing packets to understand where the relevant headers are located in the packet, even if additional headers are inserted.

In practice, many different protocols exist for encapsulation. We could use IP-in-IP to support two IP headers (one for overlay, one for underlay).

MPLS is a simple header for adding a label that identifies a service (e.g. a virtual network, a tenant). This can be used to add encapsulation for multi-tenancy.

As datacenters have become more popular, many other protocols like GRE, VXLAN, and GENEVE have been developed. Most of these work over IP, so these custom protocols are the inner overlay header, and regular IP is the outer underlay header.

Software-Defined Networking

Why Software-Defined Networking?

Previously, we saw how routing protocols can be adapted to work in datacenter contexts (e.g. equal-cost multi-path). What if we want to optimize our routing protocols even further for our specific network's constraints and use cases? The standard routing protocols might no longer work.

In this section, we'll explore **software-defined networking**, a totally new paradigm for thinking about routing and network management. In the context of routing, the SDN architecture involves having a centralized control center compute routes and distribute them to individual routers. We'll see how SDN works in the context of datacenters and the wide-area network, and discuss benefits and drawbacks of this new approach.

Brief History of Software-Defined Networking

Although we'll be looking at SDN as a new approach for specialized routing protocols, the SDN paradigm was originally designed in response to headaches at the management plane.

Recall that the management plane is critical for network operation. Routers can't do anything unless someone configures them (e.g. assigns costs to links) and tells them what to do (e.g. what routing protocol to run). Also, we need routers to report errors to keep the network up and running. A lot of this management work has historically been done manually.

Even though the management plane is so important, there's been relatively little focus on innovating it. At the control plane, we've seen lots of different routing protocols, but the way we configure and control routers has evolved more slowly.

Over the Internet's history, there's been a slow evolution toward using scripts to programmatically interact with the network. These scripts take jobs that the operator would manually do, and implement them in code (without much intelligence). For example, scripts allow automating the process of adding routers and links to the network. A script for repairing the network might say: if a router fails, check that it's actually failed, reboot it, and if it's still not fixed, report to the operator.

Despite the progress, these management systems have been the bottleneck for network operations for a long time. We still might have to wait for human intervention every time a new router is added.

In 2005, a paper by Albert Greenberg et. al. described the problem by saying: "Today's data networks are surprisingly fragile and difficult to manage. We argue that the root of these problems lies in the complexity of the control and management planes."

In response to these problems, researchers began thinking about different ways to run a network system. This led to more radical proposals that reimaged the fundamental design of routers.

The concepts we'll see were first considered in 2003, though they didn't gain much momentum at the time. Frustrations with network management accelerated the development of new management paradigms. By 2008, there was more momentum, leading to the OpenFlow switch interface (which we'll see soon).

By 2011, it was evident the industry was moving in this new direction, and the Open Networking Foundation

(ONF) was established by major network operators (Google, Yahoo, Verizon, Microsoft, Facebook) and vendors (Cisco, Juniper, HP, Dell). Nicira, the SDN-focused startup that developed the OpenFlow interface, was a \$40 million startup in 2012.

Routers are Vertically Integrated and Standardized

If we wanted to reimagine the design of routers, how would that be implemented in practice? How do technologies on routers change over time?

If your network needs a router, you'd probably purchase one from a major equipment vendor like Cisco or Juniper. In order to ensure that routers are compatible with each other, all the major equipment vendors build their routers according to some pre-defined standards.

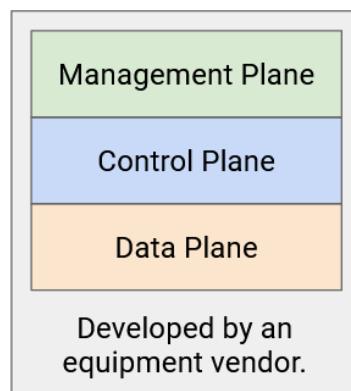
This business model can make innovation and experimentation with new approaches difficult. Suppose you had a new idea for a routing protocol. You would need to get the protocol approved by a standards body, which could take years. Then, you'd have to wait for the vendors to upgrade their manufacturing to conform to the new standard.

Standardization also makes routers less flexible for users implementing custom solutions. If you have a problem specific to your network, but no one else has this problem, your solution probably won't be adopted by the standards body. Vendors want to make routers that satisfy everybody's needs, and they won't necessarily implement a solution that's perfect for you, if nobody else wants it.

On the other hand, standardization also means that if others have a problem that you don't, the router might come with a solution to their problem, even if you don't need it. This can make routers unnecessarily complex for the purposes of your specific network.

Standardization also makes experimentation and research difficult. If you want to try a new idea to see if it works, you might not be able to buy routers that can implement your new idea. Vendors don't want to build experimental products, intended for one specific customer, that might not even work.

Another major obstacle to innovation and experimentation is routers being **vertically integrated**. The router you buy already has the functionality for all three planes wired on the chips. There's no modularity that would let you swap out just the control plane by itself.



Innovating Routers

If we did want to innovate routers, what could we innovate at each plane, and what kinds of pre-existing standards would we be working with?

The data plane is standardized by IEEE (electrical engineering group) and requires everyone to strictly follow the standards. If two routers from different vendors are connected, we have to make sure both sides are sending bits along the physical wire in the same consistent format.

Data plane innovation is usually driven by the demand for higher-bandwidth routers, and new features are not often introduced. This development happens quite slowly, in 2-3 year increments, because we have to solve physical hardware problems and design chips for increased bandwidth. Since the core data plane features are relatively stable, router innovation is not really focused on the data plane, and it's okay that the development cycle is slow.

The control plane is standardized by the IETF (the network group behind RFCs). Vendors sometimes add their own extensions, though the core features are mostly standardized. For example, we assume that every router (even if they're from different routers) are following the same routing protocol.

Control plane innovation (e.g. new routing protocols) can take several years to be adopted. You might have to submit an RFC draft proposal, and the community may spend some time discussing the proposal before agreeing on its terms.

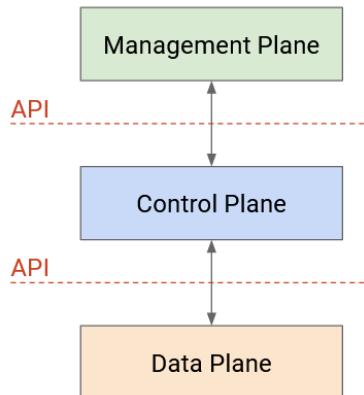
The management plane is also standardized by the IETF, though it's much less standardized. Different operators can use different software to configure their routers, and we don't really need different vendors to agree on some standardized software. Because this plane is only loosely standardized, many different approaches with different features exist.

In summary: The data plane is standardized (but we don't really have new features in mind), the control plane is standardized (but we want to try new solutions), and the management plane is not really standardized.

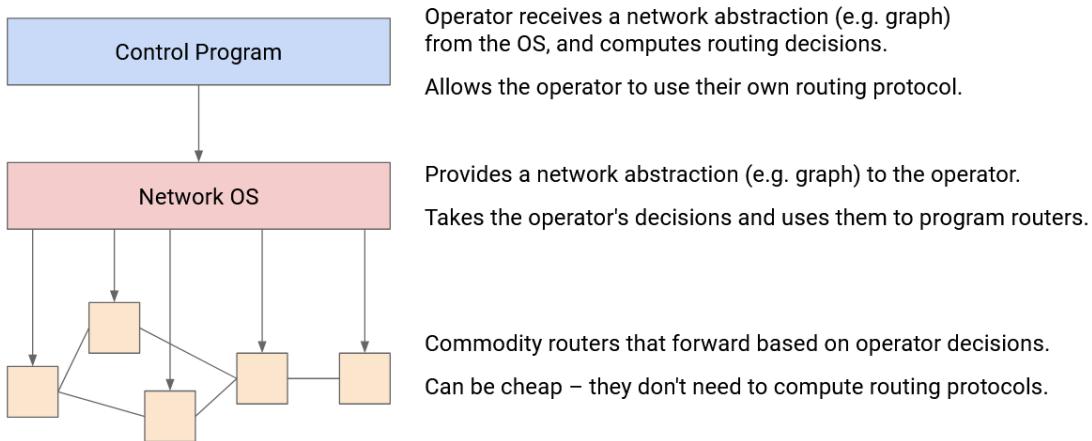
Radical Idea: Disaggregating Routers

Standardization and vertical integration were making it difficult to innovate and experiment. This led to the radical idea of disaggregating routers by splitting the planes into different layers of abstraction. Instead of buying a single router with all three planes, we could now buy data and control plane functionality separately. This allows us to change layers independently from each other.

In order to connect the three layers, we need an API between the layers of abstraction. In a vertically-coupled router, we don't care how the data plane and control plane talk to each other. However, if we buy the data plane separately, and we want to design our own custom control plane on top, we need an interface to interact with the data plane.



An even more radical idea is to stop thinking about the three planes in terms of only the router, and instead design a new system architecture that naturally splits up the data plane and control plane.



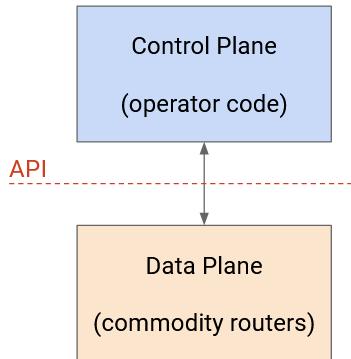
At the bottom, we have commodity network devices. You can think of these as buying just the data plane by itself. These routers receive instructions from the control program via the network OS, and simply forward packets according to those instructions. These routers don't need to think about routing protocols at all, so they can be cheaper.

In the middle, we have the network OS. You can think of this as the API connecting the data plane routers and the control plane program. The network OS provides an abstraction of the routers (e.g. as a graph) that can be passed up to the control program. Then, the control program can send routing instructions to the network OS, without worrying about how to program specific routers. The network OS can take those instructions and program them onto individual routers.

At the top, we have the control program. You can think of this as buying or implementing the control plane by itself. Here, the operator receives an abstraction of the network (e.g. graph) from the network OS, and can use that to write their own custom routing protocol. Then, the resulting routes can be passed to the network OS, which will program them onto routers.

OpenFlow API Format

OpenFlow is an API for interacting with the data plane of a router. The operator writes their own fancy code, separate from the router, that computes routes through the network. Then, those routes can be programmed onto the forwarding chip.



The OpenFlow paradigm is different from traditional routers, where the control plane is implemented in the router, and there's no clear API for programming custom routes onto the forwarding chip.

The OpenFlow API defines a **flow table** abstraction to describe routes and forwarding rules. The operator code can output any rules and routes it wants, and install them on the router, as long as they're in the flow table format.

The basic building block of the API is a flow table, which you can think of as a generalized version of a forwarding table. Each flow table consists of key-value pairs, just like a forwarding table. The key specifies what to **match** the packet against. This could be a destination prefix, an exact destination, a 5-tuple, or other relatively simple matches. The corresponding value specifies what **action** to set when a packet matches. The action could be sending the packet to a next hop (like a forwarding table), but could also specify fancier actions like adding an extra header.

The output format is a sequence of one or more numbered flow tables, where each table has its own different match-action entries. These flow tables can then be programmed onto the forwarding chip.



When a packet arrives at a router, it is checked against each table in order (e.g. Table 0, Table 1, Table 2, etc.), and when there's a match, we write down the corresponding action (without executing it yet). Eventually, once the packet is checked against the final table, any action(s) we wrote down are applied to the packet.

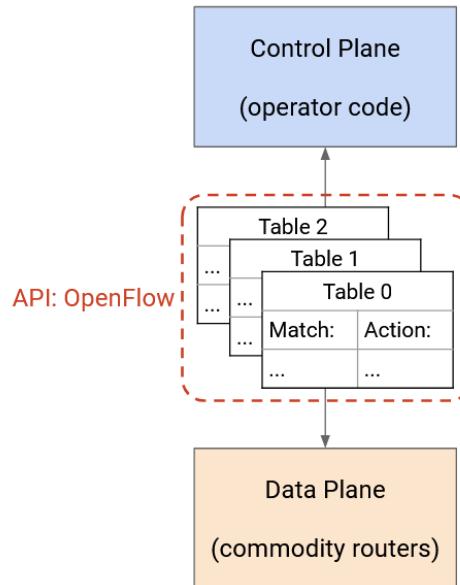
There are also special actions for skipping to later tables, which we can use in rules like: If the source port matches this number, skip to table 5 to set additional actions.

Example Flow Table	
Match:	Action:
Destination IP is in 192.168.0.0/16	Forward to next-hop R2
5-tuple is exactly (1.1.1.1, 2.2.2.2, TCP, 255, 53)	Add encapsulation and forward to R3
Source IP is exactly 10.2.15.3	Decapsulate and forward to R3

The operator can run any code they want to generate flow tables, and the flow tables can be more general than a destination/next-hop forwarding table. However, the rules (match/action pairs) that we generate are still constrained by the specialized forwarding chip hardware. The forwarding chip is optimized for speed, and probably can't handle complex match rules like "if the TCP payload is in English, set this action."

As a result, the flow tables we see in practice end up looking pretty similar to the tables we've already seen. Common match rules include longest prefix matching on IP destinations, 5-tuples to identify flows, and exact matches on encapsulation headers (e.g. MPLS).

If the forwarding rules aren't so different, why use OpenFlow at all? Remember, the main advantage is that it gives the operator total freedom at the control plane. We're not limited to distance-vector or link-state protocols anymore.



Benefits of a Flexible Control Plane

Our new architecture gives the operator flexibility to implement their new routing protocol at the control plane. What are some benefits of this approach?

The operator can implement custom routing protocols best-suited for the operator's specific needs. The operator is no longer constrained by standards bodies and vendors.

Flexibility also gives us an opportunity to simplify. For example, if the standardized protocol includes features we don't need, we don't have to implement them in our custom solution. Simpler protocols can have less code and simpler code, which might allow for easier development and maintenance of that protocol.

Finally, a flexible control plane enables centralized computation of routes at the control program, instead of distributed across multiple routers. Centralization comes with several benefits as well.

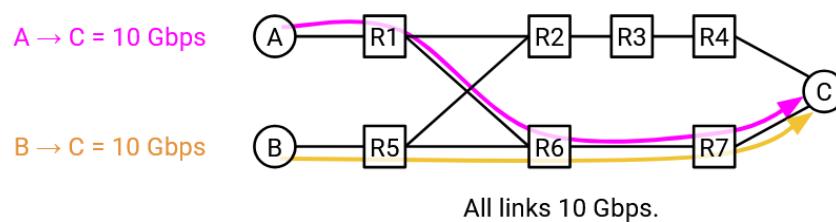
Centralization can result in more intelligent routing decisions that lead to excellent performance. In a 2013 report from Google, engineers who deployed an SDN architecture noted that "centralized traffic engineering service drives links to near 100% utilization, while splitting application flows among multiple paths to balance capacity against application priority/demands." A 2013 paper from Microsoft describes using an OpenFlow controller to "achieve high utilization with software-driven WAN."

More intelligent routing decisions can help optimize other criteria besides performance, that a standard routing protocol can't easily optimize. For example, a US government network might implement a geofencing rule that says, don't send traffic via links that are in Canada. Or, a broadcast TV network might want to optimize for path diversity to increase reliability. We can enforce that two flows travel via paths that don't share any links, so that if a link goes down, only one of the flows is affected. The two paths can serve as backups for each other.

Centralization can also make it easier for routing protocols to converge. In a distributed protocol, if the network changed, the routers have to coordinate to converge on a new routing state. In this centralized model, if a link fails, that router could tell the boss, and the boss could recompute routes and install the new routes on the routers.

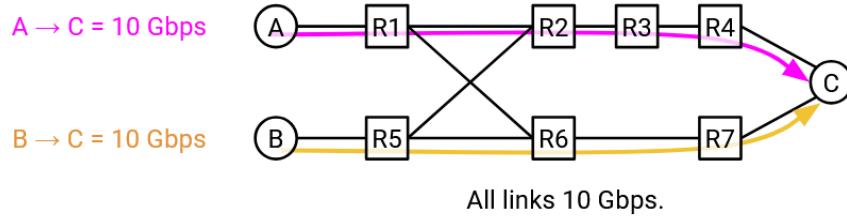
Traffic Engineering

A flexible control plane allows us to perform **traffic engineering**, which means we can route traffic in a more intelligent and efficient way than a standard distributed routing protocol could.



Suppose there are two connections, S1-D at 10 Gbps and S2-D at 10 Gbps. If we just ran standard least-cost routing, both flows would send traffic along the bottom path. The bottom path would be congested (20 Gbps on 10 Gbps link), while the top path's bandwidth is sitting there unused.

With a more intelligent routing scheme, we could send S1-D traffic along the top path, and S2-D traffic along the bottom path. Using traffic engineering, we've forced some packets to take a longer route, in order to better utilize the bandwidth in the network.

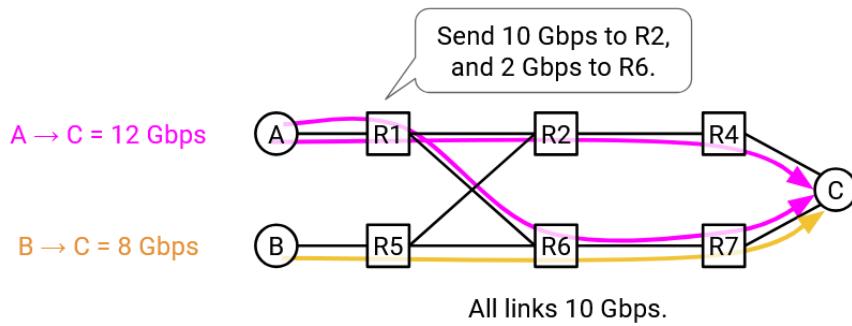


To compute these routes, we can modify least-cost routing, and instead enforce that traffic should be on the shortest path that has sufficient capacity. We can also enforce other constraints instead of capacity, such as latency. The resulting algorithm is called **constrained Shortest Path First (cSPF)**.

Now, suppose that S1-D needs 12 Gbps, and S2-D needs 8 Gbps. cSPF will send the flows along different paths to maximize bandwidth, but S1-D is sending 12 Gbps over a 10 Gbps link.

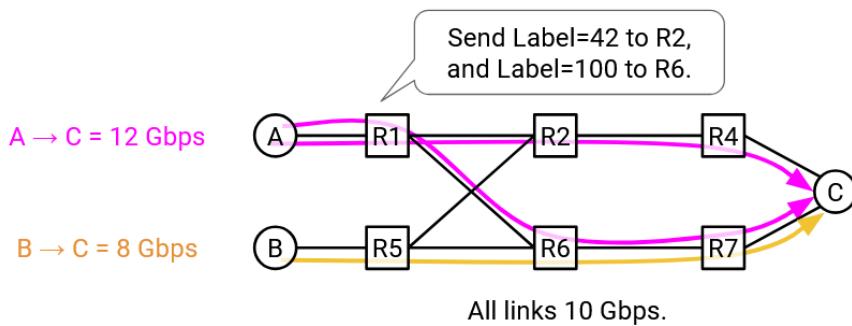
To fix this, our traffic engineering can be even more intelligent, and split traffic in a flow across different paths. S1-D can send 10 Gbps of its traffic along the top path, and the remaining 2 Gbps along the bottom path.

Again, our traffic engineering allowed us to implement custom logic that resulted in better utilization of the network capacity.



How do we actually implement split paths through the network, using the OpenFlow API from earlier? Remember, our routing decisions should still follow simple rules that forwarding tables can understand.

One approach is to use encapsulation. At the sender, we can add rules to add an extra header, where some packets get label 0, and the rest get label 1. This label tells us which path to send the traffic along.

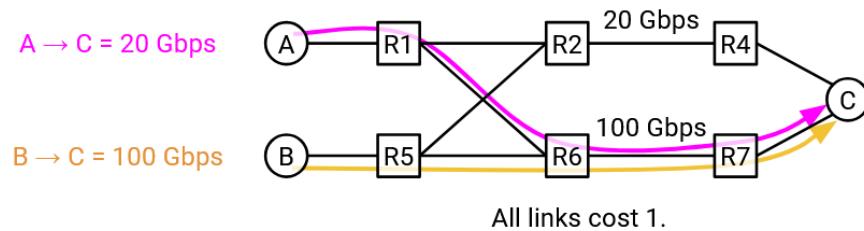


Now, at R1, we can add simple rules to route label 0 packets upwards to R2, and label 1 packets downwards to R3. This idea can be applied in addition to the other rules we had for constrained least-cost routing (e.g. the flow tables might have other entries for other destinations or other flows).

Centralized Traffic Engineering and Globally Optimal Decisions

One major difference in the SDN model of custom routing protocols is centralization. In the original model, every router was running its own routing protocol. Now, we can have a single computer outside of the routers compute all the routes, and then use the flow table API to install those routes on the routers.

Centralization allows us to make **globally optimal decisions**. In a distributed protocol, each router is making the best decision for itself, but that might not be the best decision for other routers. In the centralized model, the boss can use its global view of the network to decide what's best for everybody, and tell the routers to follow that decision.



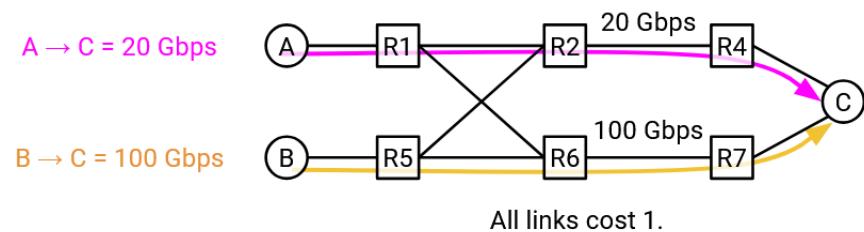
Consider this network with two flows, S1-D at 20 Gbps, and S2-D at 100 Gbps. Assume we haven't implemented support for splitting a flow onto multiple paths.

Suppose the 20 Gbps S1-D flow starts first. Using constrained shortest path first, S1 could choose to use the bottom path. From the perspective of S1, this is a locally optimal decision (top and bottom paths both equally good).

Later, the 100 Gbps S2-D flow starts. Now, using constrained shortest path first, S2-D doesn't have any single path that meets its demands. The top path (20 Gbps) and bottom path (80 Gbps) both have insufficient capacity.

The key problem here is, each individual router made its own decision independently, without coordination.

By introducing a centralized controller, the controller can look at the overall network structure and the demands of each flow, and assign paths to each flow more intelligently. The resulting decision is globally optimal, and increases network efficiency.



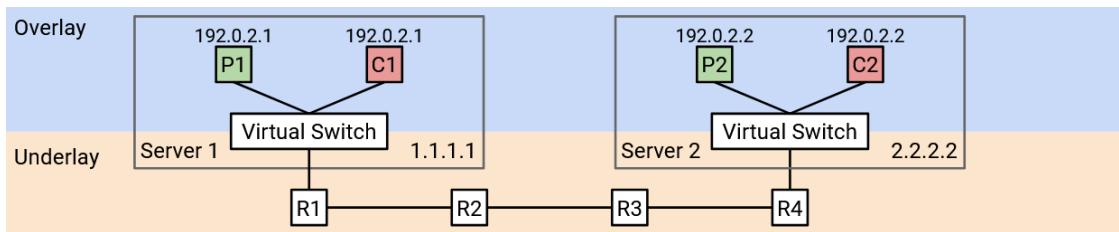
Centralized traffic engineering can make even more intelligent routing decisions, depending on what the operator wants to optimize. For example, we could classify flows as high-priority or low-priority, and make decisions that optimize both network utilization and the needs of different applications.

SDN in Datacenter Overlay

In the previous section, we saw that virtual switches can apply encapsulation to connect the overlay and underlay networks. Given a virtual address, we can add a header with the corresponding physical address, which allows the packet to be sent along the underlay network. But, how do we know the mapping between virtual addresses and physical addresses?

We also saw that encapsulation can be used to support multiple tenants in a single datacenter, each running their own private network. Switches can add headers with a virtual network ID. But, how do we know which virtual network ID to use?

A centralized SDN controller can be used in the datacenter to solve these problems. Each tenant can operate its own controller. When a new VM is created, the SDN learns about its virtual and physical addresses. Then, the SDN can update the forwarding tables in the other virtual switches, adding encapsulation rules with the new virtual/physical address mapping.



For example, suppose Coke VM 2 is created with virtual IP 192.0.2.1 and physical IP 2.2.2.2. The SDN knows Coke VM 1 lives on physical server 1.1.1.1, so it can go to the virtual switch on 1.1.1.1 and add an encapsulation rule for the new Coke VM 2.

The flow table at 1.1.1.1 might say: If you receive a packet with destination 192.0.2.1, add a header with Coke's virtual network ID of 42. Also, add a header with the corresponding physical address 2.2.2.2. Then, send the packet along the underlay network.

Benefits of SDN in Datacenter Overlay

Why might we use a centralized SDN architecture to support virtualization and multi-tenancy in datacenters, instead of a more standard routing protocol?

The centralized SDN architecture allows us to cleanly split the overlay and underlay networks into two scalable layers. In a traditional architecture, the routers in the underlay network would have to process the custom encapsulation headers (e.g. virtual network IDs). SDN allows the underlay network to remain simple, without thinking about virtualization or multi-tenancy.

Centralization gives us a simple way to implement the control plane at end hosts, without any complicated routing protocols. The controller learns about a new host and updates the other hosts accordingly. Without

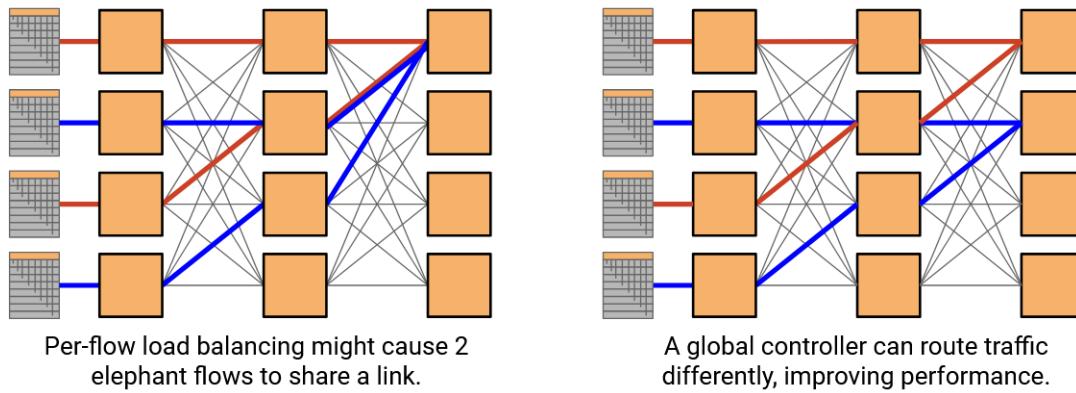
a centralized controller, we might need some complex distributed scheme to figure out which encapsulation headers to add.

This SDN architecture also shows us why overlay networks can scale well. The SDN controller for a tenant only needs to know about the VMs belonging to that specific tenant. By contrast, if we used a traditional architecture, a new Coke VM might have to be advertised to all the other VMs, even Pepsi VMs.

SDN in Datacenter Underlay

The datacenter underlay is a physical network, just like any other network, although with a special topology. Many general-purpose network challenges, like achieving high utilization of links, also apply to datacenter underlay networks. That means we can apply SDN to the underlay network as well.

SDN at the underlay network can help us efficiently route packets through the datacenter. For example, the operator might want to send mice flows along links with small delay, and elephant flows along links with high bandwidth.



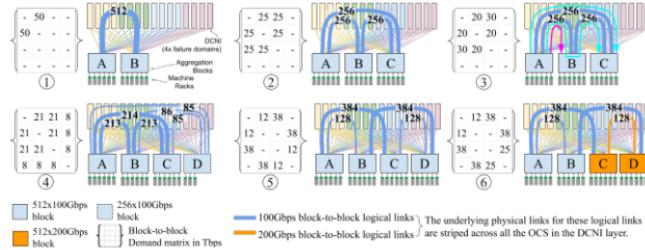
In our underlay Clos network, per-flow load balancing (hash 5-tuple to choose a path) could still send multiple elephant flows along the same path. Even if two elephant flows used different paths, the paths might share links, and those links might become congested. An SDN controller could solve this problem by coordinating the flows and placing them onto non-overlapping paths.

Jupiter Evolving: Transforming Google's Datacenter Network via Optical Circuit Switches and Software-Defined Networking

[Link](#)

Leon Poutievski et. al. (Google)

ACM SIGCOMM 2022



This 2022 Google paper describes eliminating layers in the Clos network (fewer links, cheaper datacenter) by using SDN to route traffic more intelligently.

Hyperscale datacenters often use SDN in both the overlay and underlay networks. These are usually implemented as decoupled systems. There's one SDN thinking about the underlay, and a separate SDN thinking about the overlay.

SDN in Wide Area Networks

In addition to datacenters, SDN can be useful in general wide-area networks, especially when efficient utilization of bandwidth is critical. For example, in the traffic engineering example from earlier, imagine if our 10 Gbps links were undersea cables. There's no cheap way to add additional bandwidth, so optimizations have to instead focus on efficient utilization of the bandwidth we do have.

Drawbacks of Centralized Control

Centralization doesn't come for free, and has some drawbacks.

One drawback is reliability. In a traditional network, if a router fails, the routing protocol converges around the failure. The other routers can reroute traffic along other paths. By contrast, if the central controller fails, we don't have a way to update the network anymore, and the routers don't know how to adjust to changes.

Note: We've drawn the centralized controller as a single entity, but it doesn't need to be run on a single server. The control plane computation could happen across multiple servers, where those servers coordinate to operate in a logically centralized way. This is different from the original model, where routers coordinated but still made their own distributed decisions. This helps to avoid having a single point of failure in hardware, though the controller as a logical unit could still fail (e.g. bug in the code).

Centralization also introduces scalability problems. The controller has to make decisions for everybody, which can get expensive for large networks. By contrast, in a traditional network, each router only has to perform computations for itself.

Centralization could also introduce different types of complexity. In a traditional network, we could buy a router and connect it, and it more or less starts working right away. With a central controller, we

have additional infrastructure challenges. Where do we put this controller? How do we connect it to the individual routers in a reliable way?

This is an active area of research, including a project by Sylvia Ratnasamy and Rob Shakir (Berkeley CS 168 instructors).

SDN in the Management and Data Plane

We've seen SDN as a new way to implement the control plane. But, the initial frustration that led to the development of SDN was at the management plane.

It turns out, many of the design paradigms that SDN used at the control plane can also apply to the management plane. For example, we saw that SDN relies on well-defined, programmatic APIs (e.g. OpenFlow).

TODO ran out of time in SP24.

Host Networking

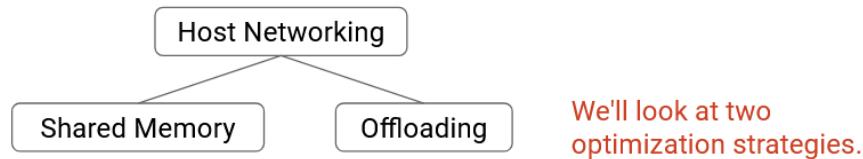
What is Host Networking?

Traditionally, the bottleneck of the network is inside the network infrastructure, not at the end hosts. However, in modern high-performance datacenters, as network performance demand continues to increase, the end hosts are struggling to keep up with the demand.

In particular, the CPU running network protocols like TCP is no longer able to deliver the high performance that the datacenter needs. CPUs are expensive, and delivering high performance means that the CPU is spending all its time running network protocols, with fewer resources allocated toward running the actual applications.

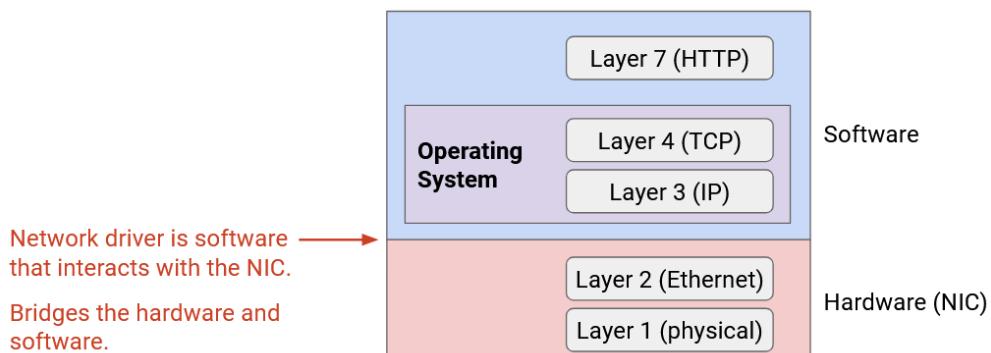
Also, the actual protocols that we've been running, like IP and TCP, are no longer able to meet modern high performance demands.

To solve these two problems, we turn to **host networking**, which involves optimizations at the end hosts (as opposed to inside the network).



Optimization: Shared Memory in User Space

Recall that at the end host, Layers 1 and 2 are implemented in hardware at the network interface card (NIC). Layers 3 and 4 are implemented in software in the operating system (on the CPU). Layer 7 is the application itself.

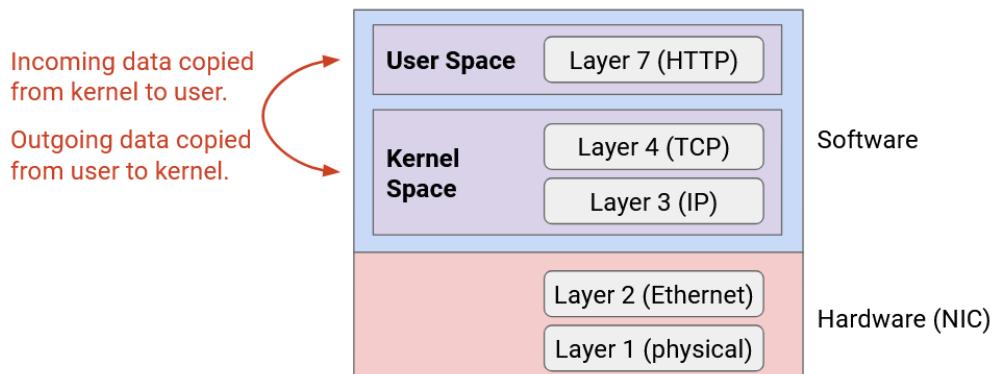


Recall from a prerequisite class (e.g. CS 61C at UC Berkeley) that modern computers are designed with virtual memory, so that each application gets its own dedicated address space, isolated from other applications. In particular, each Layer 7 application gets its own dedicated address space in **user space**. By contrast, the

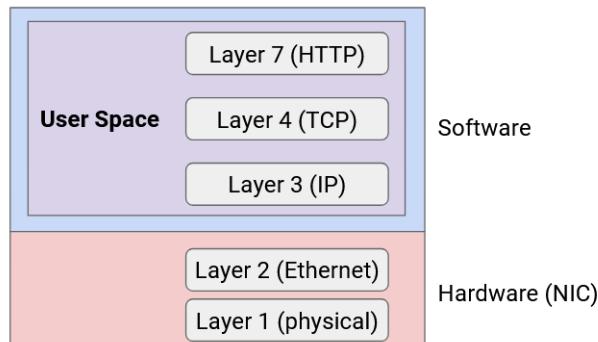
operating system itself runs in **kernel space**, which is a special part of memory that applications in user space cannot access.

This memory management model means that when we pass packets down the stack to send data, we are constantly copying data from user space to kernel space. Also, when we pass packets up the stack to receive data, we are constantly copying data from kernel space to user space. Copying bits between kernel space and user space is expensive and kind of pointless.

Another problem with this memory management model is, programming in kernel space is difficult. If we wanted to modify TCP and optimize it for our purposes, we would have to reach into the operating system and program at a very low level. Deployment and testing is harder and slower in the kernel space than in the user space.



To solve these two problems, we can move the networking stack (e.g. Layer 3 and 4 protocols) out of kernel space, and into user space. Now, Layers 3, 4, and 7 can all access a shared address space, and no copying back-and-forth is needed. Also, iterating and innovating in user space is now easier.



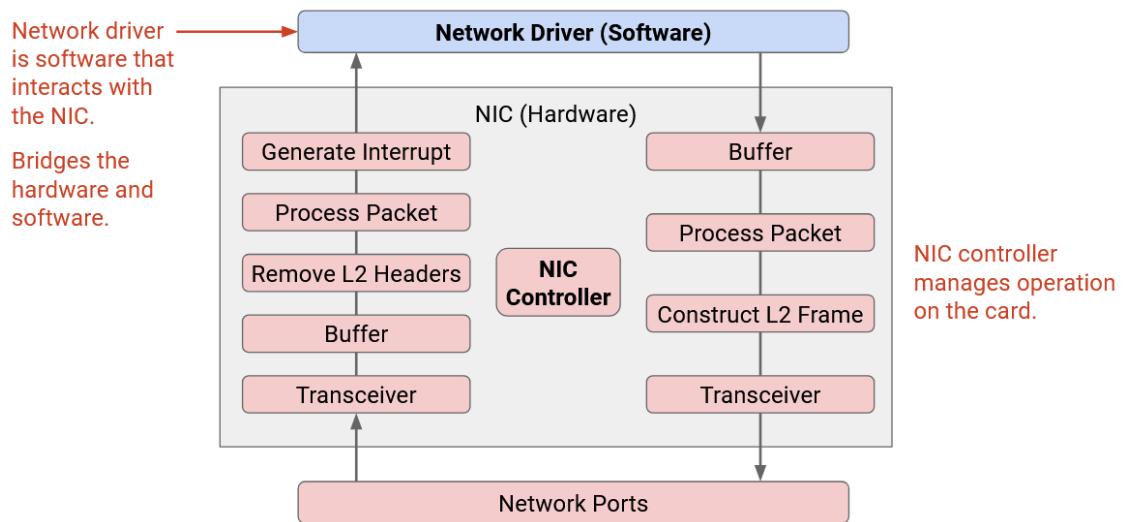
Using shared memory in user space helps us eliminate some extraneous work like copying back-and-forth, but it still isn't enough to make our hosts meet modern performance requirements.

Optimization: Offloading to NIC

CPUs are not fast enough to run network protocols (e.g. IP, TCP) at modern performance speeds. Also, using CPUs to run network protocols leaves less CPU resources for the applications themselves to use.

To solve this problem, we can offload the networking stack out of the CPU (software), and into the NIC (hardware).

The NIC is a natural place for offloading operations. Every packet has to pass through the NIC, so the NIC can do some extra processing and save the CPU from doing that work.



The **network driver** is a piece of software in the OS that programs and manages the NIC. The driver provides an API that allows higher-level programs in the OS to interact with the NIC. You can think of the driver as the bridge between hardware and software.

What are the benefits of offloading? It frees up CPU resources for the application to use. Also, specialized processing in hardware can be more efficient than processing on general-purpose CPUs. Here, efficiency refers to both speed and power consumption. Finally, running operations in hardware gives us not just lower latency, but also more predictable and consistent latencies. When running applications in software, the CPU has to schedule different processes, which can add unpredictable delay. (For example, if I have a packet to process, the CPU might have to finish its current job before switching over to processing my packet.)

Brief History of Offloading: Epoch 0

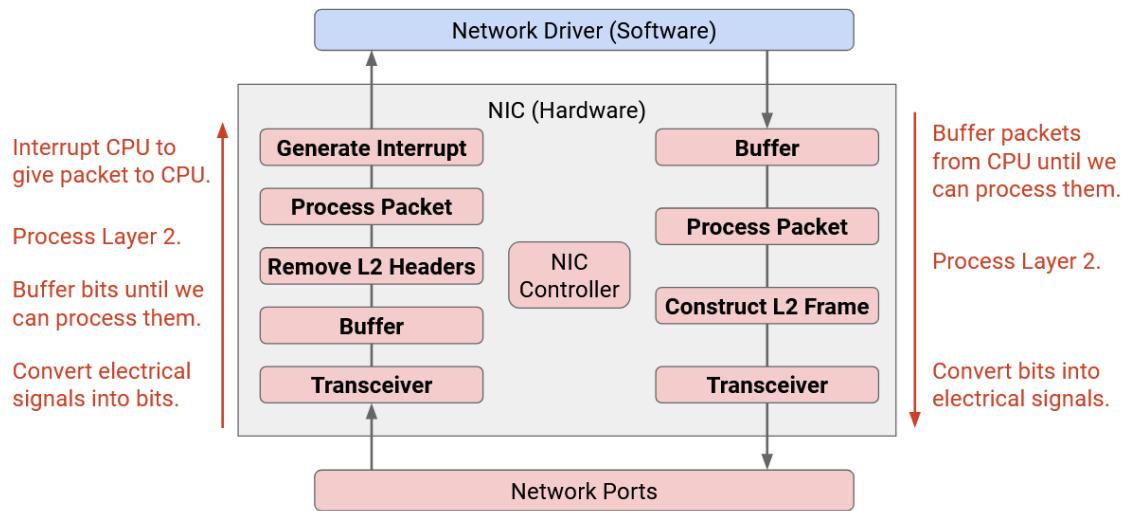
Offloading operations from the OS (software) to the NIC (hardware) is an active, ongoing area of research. There have been three epochs of development, where increasingly complicated operations have been offloaded to the NIC.

Epoch 0: Before any offloading, let's see what the NIC does in the standard networking stack we've seen so far.

The NIC has a central controller processor that manages operation on the card.

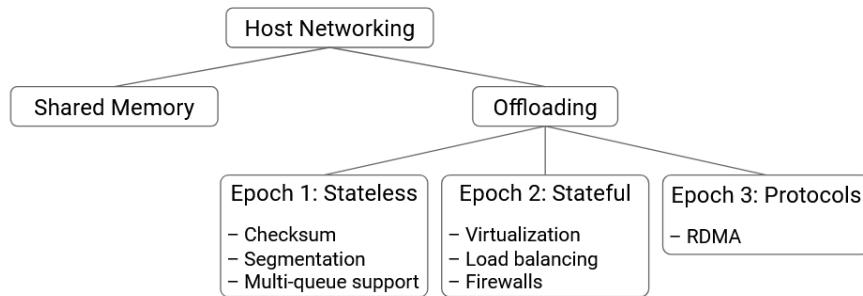
For incoming packets, the transceiver converts electrical signals into digital signals (1s and 0s) and puts those bits in a buffer. Then, the NIC reads bits from the buffer, parses them as Ethernet frames, processes the frame (e.g. verifies checksum), and removes the Layer 2 header. Finally, the NIC generates an interrupt to tell the CPU to stop what it's doing and collect the resulting Layer 3 packet for further processing.

For outgoing packets, packets from the network driver are placed in a buffer. The NIC reads bits from the buffer and processes them to construct Ethernet frames. Then, the frame is passed to a transceiver, which converts the digital bits to electrical signals.



In the standard networking stack, you can think of the NIC as a doormat that passes incoming packets to the OS, and sends outgoing packets for the OS, but performs very minimal processing on those packets.

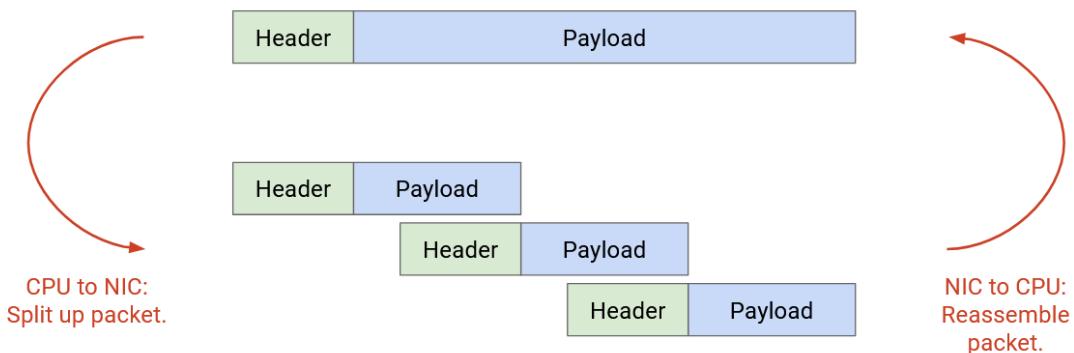
Brief History of Offloading: Epoch 1



The first operations that we tried to offload to the NIC are simple, stateless operations. These stateless operations can be done independently per packet, and the NIC doesn't have to remember any state across multiple packets.

One stateless operation we can offload is checksum computations, not just at Layer 2, but also at Layers 3 and 4. The NIC can validate these checksums (for incoming packets) and compute these checksums (for outgoing packets), so that the CPU doesn't have to.

Another stateless operation we can offload is segmentation. In our standard model, if the application has a huge file to send, then the OS is responsible for splitting up the file into smaller packets. Then, at the recipient, the OS is responsible for reassembling those packets. As an optimization, we can make the NIC deal with splitting up and reassembling packets. Now, the OS no longer has to deal with a ton of small packets, and can instead deal with a few large packets, which is more efficient (e.g. fewer headers to process).



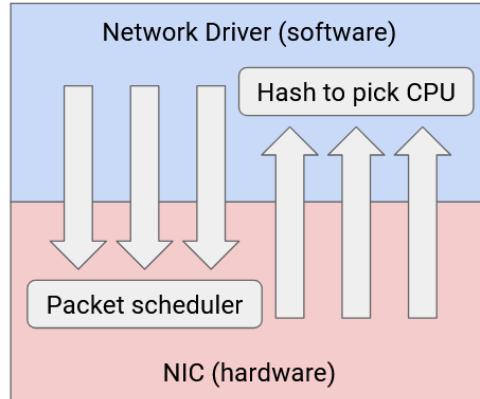
With segmentation, there's a trade-off between smooth connections and CPU efficiency. If the application hands large packets to the NIC, the CPU has less work to do. However, the NIC now gets large bursts of data, and the connection is more bursty. By contrast, if the application hands smaller packets to the NIC, the CPU has more work to do, but the NIC gets a steadier stream of data, and the resulting connection is smoother.

There are some challenges associated with aggregating small packets. What if an intermediate packet is lost? Then the NIC might have to pass up a bunch of small packets, and is unable to combine them into one big packet. What if some packets have a flag (e.g. ECN for congestion) set, and others don't? Should the resulting aggregated packet have the flag set or not?

The third stateless operation we'll look at offloading is multi-queue support. In our standard model, the NIC has one queue for outgoing packets, and one queue for incoming packets, and all applications share these queues. The network driver (in software) was responsible for load balancing, in case multiple applications or multiple CPUs were sending and receiving data.

We can instead offload this load balancing job to the NIC. Now, the NIC has multiple transmit queues, and multiple receive queues. For example, in a multi-processor system, each CPU can have its own dedicated transmit/receive queues. The NIC maintains all the queues in parallel, ensuring isolation and load-balancing between the different CPUs. The NIC can also prioritize certain queues over others.

Even though the NIC has multiple queues, it ultimately still has to send out all the packets along one wire. Therefore, the NIC needs some packet scheduler to decide which queue to send from next. The scheduler can be programmed to achieve the desired load-balancing behavior (e.g. if we want to prioritize one queue over another).



One challenge with multiple queues is mapping packets to queues. When a CPU has some data to send, which queue does it use? In particular, we want to make sure that all the packets within a single flow end up in the same queue (and not spread out across many queues). This helps us ensure that packets in a flow are sent in-order. Recall that in TCP, sending packets out-of-order works, but is bad for performance (e.g. receiver has to buffer out-of-order packets).

When processing incoming packets from the various receive queues, the NIC can hash the packet to decide which CPU will handle that incoming packet. Then, the NIC interrupts that CPU and tells it to process the packet. The hash-based behavior is similar to ECMP (Equal-Cost Multi-Path Routing), and helps us ensure that all packets in the same flow are processed in order by the same CPU.

Brief History of Offloading: Epoch 2

Later, we started to offload more complicated, stateful operations to the NIC.

The development of Epoch 2 has been driven by virtualization in datacenters, where multiple virtual machines run on the same physical server. For example, in virtualization, we needed a virtual switch to forward incoming packets to the appropriate VM. We showed the virtual switch running in software, but the virtual switch could also be implemented in hardware.

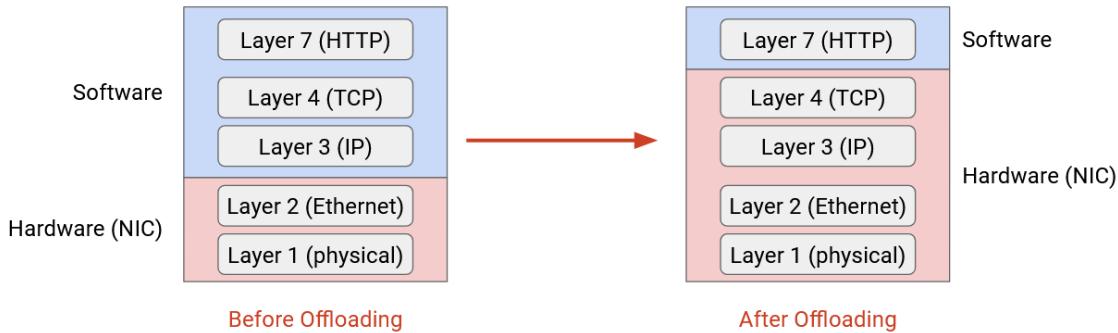
Firewalls and bandwidth management are another example of a stateful offload. In software, we can implement a firewall that enforces security policies (e.g. drop all incoming packets from this malicious IP). We can also enforce policies to manage bandwidth between users (e.g. User A can only send 100 packets per minute, any excess is dropped). These security policies could be checked by hardware instead.

To implement these stateful operations, we can use a match-action pair table, similar to the OpenFlow tables (from the SDN section). This API allows the software to program different policies onto the hardware, so that the hardware can process packets according to those policies. As we saw earlier, the match could be against a 5-tuple or some other header fields. The actions could be dropping packets, forwarding packets to a specific next-hop, or modifying headers.

Match	Action
Source IP = 10.1.8.9	Add encapsulation header, forward.
5-tuple matches (10.1.2.3, 50000, TCP, 24.1.3.0, 80)	Allow 100 packets per minute. Drop excess.
Source IP = 76.124.1.2	Malicious source. Drop.

Brief History of Offloading: Epoch 3

This is the current era of offloading. There are ongoing efforts to offload entire protocols, like TCP, out of the OS and onto the NIC. This epoch is being driven by even higher performance demands, especially with applications like AI/ML (artificial intelligence, machine learning) with high performance requirements.



Ideally, we'd like to let the application directly hand data to hardware, and let the hardware perform all the necessary network processing at Layers 4, 3, 2, and 1. The OS is entirely out of the picture, and all the network protocols are implemented directly in hardware.

While there's been some experimentation with offloading standard networking protocols like TCP onto the NIC, they haven't been deployed at scale. Instead, we've designed new networking protocols like RDMA, which are specially designed to allow implementation directly in hardware.

RDMA: Remote Direct Memory Access

RDMA offers an abstraction where Server A can directly access the memory in Server B, without the involvement of the OS or the CPU in either server. RDMA can be implemented directly in hardware, replacing the standard TCP/IP software networking stack.

Suppose that Server A wants to send a 10 GB file to Server B. In the standard networking stack, the CPU reads the file from memory, processes it (e.g. TCP/IP), and passes the resulting packets to the NIC. At the recipient, the NIC passes the packets to the CPU, which processes the packets, and writes the resulting file payload into memory. Notice that the CPU is involved in processing every single packet of the 10 GB file.



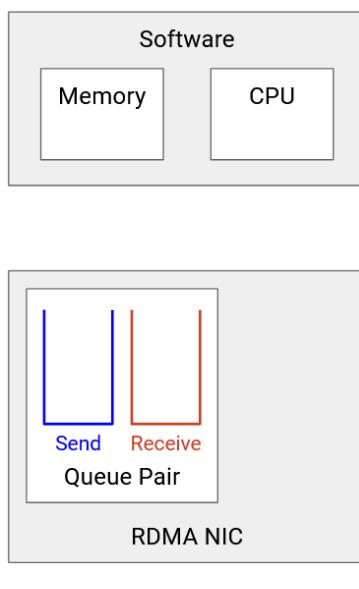
Without RDMA: CPU involved in data transfer.

In the RDMA abstraction, the NIC reads the file from memory and sends it out, with no CPU involvement. At the recipient, the NIC processes the incoming bytes and writes them to memory, again with no CPU involvement. Note that the CPU is still needed at the beginning to set up the transfer, and at the end to complete the transfer. But the bulk of the 10 GB file transfer is done without the CPU.

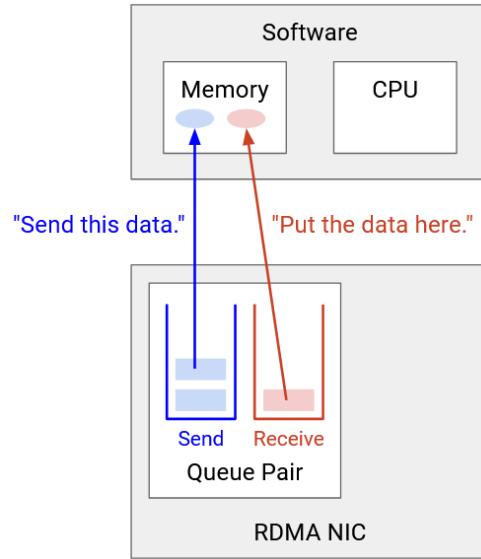


RDMA: CPU is minimally involved in transfer!

To use RDMA, programmers no longer use the socket abstraction. Instead, the main abstraction we use is the **queue pair**. The send work queue has all of the pending jobs where data needs to be transferred from me to somebody else. The receive work queue has all of the pending jobs where I need to receive data from somebody else. A single NIC can have multiple queue pairs, where each offers different service to the programmer. For example, one pair might offer reliable, in-order delivery, while another pair might offer unreliable delivery. A queue pair configured to be reliable and in-order is the closest to a traditional TCP connection.

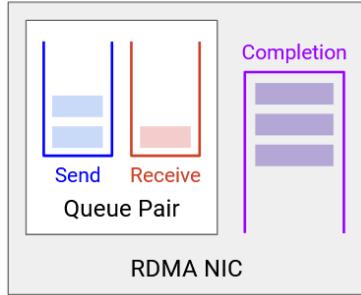
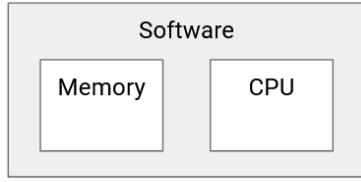


Each element in the queue is called a **work queue element (WQE)**. A WQE lets the application describe what work needs to be done. In English, the WQE in the receive queue might say, “Take 100 MB starting from address 0xfffff1234 on the remote server, and write them to address 0xfffff7890 in my local memory.” In code, the WQE is a struct that contains these instructions, e.g. a pointer to where we’re writing the received data.



Notice that the WQE abstraction gives the RDMA protocol a higher-level view of the application. In the TCP/IP stack, the network just sees a bytestream, but in RDMA, the WQE allows the application to describe the job in more detail (e.g. specifying the start and end of a block of data being transferred).

When a job is finished, the WQE is removed from the queue, and the NIC creates a new struct called a **Completion Queue Element (CQE)**, describing what happened to the job (e.g. success or failure). This CQE is stored in the Completion Queue, and sits there waiting until the application is ready to read the CQE and understand what happened to the job.



Notice that RDMA is asynchronous. Applications can add jobs (WQEs) to the queue pairs whenever they want, and the NIC will process the jobs in order. Similarly, when the job is done, a CQE is placed in the completion queue, and the application can read the CQE whenever it wants. (Contrast this with the TCP/IP stack, where incoming data triggers an interrupt for the CPU to handle that data.)

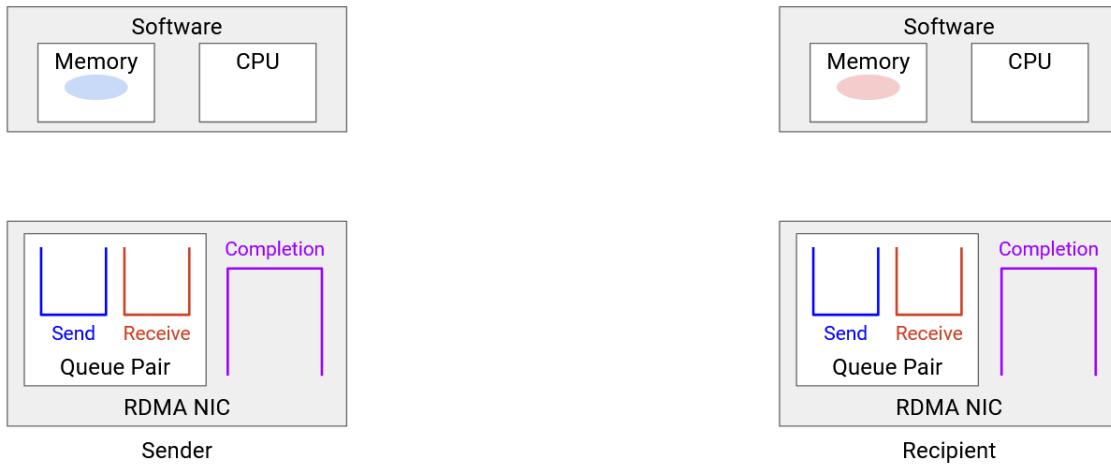
RDMA Example

RDMA can be used for several different operations between servers. Each operation has its own performance specifications (e.g. different latencies), and different semantics (e.g. different error messages). As an example, let's look at an RDMA send operation, where Server A reads a file from its memory, transfers that data, and Server B writes that file into its memory.

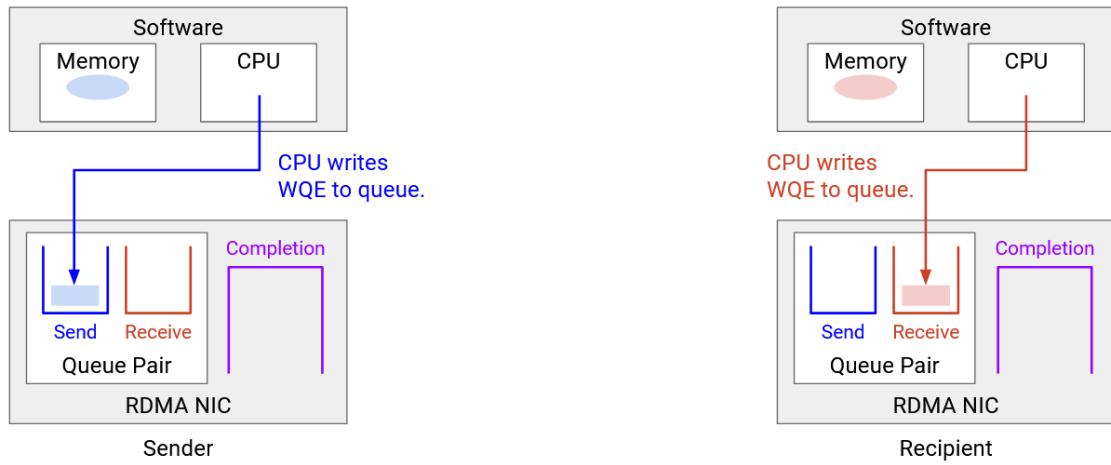
1. Each server designates some section of its memory to be accessible by the NIC for RDMA transfers. Server A designates the memory corresponding to the file as NIC-readable. Server B designates a blank buffer where it will receive the file as NIC-readable.

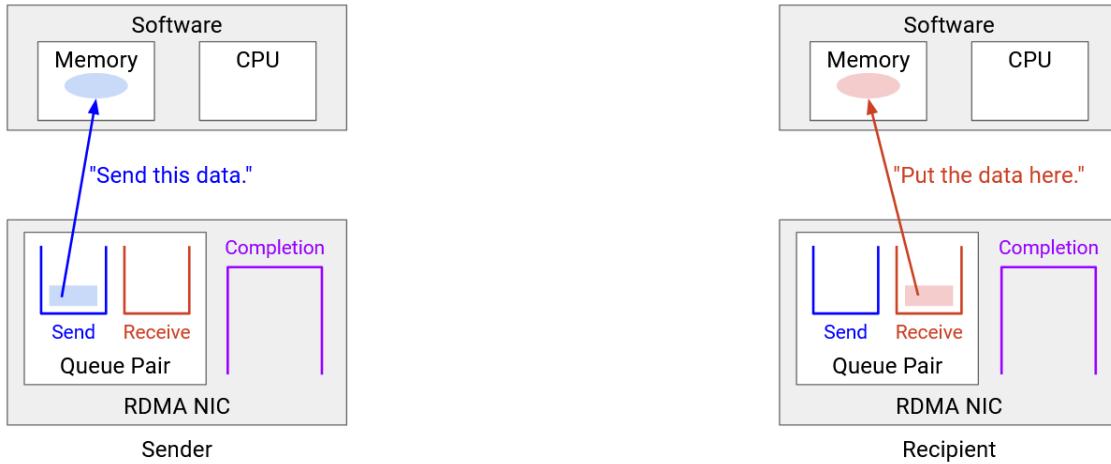


2. Each server sets up queues. Both NICs now have a send queue, a receive queue, and a completion queue. Note that this step can be done out-of-band, using a traditional protocol like TCP to coordinate between the two servers.

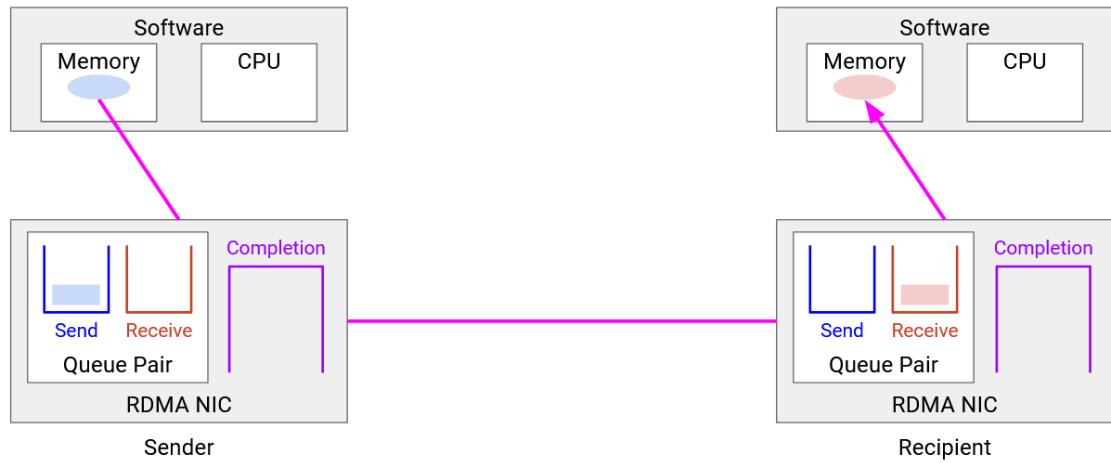


3. Server A creates a WQE in the send queue. This WQE contains a pointer to the file, indicating the data to be sent. On the other side, Server B creates a WQE in the receive queue. This WQE contains a pointer to the blank buffer, indicating where the received data should be written.

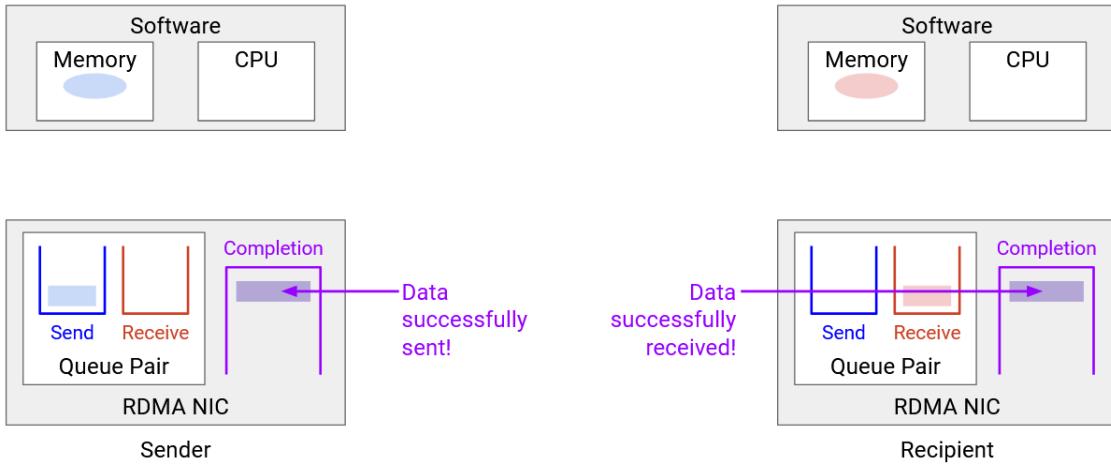




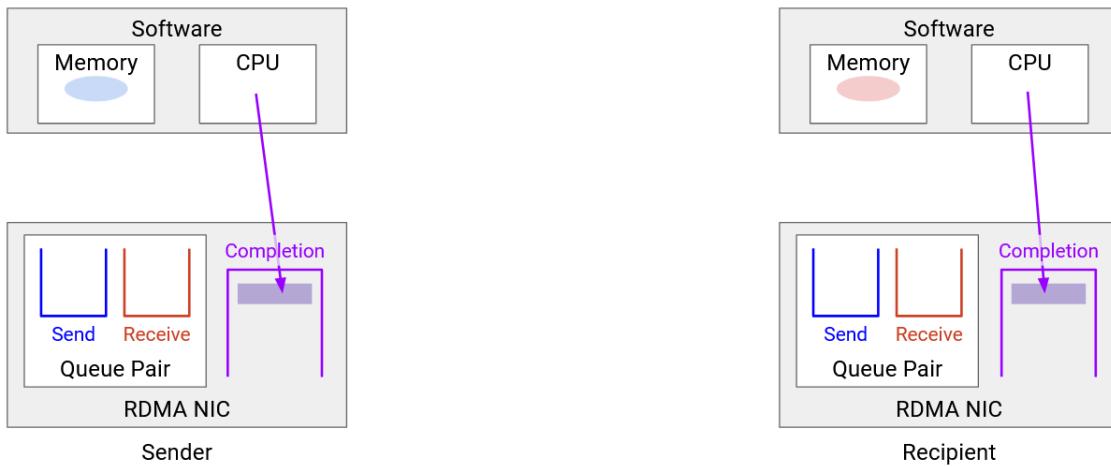
4. Once the transfer is queued on both sides, the data transfer can occur, with no involvement from software. The NIC handles everything, including reliability, congestion control, and so on.



5. When the transfer is done, the WQEs are removed from the queues. Both NICs generate a CQE, indicating that the transfer is done, and including any relevant status messages (e.g. error messages). Server A's CQE indicates that the data was successfully sent, and Server B's CQE indicates that the data was successfully received.



6. Eventually, the applications read the CQE to understand what happened to the transfer.



RDMA Pros, Cons, Applications

RDMA gives us high-performance data transfer (low latency, high bandwidth), and frees up the CPU for applications. However, RDMA doesn't come for free. RDMA requires specialized hardware and software, and is generally more complex than the traditional networking stack. Remember, RDMA is replacing the TCP/IP stack, so it has to implement all the TCP/IP functionality like reliability and congestion control, all directly in hardware.

RDMA also has some limitations, and usually works best in datacenters where the two servers are physically near each other. If the two servers are far away, the dominant delay comes from sending data across the network, and the time savings from RDMA are negligible. By contrast, if the two servers are nearby, the host processing packets could be the dominant delay, so RDMA gives significant time savings.

RDMA has been applied in many different settings that require high-performance, low-latency computing. Examples include scientific research, financial modeling, weather forecasting, machine learning, and search queries. In cloud computing, RDMA can be used to migrate a large VM from one physical server to another,

freeing up the CPU for customers to use. In AI/ML training, RDMA not only frees up the CPU and gives us low latency, but it also gives us predictable latency, which is important when different servers need to coordinate to train AI/ML models.

Implementing RDMA

Remember, RDMA replaces the TCP/IP networking stack, so RDMA is responsible for reliability, congestion control, and so on. There are two broad philosophies for how to implement this.

One option is to implement these features in the network itself, e.g. reliability at the switches. This is the idea behind Nvidia's InfiniBand.

Another option is to implement these features in the NIC, underneath the queue-pair abstraction. This is the idea currently being pursued at Google.

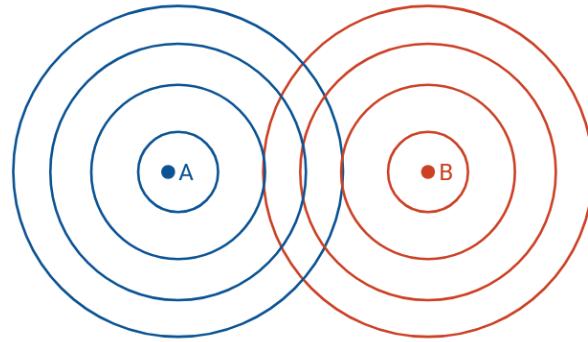
In both cases, the application and the OS in software gets the illusion of reliable, in-order delivery via the queue pair abstraction. The difference here is how RDMA actually implements those service guarantees.

Wireless

Introduction to Wireless Technologies

Wireless communication technologies actually predate the Internet. In the 1880s, the photophone (Bell and Tainter) attempted to send data wireless using a light beam. In the 1890s, the wireless telegraph (Marconi) attempted to send data using radio waves. Also in the 1890s, experiments with millimeter waves (Bose) were attempted, and today, millimeter wave is becoming an active area of research again.

Conceptually, you might imagine that wireless communication consists of invisible particles traveling along an imaginary link from point A to point B, but that's not actually very accurate. In reality, wireless communication is more like ripples on a pond. When you transmit data wirelessly, you create ripples that propagate outward and weaken over distance. If others are also transmitting data, the ripples can constructively and destructively interfere with each. The ripples can also reflect or refract against objects like boats on the pond, or the edge of the pond.



In this section, we'll look at four key differences between wired and wireless communications. The differences mostly affect Layer 1 (physical) and Layer 2 (link), with a few exceptions that we'll look at later (notably, breaking the end-to-end principle and implementing reliability at Layer 2 for performance).

Difference 1: Wireless is a fundamentally shared medium. Wired is not.

Difference 2: Wireless signals get weaker over longer distances. Wired signals do not.

Difference 3: Wireless environments can change rapidly. Wired environments do not.

Difference 4: Packet collisions are much harder to detect in wireless systems.

Difference: Wireless is a Shared Medium

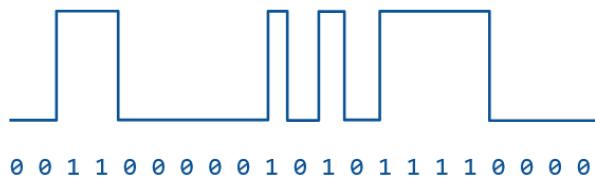
Difference 1: Wireless is a fundamentally shared medium. Wired is not.

Wired links are private (point-to-point) by default. Intuitively, a wire connects two devices. Creating a multi-point bus, where a single wire is connected to many devices, requires extra work. It's difficult for external signals to interfere with the signal on the wire (e.g. we can wrap a shield around the wire). Along the wire, we use electrical signals to transmit data (e.g. high voltage is 1, low voltage is 0).

Wireless links have the opposite properties. By default, wireless links are shared. Intuitively, if you transmit a signal, the signal radiates outwards in all directions. Creating a private point-to-point link between two hosts requires extra work. It's difficult to shield a signal from external interference. Instead of electrical signals, we encode bits using radio waves to transmit data.

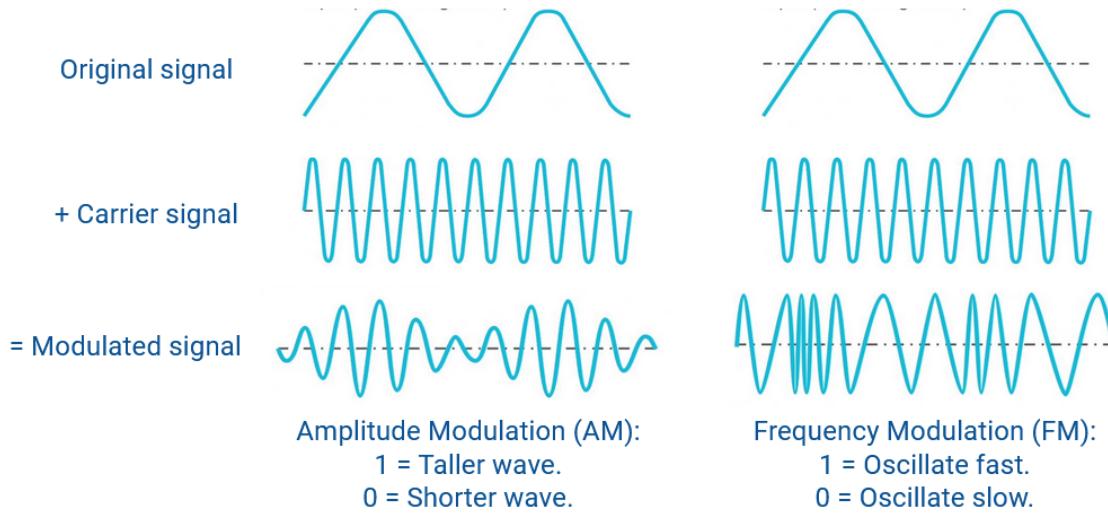
Encoding Data over Wireless Link

How do we encode data into electromagnetic waves at Layer 1? We could just take our sequence of 1s and 0s and draw it as a wave, but the resulting wave is probably low-frequency, and it turns out a low-frequency signal is weak and difficult to transmit.



Instead, we have to use **modulation** to transmit our data. We start with the carrier signal, which is just a constant-frequency wave (e.g. a sine wave). This wave carries no information, but it's high-frequency, so it's much easier to transmit. Then, we impose our data signal (also called the modulation signal) on top of the carrier signal. The resulting wave is high-frequency (easy to transmit), and also contains the data we want to send! Note that the receiver will need to take the modulated waveform and re-extract the 1s and 0s out of that waveform.

There are several strategies for modulating our data signal on top of the carrier signal. In amplitude modulation (AM), we vary the height of the carrier signal based on the input signal. To transmit a 1, make the sine wave tall, and to transmit a 0, make the sine wave short. In frequency modulation (FM), we vary the frequency (width) of the carrier signal based on the input signal. To transmit a 1, make the sine wave skinny (higher-frequency), and to transmit a 0, make the sine wave fat (lower-frequency). Other more complex modulation strategies exist, such as phase modulation, or a combination of amplitude and phase modulation.



Noise and Interference

Because wireless is a shared medium, we need to deal with noise and interference, which can corrupt the received signal. Noise always exists, even if nobody else nearby is transmitting data. (As an analogy, even if nobody around you is talking, there's still ambient noise from nature.) This ambient background noise is called the noise floor. By contrast, interference refers to other transmitters intentionally sending signals that interfere with our signal.

SINR (Signal to Interference and Noise Ratio) is a metric we can use to measure the quality of a wireless connection at the receiver. As the name implies, SINR is the power of the signal, divided by the power of the interference plus noise.

$$\text{SINR} = \frac{P_{\text{signal}}}{P_{\text{interference}} + P_{\text{noise}}}$$

SINR is a dimensionless quantity, since it's a ratio of two numbers. It can also be expressed in terms of decibels (dB), which is a logarithmic way to measure a ratio. At 0 dB, the ratio is 1, and when the SINR increases by 10 dB, the underlying ratio is 10 times greater (e.g. signal is 10 times more powerful, or noise/interference is 10 times weaker).

Ratio	Ratio in dB
1	0 dB
10	10 dB
100	20 dB
1000	30 dB
10000	40 dB

$$\text{SINR}_{\text{dB}} = 10 \cdot \log_{10} \left(\frac{P_{\text{signal}}}{P_{\text{interference}} + P_{\text{noise}}} \right)$$

What does this equation tell us? It tells us that if there's more noise, we have to transmit the signal with more power. It's also possible to employ coding gain (think: error-correcting codes), so that even if the signal is weak and gets mixed in with noise and interference, we're sending the signal with enough redundancy to allow the receiver to re-extract the signal.

The Shannon capacity gives us a theoretical limit on how much data per unit time we can send along a channel, given the amount of noise and interference along that channel. The equation works not just for wireless links, but also other types of links (e.g. wires).

$$C = B \cdot \log_2(1 + \text{SINR})$$

In this equation, B is the bandwidth of the channel. SINR is the signal-to-interference-and-noise ratio. C is the theoretical limit of how much data per unit time we can send along this channel, measured in bits per second. Note that in this equation, bandwidth is measured as the difference between the highest frequency and the lowest frequency that the receiver understands.

What does this equation tell us? It tells us that as bandwidth increases, we can send more data per unit time. It also tells us that as the SINR increases (stronger signal, or less noise), we can send more data per unit time. If we need a link with a specific target capacity (e.g. 1 Mbps), we can plug in the physical characteristics of our link into this equation to see if our link meets the desired capacity.

As an example, consider the plain old telephone system. This system has 3 kHz bandwidth, which means that telephones understand frequencies between 300 Hz and 3300 Hz. Also, this system has a SINR of roughly 20 dB, which translates to a ratio of 100 (0 dB = 1x, 10 dB = 10x, 20 dB = 100x, 30 dB = 1000x, etc.). Plugging these values into our equation, we get that $C = 4000 \cdot \log_2(1 + 100) \approx 20000$, which tells us that the telephone system can transmit roughly 20 kbps (kilobits per second).

Difference: Attenuation

Wireless signals get significantly weaker over longer distances. By contrast, wired signals do get slightly weaker over distance, but the effect is far smaller. In wireless systems, our design must account for attenuating signals, whereas in wired systems, attenuation is usually not a key design concern.

This creates a fundamental trade-off when designing wireless systems. We want to maximize performance by making our link accurate, fast, and long-range. But, we also want to minimize our resource use by conserving energy (e.g. laptop power) and using less of the frequency spectrum (which can be expensive to reserve). Unfortunately, a better signal requires more power or more frequency bandwidth.

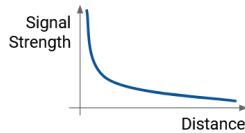
Free Space Model

One simple way to model signal attenuation is the free-space model (also known as the line-of-sight model), where we assume that the transmitter and receiver exist in a totally empty environment. Signals radiate outwards in all directions, with no obstacles (not even the Earth's surface).

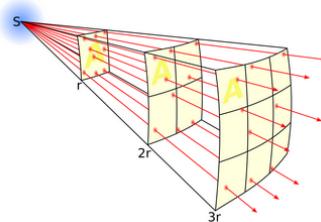
In this model, the power of the signal is inversely proportional to the distance between the transmitter and receiver. This is due to the inverse-square law:

$$P_r \propto \frac{P_t}{d^2}$$

In this equation, P_r is the power at the receiver, P_t is the power at the transmitter, and d is the distance between the transmitter and receiver. If we double the distance, the signal at the receiver is $1/4$ as strong. If the distance is 10 times larger, the signal at the receiver is $1/100$ as strong.



Intuitively, the inverse-square law applies here because the signal is radiating outwards in all directions. At any instant, the signal has radiated out to a sphere around the transmitter, and the sphere grows as the signal radiates further outwards. The surface area of a sphere with radius r is $4\pi r^2$, so as the signal propagates out, it's spread out over an area that grows quadratically (with the square of the distance). For example, when the distance doubles, the resulting sphere has 4 times larger surface area. Therefore, the signal is spread out over an area that's 4 times larger, so the signal is $1/4$ as strong.



Besides the distance, we also need to consider the antennas being used by the transmitter and receiver. This leads us to the Friis equation for measuring signal strength across a distance:

$$\begin{aligned} P_r &= P_t \cdot G_t \cdot G_r \cdot \left(\frac{\lambda^2}{4\pi} \right) \left(\frac{1}{4\pi d^2} \right) \\ &= P_t \cdot G_t \cdot G_r \cdot \left(\frac{\lambda}{4\pi d} \right)^2 \end{aligned}$$

In this equation, as before, P_r is the power at the receiver, and P_t is the power at the transmitter. G_t is the gain at the transmitter, and G_r is the gain at the receiver. λ is the wavelength, and it's used in this equation to represent the area of the antenna. d represents the distance between the antennas.

What does this equation tell us? To compute the signal strength at the receiver, we start with the signal strength at the transmitter, P_t . Then, we multiply by the gains of the two antennas, G_t and G_r . Intuitively, a higher gain means that the antenna is better at sending or receiving signals.

As we saw earlier, distance affects signal strength according to the inverse-square law, which explains the $\frac{1}{4\pi d^2}$ term.

Finally, the $\frac{\lambda^2}{4\pi}$ term relates to the aperture (think of it like area) of the receiver antenna. Intuitively, if you shine a light on a piece of paper, the light will hit that paper. If you use a larger sheet of paper, more light will hit the paper. The effective aperture (think: area) of the antenna can be computed as $\frac{\lambda^2}{4\pi}$, though we won't prove it here. Note that the $(4\pi)^2$ in the equation actually comes from two factors of 4π , one from the inverse-square law and one from the effective aperture equation.

We can also rewrite the Friis equation by dividing both sides by P_t :

$$\frac{P_r}{P_t} = G_t \cdot G_r \cdot \left(\frac{\lambda}{4\pi d} \right)^2$$

What does this equation tell us? The relative signal strength at the receiver (e.g. half as strong, or 1/100 as strong, as the signal strength at the transmitter) is a function of the antenna gains, the inverse of the square of the distance, and the effective aperture (think: area) of the antenna.

Yet another way to rewrite the same Friis equation is to take the log of both sides, allowing us to express the power and gain in terms of decibels:

$$P_r^{\text{dB}} = P_t^{\text{dB}} + G_t^{\text{dB}} + G_r^{\text{dB}} + 20 \log_{10} \left(\frac{\lambda}{4\pi d} \right)$$

The free space model is a useful theoretical model to measure the ideal signal strength at the receiver, though in practice, physical obstacles (e.g. the Earth's surface) prevent us from achieving this ideal value.

Link Budget

If signals get weaker over distance, how do we know if a link will actually work? In other words, how do we know if the receiver will actually detect an intelligible signal?

To measure if a link is viable, we can compute a link budget, which accounts for all gains and losses along the link.

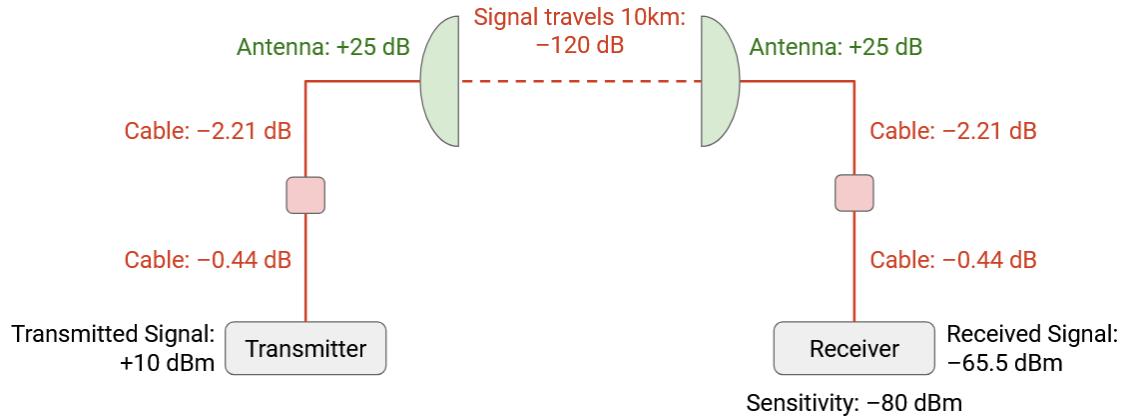
$$P_r^{\text{dB}} = P_t^{\text{dB}} + \sum \text{gains} - \sum \text{losses}$$

In this equation, P_r is the signal power at the receiver, and P_t is the signal power at the sender. All gains (e.g. a stronger antenna gain) add to our link budget, and all losses (e.g. path loss from long distance) cost us link budget.

Adding all gains and subtracting all losses tells us the signal strength at the receiver. We can compare this against the sensitivity of the receiver, which is the signal strength needed for the receiver to extract useful information. This comparison tells us our link budget. If the overall budget ends up positive, then this is a

viable link, and we're in the money. If the overall budget ends up negative, then this is not a viable link, and we're in trouble.

Notice that the link budget is computed in decibels, which are logarithmic. This allows us to use addition and subtraction instead of multiplication and division. For example, a gain of 1000x power is represented by adding 30 decibels, and a loss down to 1% of power is represented by subtracting 20 decibels.



Here's an example of computing the link budget. The signal power at the transmitter is 10 dB. The signal travels along a cable, a lightning arrestor (you don't have to know what this is), and another cable, losing 0.44 dB, 0.1 dB, and 2.21 dB along the way. Then, the signal is broadcast on an antenna, which gives us a 25 dB increase. Then, the signal travels across 10 kilometers of space, losing 120 dB along the way. Then, the signal is received by an antenna, giving us a 25 dB increase. Then, the signal travels along some more cables, losing 0.44 dB, 0.1 dB, and 2.21 dB, before finally reaching the receiver. If we add up all the gains and subtract all the losses, we can compute that the signal strength at the receiver is -65.5 dB.

We can now compare this signal strength against the receiver sensitivity, which is -80 dB. This tells us that the receiver can pick up any signals above -80 dB. Since -65.5 dB is above -80 dB, our link budget is positive, and our link should work!

The **link margin** is the difference between the signal strength at the receiver, and the receiver sensitivity. If we received a 30 dB signal, and our sensitivity lets us detect anything over 10 dB, we have a link margin of 20 dB. In the example from before, our link margin was 14.5 dB.

The link margin tells us about the quality of our link. If the link margin is negative, the link won't work, and the signals won't be received. A higher link margin is good because it means our signal is more reliable and more robust to interference and other issues.

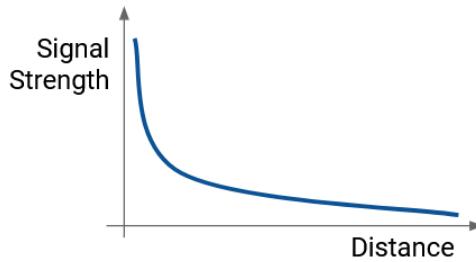
Difference: Environments Change

Wireless environments can change rapidly. The devices can move around. The environment could change (e.g. a physical obstacle moves in between the devices). Other communications could start interfering with our communication.

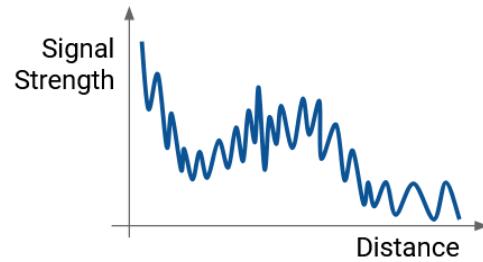
In the free-space model from earlier, we set the distance between the devices, d , to be a constant. But what

if the devices are moving? Also, we assumed there were no obstacles and no interfering signals in the environment. How does our model change in the presence of these factors?

In the free-space model, assuming the antennas stay the same (same gain, same aperture), we got a nice, smooth graph where signal strength decreased as distance increased. After accounting for a changing environment, the resulting graph of distance vs. signal strength is much more wobbly.



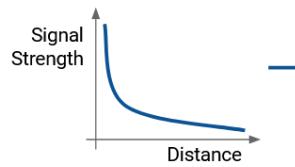
Free-space model:
Signal weakens over distance.



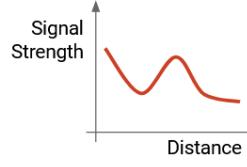
After accounting for obstacles:
Signal strength fluctuates!

This graph is actually the sum of three smaller graphs. Each one shows how a different characteristic of the environment affects the signal strength, as a function of distance. Notice that some characteristics change slowly as distance increases, while others change rapidly and erratically as distance increases.

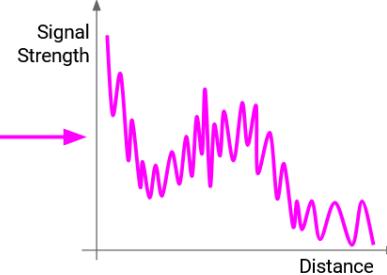
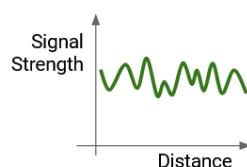
Free-space loss:
– Due to inverse square law.
– Fluctuates very slowly.



Shadowing:
– Due to obstructions.
– Fluctuates quickly.



Multipath fading:
– Due to signal colliding with itself.
– Fluctuates very quickly.



The first characteristic is free-space path loss. We've already seen this from the free-space model, which shows us that the signal strength decreases slowly and consistently over longer distances, according to the inverse-square loss.

The second characteristic is shadowing. This occurs when physical obstacles between the transmitter and the receiver block the signal. The signal must now be refracted or reflected to get around the obstacle, and the resulting signal at the receiver ends up weaker.

Depending on where the obstacles are located, the signal could get weaker or stronger as distance increases. For example, if I walk in front of a building, the signal will get a lot weaker, but if I eventually walk past the building, the signal might get stronger again.

The third characteristic is multipath fading. This occurs when waves reflect and refract on physical obstacles, which causes offset versions of the signal to arrive at the receiver. In particular, if a signal takes different paths of different lengths to reach the receiver, the signals might arrive out-of-phase with each other, causing interference.

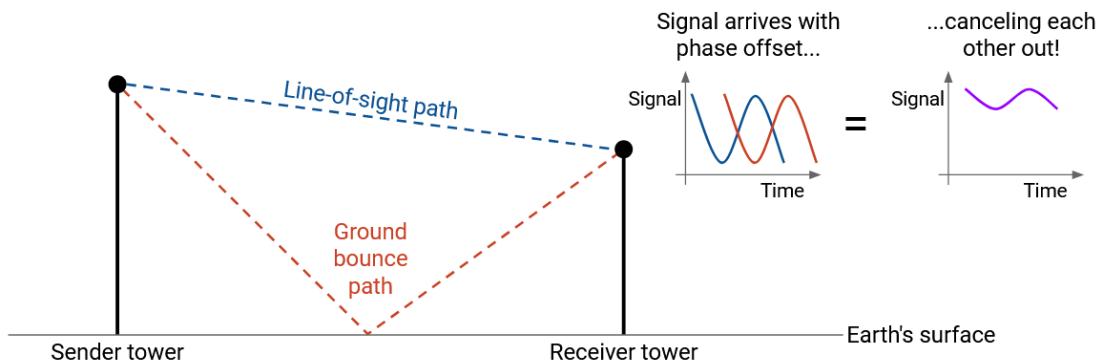
Multipath fading can cause very fine-grained changes in the signal strength. Changing the distance just a little bit might cause the signal strength to get stronger or weaker.

Ultimately, if we want to consider how all three characteristics together affect signal strength over various distances, we have to look at the sum of the three graphs. If the sender and receiver stayed stationary, the signal strength would be a specific point on this graph. However, if the devices are moving, then the signal strength travels along this curve. Also, if the environment changes and obstacles enter and leave, then the graph itself would change as well.

Approximating Path Loss

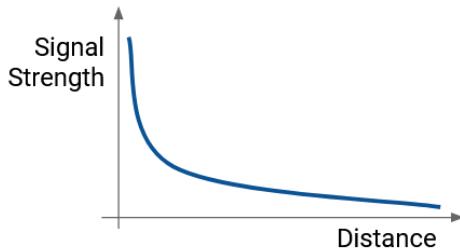
It can be difficult to approximate path loss (from free-space loss, shadowing, and multipath fading). This is especially difficult in the presence of obstacles that result in a signal taking multiple paths, causing out-of-phase signals to interfere with each other at the receiver.

One relatively simple model for approximating path loss is the **two-ray model**. In this model, we assume that the signal travels along only two paths: one line-of-sight path directly from the sender to the receiver, and one ground-bounce path that reflects off the ground to the receiver. Remember, this is still one signal radiating from the transmitter, but some waves directly reach the receiver, while others bounce off the ground to the receiver.



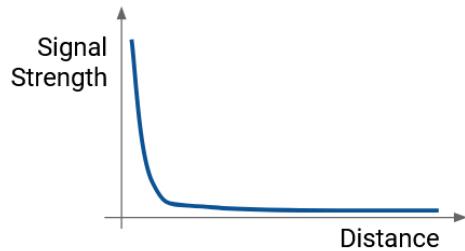
If the sender and receiver are far enough apart, the waves from the two paths will be 180 degrees out of phase. As a result, the waves from the two paths will destructively interfere and cancel out, significantly weakening the signal at the receiver. When this happens, the signal strength is no longer proportional to $1/d^2$, but instead is proportional to $1/d^4$. In other words, signal strength now falls off much faster as the distance increases.

Remember, our free-space model assumed no obstacles (not even Earth's surface), which is why we derived that signal strength is proportional to $1/d^2$. In the two-ray model, accounting for Earth's surface causes signal strength to now be proportional to $1/d^4$.



Free-space model:

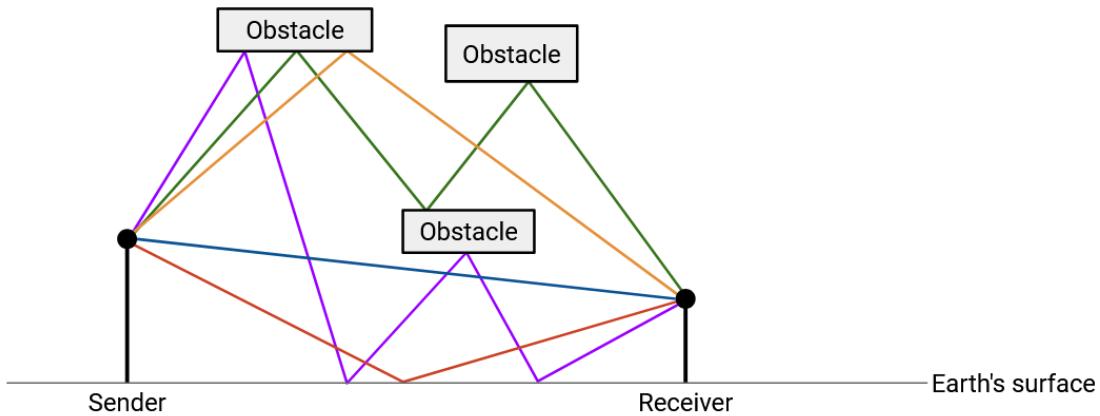
- Signal strength $\propto 1/d^2$.
- Idealized, no obstacles.



Two-ray model:

- Signal strength $\propto 1/d^4$.
- Signal bounces off ground.
- Causes destructive interference.

What if there are additional obstacles besides Earth's surface? The two-ray model doesn't account for those. In more complicated environments, we can create general ray tracing models, which account for signals being reflected, scattered, and diffracted. These models require specific information about the environment (e.g. where the obstacles are), and can be built using computer simulations. In these models, reflected versions of the signal usually dominate the signal, compared to the unobstructed line-of-sight version of the signal.



From these models, we can derive a simplified path loss model to relate distance and signal strength:

$$P_r = P_t K d^\gamma$$

In this equation, as before, P_r and P_t represent the receiver signal power and the transmitter signal power, and d represents the distance.

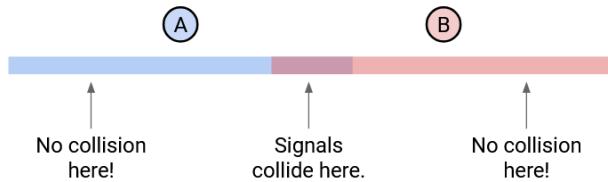
K and γ are empirically-determined constants, based on the environment and the model. For example, if there are lots of inconveniently-placed obstacles, K might be really small, causing the receiver signal strength to be weak.

In practice, γ is between 2 and 8. In the best case, signal strength is proportional to $1/d^2$, similar to the free space model. In the worst case, signal strength is proportional to $1/d^8$, and the signal gets weaker much more rapidly as you move further away.

Difference: Detecting Collisions

Wired collisions are often easy to detect. On a point-to-point link, they might not happen at all. We can usually detect collisions just by sensing the wire. There can be issues with propagation delay, but ultimately, there's just one signal on the wire that we have to sense.

By contrast, wireless collisions are much harder to detect, because there is now a spatial aspect to collisions. Waves might collide in one place, but not another.



Designing collision detection and collision avoidance is much harder in a wireless system, but it's still necessary so that multiple devices can send over the shared medium. Recall that there are many different approaches to multiple access, including fixed allocations of frequencies, and coordinating who's sending at what times. Which approach works best depends on your environment. For example, if you're in the middle of nowhere, it might be okay to just let collisions happen and deal with them when they do. In this section, though, we'll focus on the CSMA (Carrier Sense Multiple Access) approach, where you listen for signals and don't transmit if someone else is talking.

In this section, for simplicity, we'll ignore obstacles, which means that signals radiate outwards in all directions. We'll assume that signals radiate up until a certain distance at full-strength, and that signals are undetectable past that distance. Also, in our running examples, we'll simplify and assume all devices are arranged in a line, so we just need to consider signals propagating left and right. Remember, though, in real life, signals radiate outwards in three dimensions.

Problems with CSMA

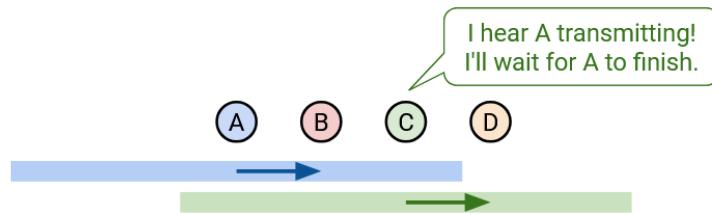
To check if someone else is talking, the radio tries to detect energy exceeding a certain threshold. If it does detect, then we conclude that somebody else is transmitting.

This strategy works fine if two well-separated pairs of devices are communicating.



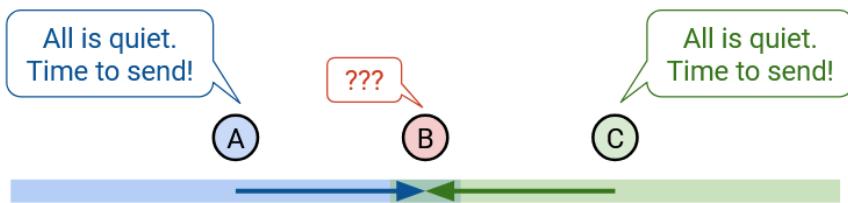
In this example, A and B want to talk, and C and D want to talk. A senses nothing, and starts transmitting to B. Notice that A's signal propagates in all directions, not just toward B. Later, C senses nothing (since it's out of A's range), so it can start transmitting to D.

This strategy also works fine if the two pairs of devices are within range of each other.



Again, A and B want to talk, and C and D want to talk. A senses nothing, and starts transmitting to B. Later, C senses a signal, since A is talking and C is within range of that signal. Therefore, C will wait until A is done, and only start transmitting to D afterward.

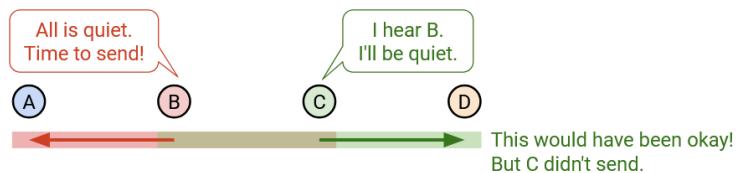
Sometimes, this strategy leads to problems.



Suppose that A and C both want to talk to B. A senses nothing, and starts transmitting to B. Later, C senses nothing, because it's out-of-range of A, so C also starts transmitting to B. There's a collision at B!

This is called the **hidden terminal problem**. In this case, the two transmitters (A and C) were out of range of each other, so they could not sense that a transmission was happening.

Here's another case where CSMA is problematic:



In this case, suppose that B wants to talk to A, and C wants to talk to D. First, B senses nothing and starts transmitting to A. Remember, B's signal propagates in all directions, including to C. Now, C wants to talk to D, but senses B's signal and stays quiet.

If you look carefully, B and C could have actually transmitted at the same time. It's true that collisions would happen in the space between B and C, but the receivers (A and D) won't sense any collisions.

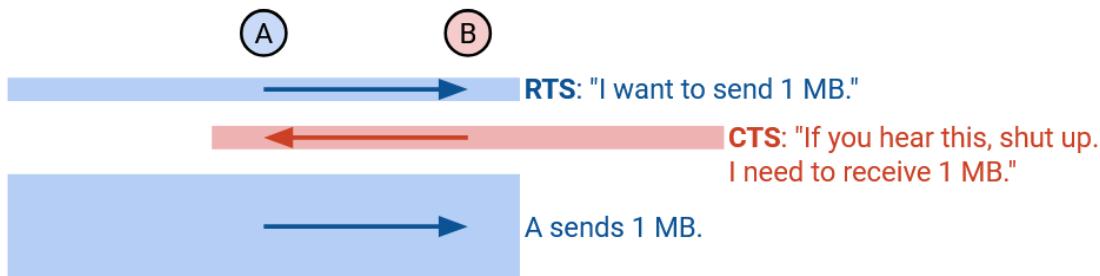
This is called the **exposed terminal problem**. In this case, the two transmissions could have occurred at the same time, but instead, one transmission is prevented from happening because C is falsely detecting a collision.

MACA for Collision Avoidance

Instead of using CSMA, **MACA (Multiple Access with Collision Avoidance)** is an approach to multiple access that will help us solve the hidden terminal problem.

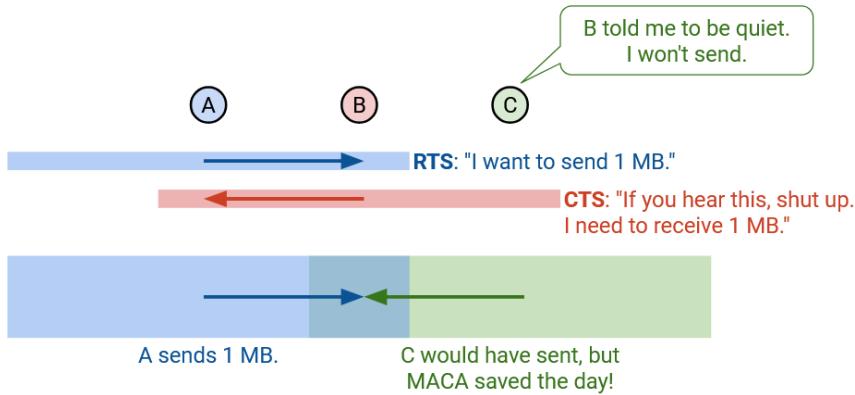
The key problem with CSMA was, the sender was detecting collisions at the sender, but the real problem is collisions at the receiver. To solve this, we will have the receiver announce whether it detects any collisions.

Suppose A wants to send data to B. A successful data transfer involves a sequence of 3 steps:



1. A transmits a **Request To Send (RTS)** packet with the length of the data. This is A saying: "I'd like to send k bits to B."
2. B transmits a **Clear To Send (CTS)** packet with the length of the data. This tells A that it's safe to send, and confirms that there are no collisions at the receiver. The CTS also warns everybody in B's range: "I'm B, and I'm about to receive k bits, so please don't talk during this time."
3. A transmits the data, and B receives the data. The CTS warning ensures that everybody else in range of the receiver stays quiet during this time.

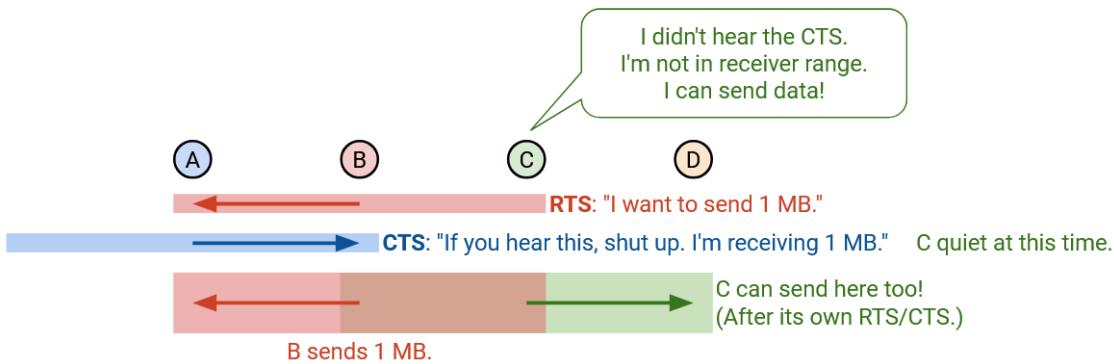
This protocol solves the hidden terminal problem. Remember, in the hidden terminal problem, A and C both sense quiet and start transmitting, causing a collision at B. With this protocol, if A sends an RTS, B will transmit a CTS, warning everybody in B's range (including C) to be quiet.



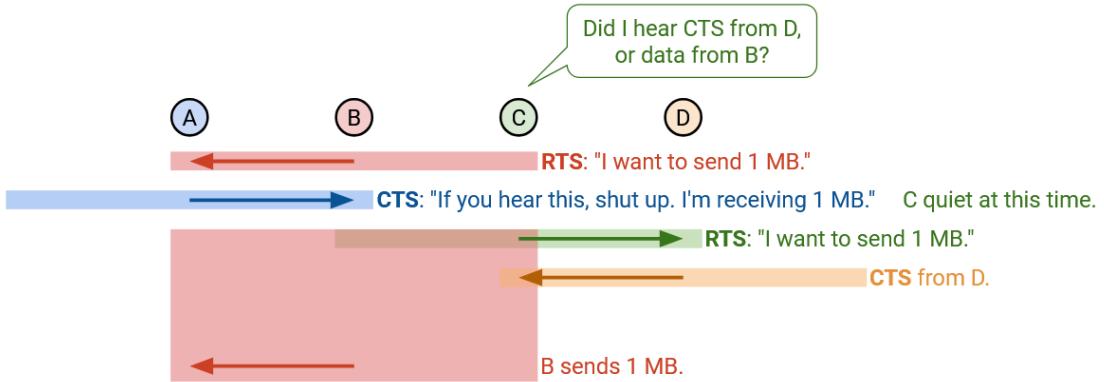
If you hear an RTS packet, this means you're in range of the sender. The sender is about to listen for a CTS. Therefore, you need to be quiet and wait for one time slot, which is long enough so that you don't clobber out the CTS at the sender with data of your own. In other words, you need to be quiet and let the sender receive a CTS.

After the RTS, if you then hear a CTS, that means you're also in the range of the receiver, so you must also be quiet during the data transfer. If you don't hear the CTS, that means that you're out of range of the receiver, and you can transmit data yourself.

Under certain assumptions, this protocol solves the exposed terminal problem. Remember, in the exposed terminal problem, B is sending to A, and C is sending to D. With CSMA, C senses B's signal and stays quiet, though it could have safely transmitted. With this protocol, if B sends an RTS, C will defer for one time slot (to avoid clobbering the CTS at B). Then, because C didn't hear the CTS, this means that C is out of range of the receiver (A), so C can safely start transmitting to D.



The assumption we make for this to work is, C must be able to hear the CTS from D. Remember, even though C is the sender, it must receive the CTS before it can start sending. However, C is actually hearing the data from B as well, so it might not be able to hear the CTS to start sending. The key problem here is: In CSMA, the sender only ever sends. But in MACA, the sender actually has to receive a CTS before it can start sending, and that CTS might be clobbered in the exposed terminal case.



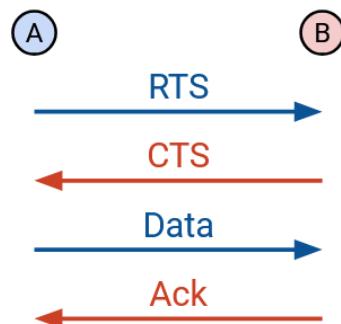
If we send an RTS, but we don't hear a corresponding CTS, this means that we are not clear to send. There's a collision at the receiver, maybe because the receiver is currently receiving data, or because the receiver gets two requests at the same time. If this happens, we apply binary exponential backoff (similar to CSMA/CD), and wait up to twice as long before sending another RTS.

In MACA, each device maintains a CW (Contention Window) value, which tells us how long after a collision we should wait before re-requesting. If we detect a collision (no CTS), we pick a random number between 0 and CW, and wait that long before re-requesting. The minimum value is 2 slots, and the maximum value is 64 slots, where one slot is the time it takes to transmit an RTS. On a successful RTS/CTS, we reset the contention window back to the minimum value of 2. On a failed RTS (no CTS), we double the contention window, clamped to avoid exceeding the maximum value of 64.

MACAW Feature: ACK (For Reliability)

MACAW (Multiple Access Collision Avoidance for Wireless) offers some improvements over the MACA protocol.

The first improvement is adding acknowledgements for reliability. As before, the sender transmits an RTS, and the receiver transmits a CTS, and the sender transmits data. Now, we have an extra step at the end, where the receiver transmits an ack.



If the data is lost, then there won't be an ack, and the sender will have to retry, starting over with a new RTS. If the data is correctly sent but the ack is lost, then the sender will retry with a new RTS, but the receiver can immediately reply with the ack instead of the CTS.

Why did we add acks? Remember, the end-to-end principle said that reliability must be implemented at the end hosts for correctness. However, in this case, we're implementing reliability in the network, along a single link, solely in order to improve performance. If we didn't implement reliability on the link, TCP would still guarantee correctness, but a lost packet would cause TCP to slow down significantly (recall, congestion window halves). By contrast, by implementing reliability on the link, we can recover from packet losses more efficiently.

MACAW Feature: Better Backoff (For Fairness)

The MACA protocol is unfair when two colliding hosts want to send data. In particular, winners tend to keep winning, while losers keep to keep losing.

Here's an example of unfairness. Suppose A and B both have their windows set to 2, and they simultaneously attempt to reserve the channel. Let's assume A wins, and B loses. Then A's window stays at 2, while B's window doubles to 4. This means that A probably gets to reserve the channel again sooner, and will probably win again. This also means that by the time B tries again, A has already captured the channel again, and B's window doubles again to 8. This pattern continues, and A keeps re-capturing the channel quickly, while B keeps failing and waiting increasingly longer before trying (and failing) again.



To solve this problem, instead of each device having its own CW, we'll have everybody share the same CW. The packet header now contains a field for the CW value, and if you receive a packet, you set the CW to the value in the packet. Since everybody now has the same CW, the retry mechanism doesn't favor any one device. Everybody picks a random value between 0 and CW and waits that long. (Note: We're slightly simplifying here, but this is true if all devices are in range of each other.)

MACAW also changes the CW update rules to be more gentle. As before, the minimum value is 2 and the maximum value is 64. On a failed RTS (no CTS), we multiply CW by 1.5 (instead of doubling), and again clamp to avoid exceeding 64. On a successful RTS/CTS/DATA/ACK transmission, we decrease CW by 1 (instead of resetting all the way to 2). Note that on a successful RTS/CTS, but a failed ACK, the CW does not change. This approach is sometimes called Multiplicative Increase, Linear Decrease (MILD).

MACAW Feature: DS (For Exposed Terminals)

Recall our exposed terminal example from earlier, where B wants to communicate with A. B sends an RTS, A sends a CTS, and B starts transmitting data. At this point, C did not hear the CTS, which means C is out-of-range of the receiver and can safely transmit data. However, in order to transmit, C must hear the CTS. This might not happen, since C is also hearing B's data, and there might be a collision between B's data and D's CTS.

MACAW concludes that in the exposed terminal case, B-to-A and C-to-D actually cannot send data simultaneously. Yes, we're admitting defeat, and it turns out neither MACAW nor CSMA solves the exposed terminal problem.

This means that if we're in the range of the other sender, we actually can't send data (even if we aren't in the range of the other receiver). To repeat, this is because we'll be hearing the data from the other sender, which means we can't hear the CTS we need to start sending.

To solve this problem, we add an extra Data Sending (DS) packet before the data. This is the sender warning everybody: I'm about to send k bits of data, so you need to be quiet during this time.



The protocol now has 5 steps:

1. Sender transmits RTS, requesting to transmit k bits of data.
2. Receiver transmits CTS, telling everyone in range: Be quiet, I'm receiving k bits of data.
3. Sender transmits DS, telling everyone in range: Be quiet, I'm sending k bits of data. (Others can't send data, because my data will clobber the CTS you need to receive for your transmission.)
4. Sender transmits the data.
5. Receiver transmits the ack.

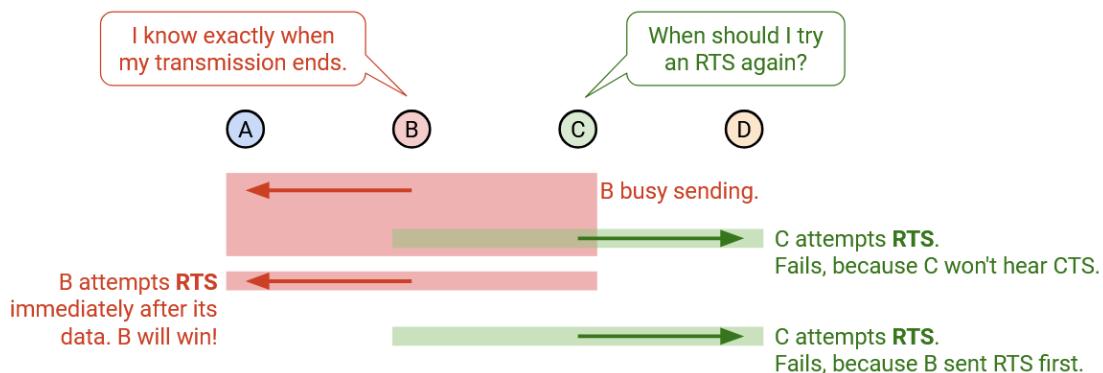
Note that the RTS and DS are not redundant. The RTS is a request that might not be granted (e.g. maybe no CTS). The DS confirms that the request is actually granted, and enforces that everyone in the sender's range must be quiet.

MACAW Feature: DS (For Synchronization)

The DS has a second important purpose. Let's revisit the exposed terminal again, remembering that MACAW accepts defeat and forces the two transmissions to happen separately.

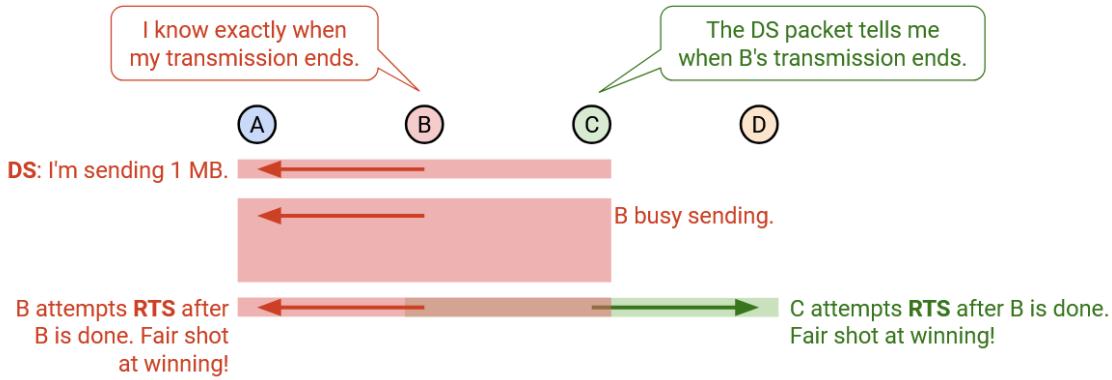
Assume there's no DS packet. Then, as before, B sends an RTS, A sends a CTS, and B starts transmitting data. C hears the RTS, and defers for one time slot (to avoid interrupting B). However, C does not hear the CTS. At this point, C is doomed to send a futile RTS, and will never hear the CTS (because it's drowned out by the data from B). C will keep retrying and sending out futile RTS requests, but it has no idea when B will stop sending data.

By contrast, B knows exactly when it will stop sending data. This gives B a huge advantage in the next round of contention. When B is done sending data, it can immediately send out another request, and it will probably win and get to keep sending data. On the other hand, C has no idea when B will stop sending data, so it has to randomly guess when to send out another request. Most likely, C will guess a time and re-request while B is still sending data, so C will lose and the request won't be granted (collision).



This lack of synchronization leads to unfairness. If I win, I'll probably win again, because I know exactly when the next round of contention will happen (it's when I'm done sending). If you lose, you'll probably lose again, because you don't know when the next round of contention will happen (you don't know when I'm done). The contention time is usually a tiny sliver of time, since most of the time is spent sending data. I know exactly when that time is, and you don't, so I'll keep winning.

The DS packet solves this problem, because it allows the sender to tell everybody when the next round of contention occurs. Now, B is using the DS packet to tell everybody: I'm starting to send k bits. Not only does C now know to not send futile RTS requests, but it also knows when B will be done sending. This gives C a much fairer shot at winning the next round of contention.

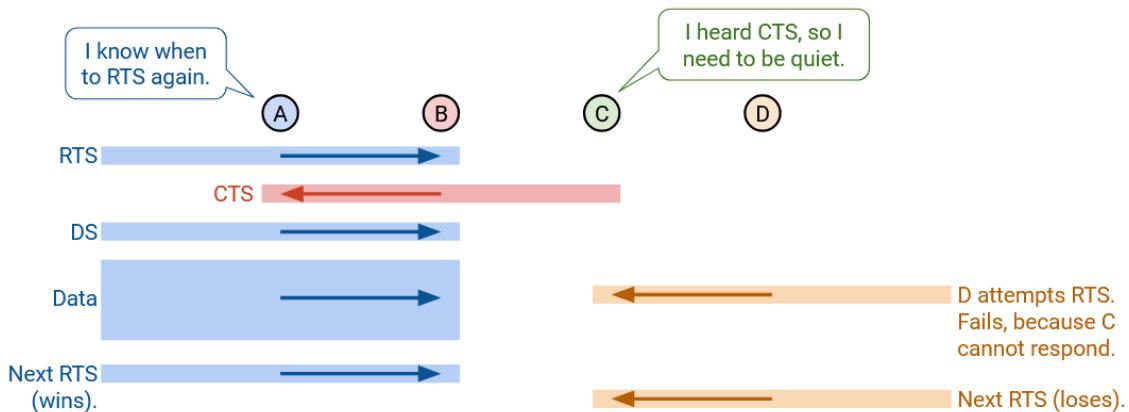


MACAW Feature: RRTS (For Synchronization)

There's another case where synchronization is critical to ensure fairness. Suppose A wants to send to B, and D wants to send to C.

A transmits to B (A sends RTS, B sends CTS, A sends DS and sends data). C hears the DS and must stay quiet while the data is sent. Now, D is clueless and doomed. D will send an RTS, and won't hear a CTS because C is staying quiet. D will keep retrying at random times, and will keep failing, because it has no idea when A will stop sending data.

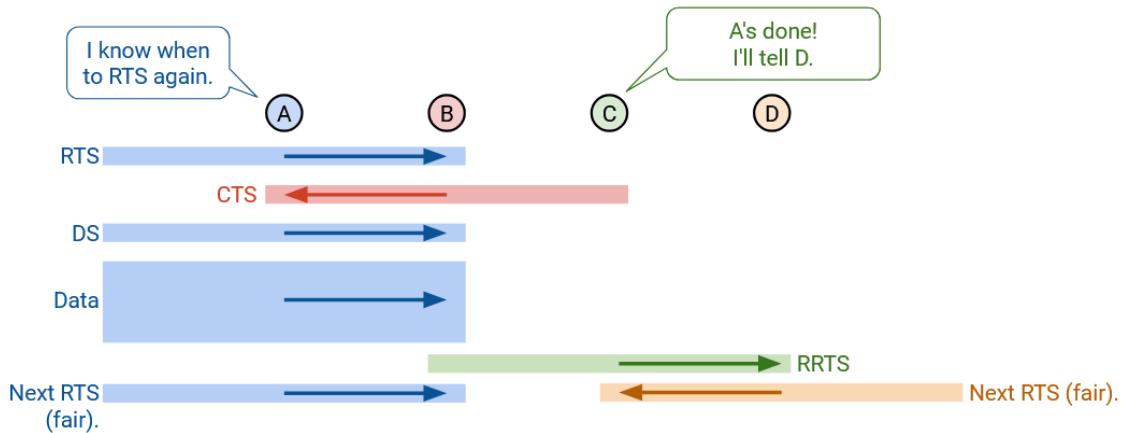
By contrast, A knows exactly when it will stop sending data. Just like before, this gives A a huge advantage in the next round of contention. A can immediately re-request and win. On the other hand, D has no idea when to re-request. The only way D will win is if it gets really lucky and sends the request immediately after A is done sending, but before A re-requests.



Notice that the DS packet doesn't help us here, because the two senders, A and D, are out-of-range of each other. A will send a DS packet and announce when it's sending data, but D won't hear it, so D is still doomed to lose.

To solve this problem, we'll let the receiver do the contending on behalf of the sender. D doesn't know when to re-request, but C does, so let's make C do the requesting instead.

When D sends the RTS, C learns that D wants to talk, but C must stay quiet until the next round of contention. Notice that C knows when the next round of contention occurs, because it will hear the ack from B. When the next round of contention occurs, C sends a new packet called a Request-for-RTS (RRTS). This immediately alerts D that the next round has begun, and allows D to immediately send an RTS. This gives D a much fairer shot at winning the contention round.



If you hear an RRTS, this means that someone in your range is trying to request, so you should be quiet for 2 time slots while they perform the RTS/CTS exchange.

In the example, if C sends out an RRTS, B hears this and stays quiet for two time slots, which allows D to send an RTS, and C to send a CTS. The CTS tells B to be quiet, and allows the D-to-C transmission to happen.

More generally, you should send an RRTS if you hear an RTS, but you're not allowed to respond, because someone else has told you to be quiet.

DS and RRTS help with synchronization and ensure more fair contention rounds, but they don't solve all our problems. Consider A sending to B, and C sending to D. Suppose C starts sending to D. At this point, if A sends an RTS, B can't hear it because the RTS is getting drowned out by C's transmissions. The only way A's RTS will reach B is during the short gap in between C's transmissions. Here, A is doomed to lose, because it has no idea when C will stop sending, while C knows exactly when it's sending. Note that RRTS doesn't save us here, because the RRTS is only sent if you hear an RTS, but B never even hears the RTS. B never learns that A wants to communicate, so B will never send an RRTS request on behalf of A. The original MACAW paper leaves this problem unsolved.

Cellular

Why Study Cellular?

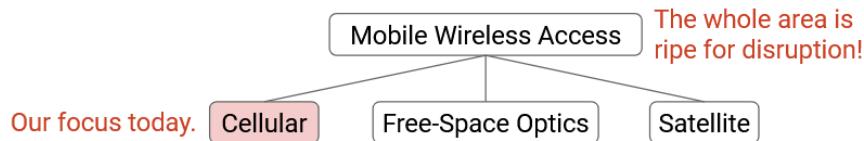
Wireless mobile connectivity is the modern standard. Your phone is able to connect to the Internet while you're in a moving car.

Traditional Internet networks can't support this. You might be able to move from your bedroom to your kitchen and still have Internet access. In that case, you're within range of your wireless home router, which is then connected via wires to the rest of the Internet. However, the traditional Internet doesn't offer seamless connections across wide distances (e.g. moving in a car).

There are many ways to implement wireless mobile connectivity, but cellular is the dominant access technology today. Over half of web traffic today originates from a cellular device!

Cellular is just one of many technologies that can offer mobile wireless connectivity. Other technologies like satellite or free-space optics also exist, though cellular networks are still the dominant approach today.

In the future, high-performance applications that require wireless mobile technology, like self-driving cars or virtual reality, could lead to more innovation. Current cellular networks might get prohibitively expensive as we try to scale them up to support future applications. Also, cellular network operators like AT&T and Verizon don't have a reputation for rapid innovation. The general consensus is that this is an area ripe for disruption in the near future, and is an active area of research.



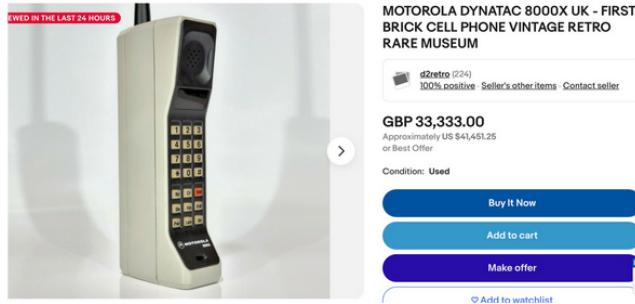
Brief History of Cellular Networks

Cellular technology has its roots in the old telephone system. Cellular networks were first developed to allow users to make phone calls wirelessly, instead of on a wired landline. The first mobile phone was sold in 1983 for \$4,000 (way more today, after inflation).



Martin Cooper made the first mobile call on this Motorola phone.

Sold for \$4,000 in 1983
(over \$12,000 today).



Apparently worth over
\$40,000 today as an antique.

Because cellular technology was derived from the telephone network (not the Internet), many of the design choices differ from the traditional Internet. For many years, cellular technology (e.g. pre-smartphone cell phones for voice calls) and the Internet developed in parallel, each with a different set of architectural choices.

For example, the cellular network uses resource reservations, while the modern Internet uses packet switching. Cellular networks often think in terms of individual users, while the Internet mostly thinks in terms of individual flows or packets. The business model of cellular networks (e.g. charge user by the minute) is different from the Internet, which generally doesn't keep track of usage as much.

In recent years, cellular networks have emerged to be more compatible with the traditional Internet. Today, you can think of a cellular network as a specialized Layer 2 local network that can interact with the rest of the traditional TCP/IP Internet.

Cellular Standards

In the traditional Internet, we saw that standards bodies help us standardize protocols like TCP and IP. The cellular network also has many standards bodies that cooperate to generate a standard.

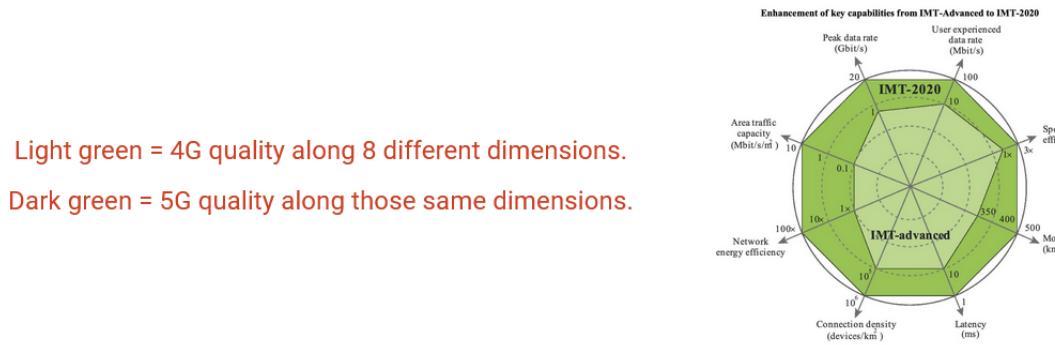
In some ways, the cellular network standards bodies have more real-life political complexity than the Internet standards bodies. In order to achieve interoperability, all manufacturers of cell phones, and all network operators (e.g. Verizon building cell towers), need to agree on protocols, all the way down to the physical layer.

The key standards body in the cellular world is the 3GPP (3rd Generation Partnership Project). The large equipment vendors and telecommunications companies all participate in this organization. The 3GPP proposes standards, which are then forwarded to the ITU (International Telecommunications Union). The ITU is part of the United Nations, and every country gets a vote, so there's some politics involved in standards as well. (Fun fact: Every country gets one vote, so the US can get out-voted by the European Union.)

Typically, a new technology generation is introduced every 10 years. Now you know what the numbers in 2G, 3G, 4G, and 5G represent (generations of cellular technology). The 5G network was defined around 2020, and operators are still working on deploying the technology. Planning for the 6G standard will start

in the next few years (late 2020s).

Each generation tries to improve on the previous generation along multiple dimensions, including peak theoretical data rate, average data rate experienced by users, mobility (connection while user is traveling at a high speed), connection density (number of devices within a specific area), and so on. Each generation usually operates around 10 times better than the previous generation, along all these dimensions.



In addition to performance improvements, the architectural design has also evolved across generations, to move away from the telephone network design and towards the Internet design. 1G phones were purely analog, designed for voice calls. 2G/3G was still mostly circuit-switched, with a focus on voice traffic (a bit of texting, barely any Internet traffic). From 4G onwards, we've moved to a packet-switched architecture, and voice is now just one of many applications running over the network.

Cellular specifications are thousands of pages and include hundreds of documents, and pretty much no one actually reads them in full. One inconvenient feature of these standards is that everything gets renamed when we move from one generation to the next. For example, cellular towers have been called a "base station", "nodeB", "evolved NodeB (eNodeB)", and a "next-gen Node B (gNB)," all meaning the same thing. In this class, we'll invent our own terminology to make the names more intuitive. If you look through a textbook or a specification, you might see different names, but the ideas we'll see should generally be conceptually consistent with textbooks and specs.

Key Challenge: Mobility

The key challenge that makes cellular networks hard is mobility. Remember, think of mobility as your phone playing a video as you're moving down the freeway (don't watch videos while driving though). There are four fundamental challenges that we'll study:

1. Discovery: As I'm moving, how do I know which cell tower to connect to?
2. Authentication: The AT&T tower may only want to offer connectivity to its own customers, but not other customers. How does the cell tower achieve this?
3. Seamless communication: If I move out of range of one tower, and into the range of a different cell tower, my connection should be seamless, with no disruption.
4. Accountability: If the customer only paid for 6GB of data, the network should stop offering the customer connectivity (or offer worse connectivity) after the customer has exceeded their limit. This requirement

comes from the old cellular network (pay per minute of a voice call), and still exists because resources in cellular networks are scarce.

Infrastructure Components: Radio Towers

What are the components of a cellular network? First, there's the radio tower.

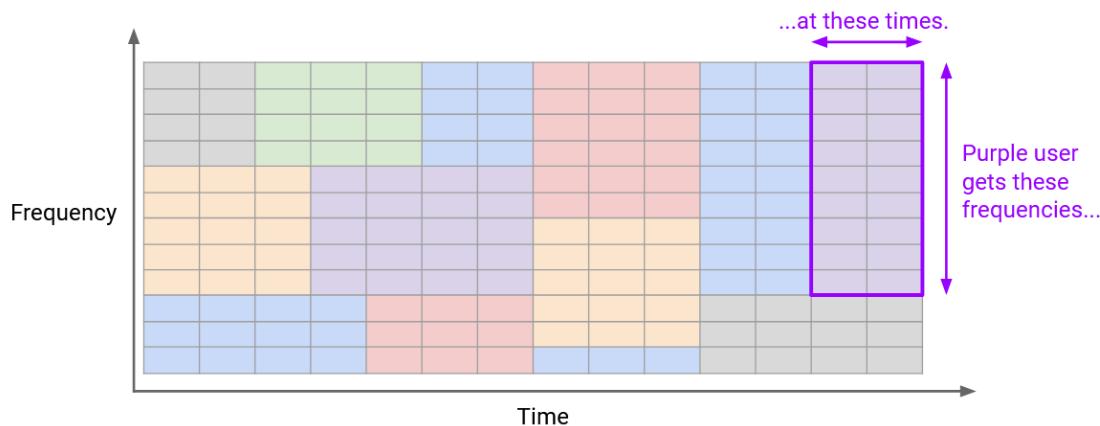
The radio tower has an antenna. Inside the tower is a radio transceiver, which converts digital bits to analog signals sent over the air interface.

Also inside the tower is a radio controller, which decides how to allocate radio resources.



You can think of the controller like a CPU running a scheduler. The controller allocates different segments of frequency and time to different customers, depending on demand and business model (e.g. how much the customer is paying). This is actually a pretty difficult scheduling problem, though we won't discuss further here.

Here's a simplified model of the radio controller allocating resources. Each colored rectangle shows us that a user (denoted by color) can use that specific frequency, at that specific time.



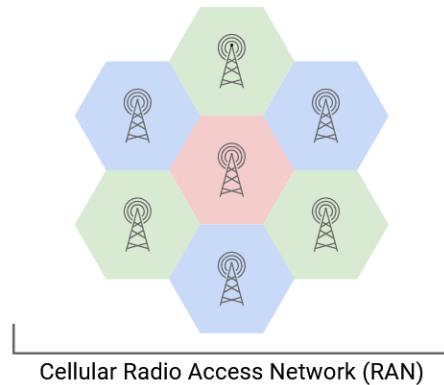
Each vertical cross-section represents one time slot, and shows you how the frequencies have been allocated to users in that cross-section. For example, in the first time slot, the blue user gets 3 frequency slots, the orange user gets 5 frequency slots, and the gray user gets 4 frequency slots.

Each horizontal cross-section represents one frequency, and shows you how that specific frequency is allocated to users over time. For example, the top row shows a frequency being allocated to gray, and later green, and later blue, and later red, and so on.

Notice that this model is sharing resources using reservations, not best-effort. A user can only send in a frequency and time that's been allocated to them by the controller.

Radio controllers were traditionally installed in the tower or near the tower, though nowadays, there's been work to move controllers into the cloud for easier maintenance and management.

Each operator runs many cellular towers, spaced out over the entire country, so that users can connect to a tower no matter where they are. The result is a Radio Access Network (RAN).

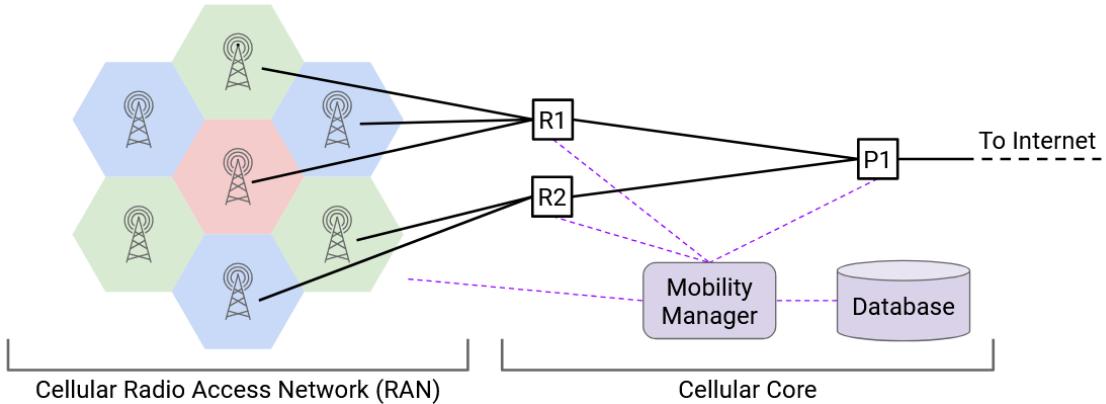


Typically, each tower gets its own set of frequencies that it can use, and frequencies are assigned such that neighboring towers get different frequency ranges. This ensures that neighboring towers don't use the same frequencies and interfere with each other. In this picture, each color corresponds to one set of frequencies. It's possible that two towers both use the blue set of frequencies, but they aren't neighboring so they won't interfere. Any neighboring towers are using non-overlapping frequencies. Note that frequencies are often allocated according to demand, so that a cell tower in downtown San Francisco gets more frequencies than a cell tower in the middle of nowhere.

Infrastructure Components: Cellular Core

A mobile user can now send data to a cell tower. The cell tower now needs to send that data to the rest of the Internet.

Each cell tower has a wired connection to the cellular core. You can think of the cellular core as the backend infrastructure of the cellular network (not user-facing).



The cellular core contains some data-plane components. You can think of these like typical routers and switches that forward packets between the users (via towers) and the rest of the network. We'll focus on two special types of routers in the cellular core.

The radio gateway is the boundary between the RAN (cell towers) and the cellular core. A cell tower forwards its data to one of these radio gateways. On the other end of the core, the packet gateway is the boundary between the cellular network and the rest of the Internet. Data from users eventually reaches the packet gateway and is sent out to the Internet as a standard TCP/IP packet.

The cellular core also contains some control-plane components. We didn't have these in the traditional Internet. User traffic doesn't reach these components. We'll focus on two control-plane components.

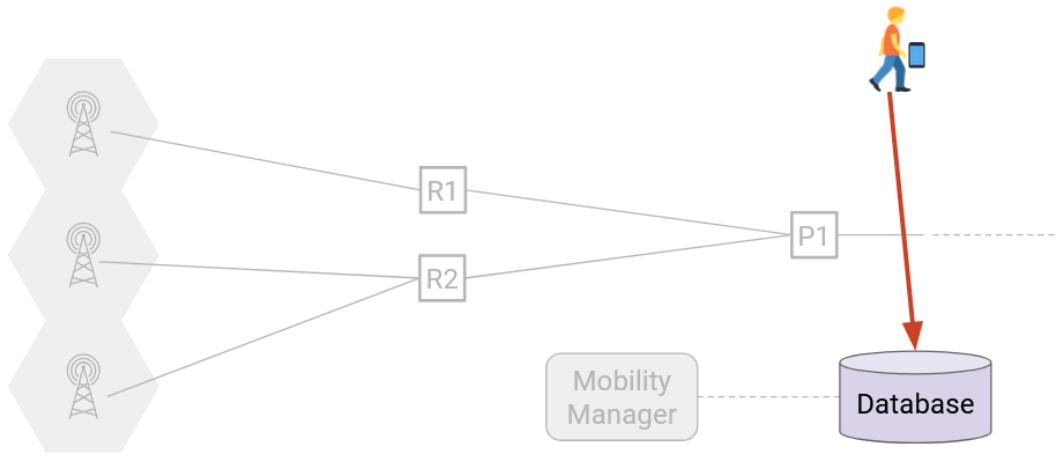
The database stores information about customers, such as: What devices does the customer own? What data plan does the customer have? Where is the customer's device right now (e.g. which tower is it connected to)?

The mobility manager is a controller (think of it like a CPU) that manages network functionality. The manager helps us authenticate a user (e.g. check if they're really a Verizon customer). The manager also helps us update configurations as the user moves around.

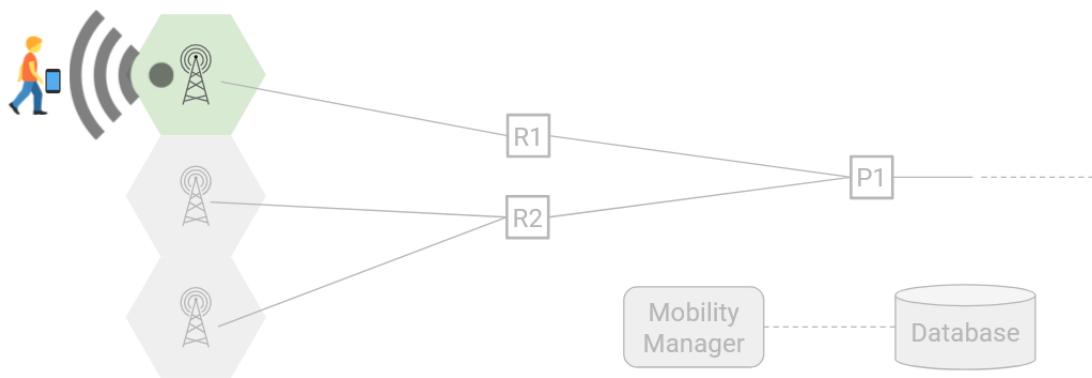
To summarize the infrastructure: User devices send data to cell towers in the RAN. The cell tower forwards the data to the radio gateway (entering the core). The data eventually reaches the packet gateway and gets forwarded to the Internet (exiting the core). Also in the core are control components (mobility manager, database) to store and manage information about customers.

High-Level Steps of Cellular Operation

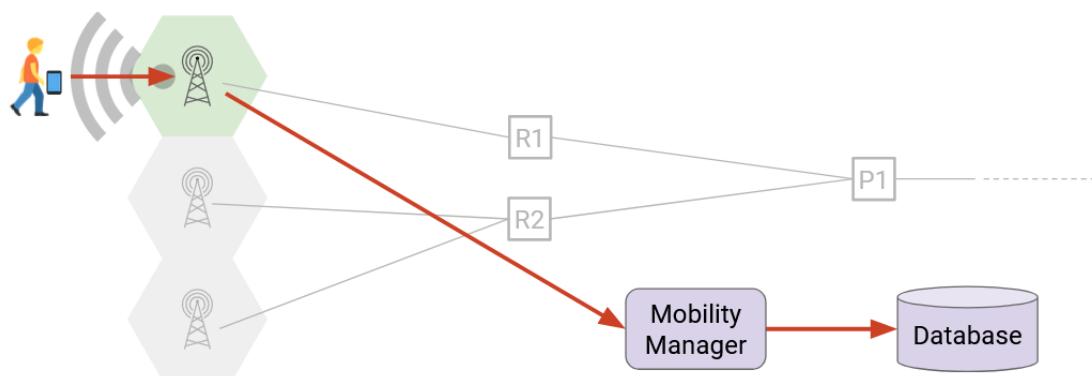
Step 0: Registration. The user registers for the cellular service. For example, you walk into a Verizon store and purchase a data plan and sign a contract. The operator now stores information about you and your service plan in the database.



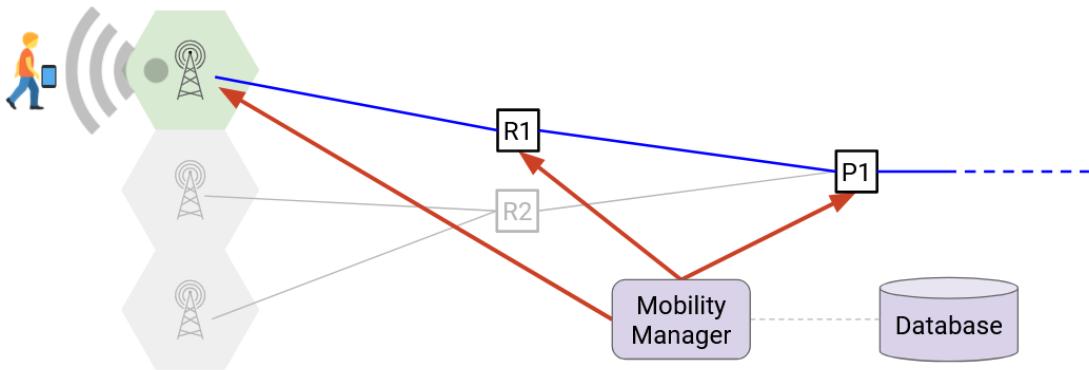
Step 1: Discovery. The user turns on their phone in the middle of nowhere. Their phone must discover which nearby towers are available, and must also pick a tower to use.



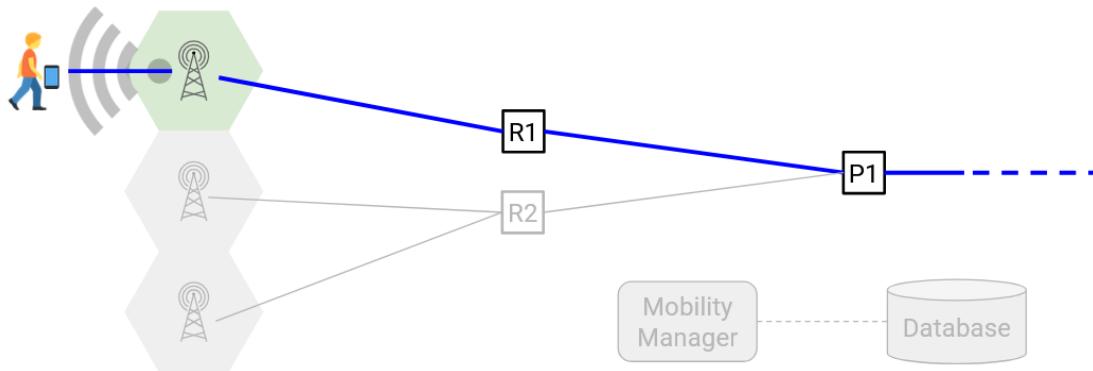
Step 2: Attachment. After picking a tower, the user's device tells the tower that it wants to connect. The tower must ask the mobility manager if the connection is allowed (e.g. check if the user has exceeded their quota).



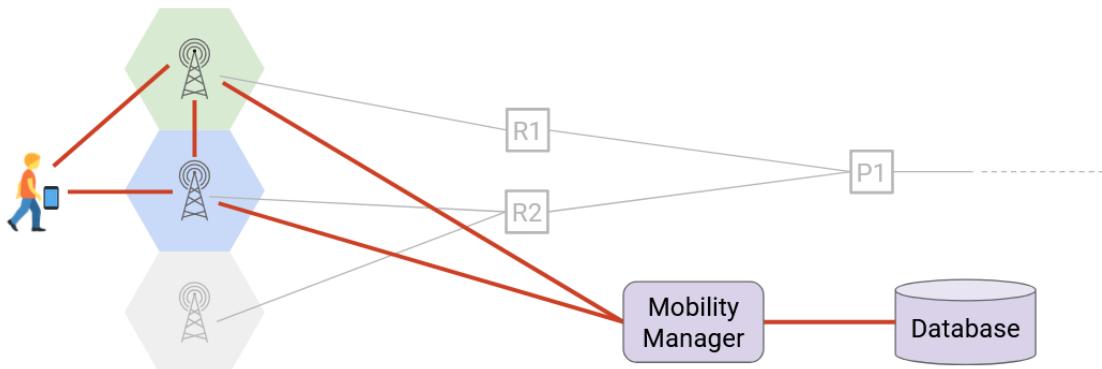
If the authentication checks out, then the mobility manager configures the tower and the routers to establish a path from the user to the Internet (via the tower and the routers).



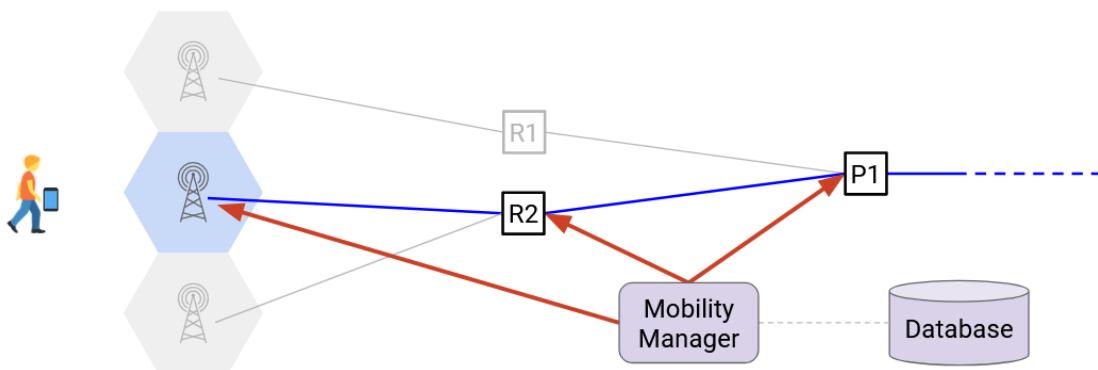
Step 3: Data exchange. The user can now send and receive data along the path configured.



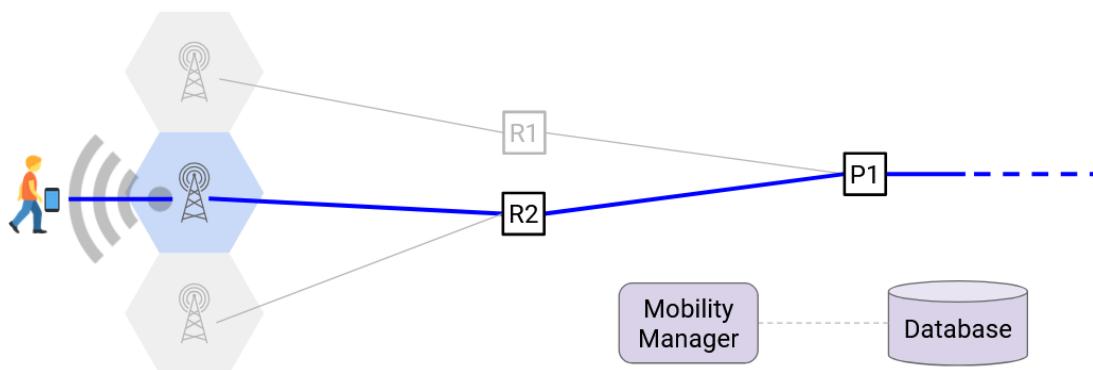
Step 4: Handover. As the user moves around, they might move away from their original tower, and closer to a new tower (in the same operator's RAN). The old tower, new tower, and the user's device all work together to decide if the user should switch towers.



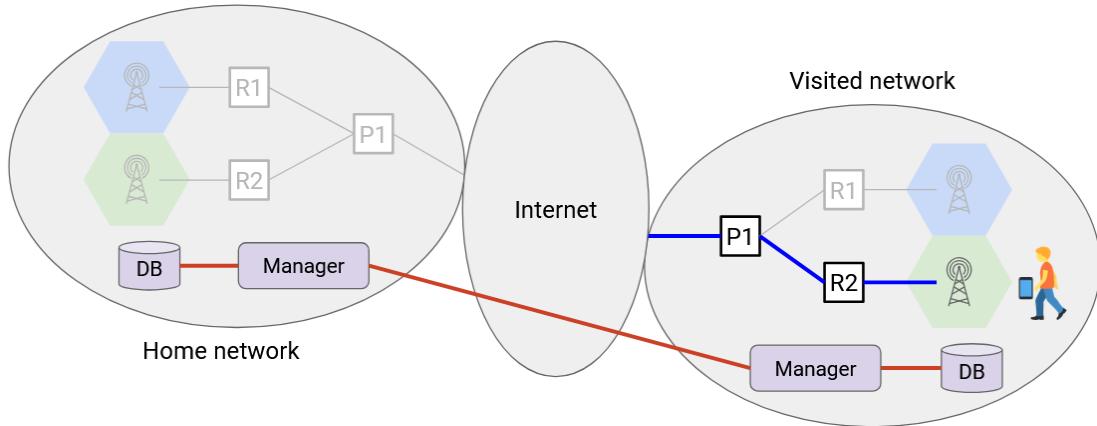
If everyone agrees that the user should switch towers, they tell the mobility manager, and the mobility manager re-configures the tower and the routers to establish a new path from the user to the Internet (now using the new tower, and possibly different routers too). This handoff must be seamless, which means the user could be sending and receiving data through the whole process, and shouldn't be disrupted. Achieving such a seamless handoff requires the network to constantly babysit the user device.



Steps 3 and 4 can repeat as the user moves around, and the best router to use keeps changing.



One final feature we need to implement is roaming. If the user goes to a different country like Germany, their operator (e.g. Verizon, US-based) might not have coverage in Germany. But, Verizon might sign a contract with Deutsche Telecom (an operator in Germany), to allow Verizon's customers to use Deutsche Telecom's infrastructure. This means that Deutsche Telecom might need to support not only its own users, but also users from other networks like Verizon.



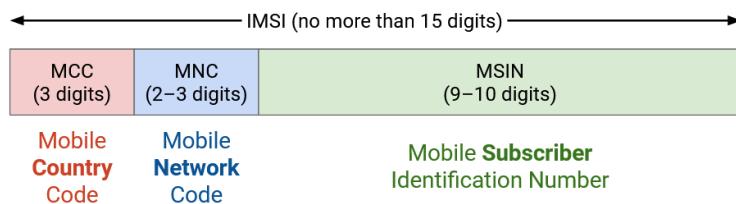
The steps of connecting in a visiting network (while roaming) are generally pretty similar, except the mobility managers in the visited network and the home network must also coordinate with each other (e.g. Deutsche Telecom checks with Verizon to see if the user paid for roaming).

Step 0: Registration

When you register for a data plan, you receive an IMSI (International Mobile Subscriber Identity), which is a unique identifier associated with your subscription. This number is securely stored in hardware in a SIM card.

Note: This is why operators like Verizon give you a SIM card to insert into your phone. If you switch phones, but stay on the same plan, you just have to transfer the SIM card into your new phone, and now your new phone is associated with the same IMSI number. Or, if you switch plans, but use the same phone, you put a new SIM card in your phone, and now that phone is associated with a new IMSI number.

The first 3 digits of the IMSI are the Mobile Country Code, identifying a country. The next 2-3 digits are the Mobile Network Code, representing your service provider (e.g. Verizon, AT&T). The remaining digits are the Mobile Subscriber Identification Number, which identifies a specific user within that service provider. The IMSI overall cannot exceed 15 digits.



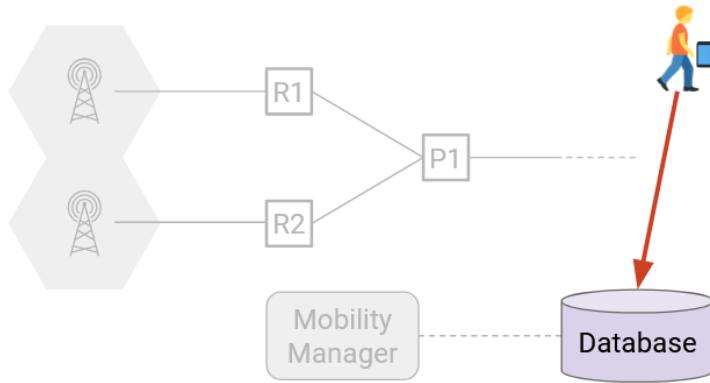
Note that the IMSI is not the same as an IP address. If you pay for a year-long data plan, you keep the same IMSI all year. But, each time you attach and connect to the network, you could get a different IP address.

There are two other identifiers used in cellular networks. They're distinct from the IMSI, and we won't cover them in a lot of detail. The IMEI (International Mobile Equipment Identity) uniquely identifies a physical device. The IMEI encodes the device manufacturer and model ("this is an iPhone 13"), and stays

the same even if you change data plans. Or, if you have two phones covered by the same data plan, you'd have two IMEI numbers, but only a single IMSI.

The other identifier is your phone number. Again, this is distinct from the IMSI or IMEI, and the digits represent different things (e.g. your area code). The phone network will need to associate your phone number with a specific IMSI to determine your phone plan.

After you register and receive an IMSI, the operator (e.g. Verizon) stores your IMSI and information about your plan in the database.



During registration, the user's device (SIM card) and the operator (database) also agree on a shared secret key. This will be useful when we do attachment.

Step 1: Discovery

How does the user device discover which towers are in range, and owned by the user's operator?

Each tower transmits periodic beacons (hello messages), telling everybody in range that the tower exists. The beacon message also includes the network operator (e.g. hello, I'm a Verizon tower), where the operator is identified by the 2-3 digit Mobile Network Code. Remember, the device's IMSI (on the SIM card) also has a Mobile Network Code, so the device can check: My SIM card says I'm in network 220, and this tower's beacon says it's in network 220, so I can use this tower.

The beacon is transmitted on a specific frequency called the control channel, so that the beacon doesn't interfere with data transmissions. Each frequency range has its own associated control channel. Recall that neighboring towers have non-overlapping frequency ranges, which also means they have different control channels (avoids interference).

The user's device might hear many beacons. The user measures the signal strength to different towers, and picks the tower (belonging to its operator) with the best signal.



There's one problem we have to solve. How does the user's device know which control channel to listen to? The device needs to tune in to the control channel in order to pick up the beacons. We have a bootstrapping problem.

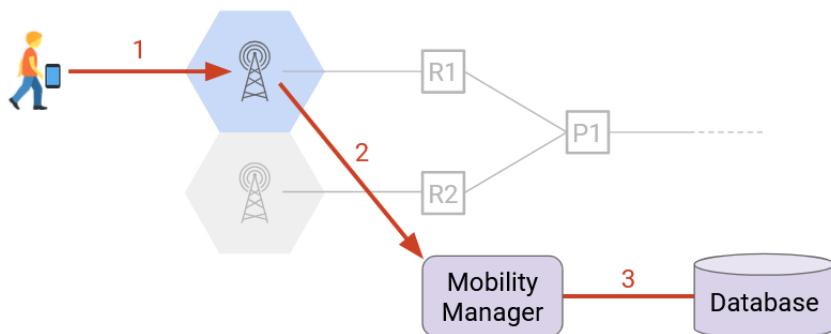
There are a few solutions to this problem. The device could just scan and try a bunch of frequencies (slow, but sometimes the only option). The operator might give the device a pre-configured list of control channels during registration. The device can also cache previously-used channels.

Note that scanning for subsequent towers after discovery is not necessary. During handovers, the old tower will tell users exactly which data frequency to use on the new tower. This is why handovers (order of 0.01–0.1 seconds) are much faster than scanning during discovery (order of 10–100 seconds).

Step 2: Attachment

1. Once a user has discovered a tower, it sends an attach request to that tower. The user includes its IMSI (subscriber ID) in the request.
2. The tower must then send the request to the mobility manager, which actually processes the request.
3. The manager looks up the IMSI in the database to learn the details about the user's service plan. The manager also performs authentication cryptographic details omitted) by using the secret key known by the device and the manager (in its database).

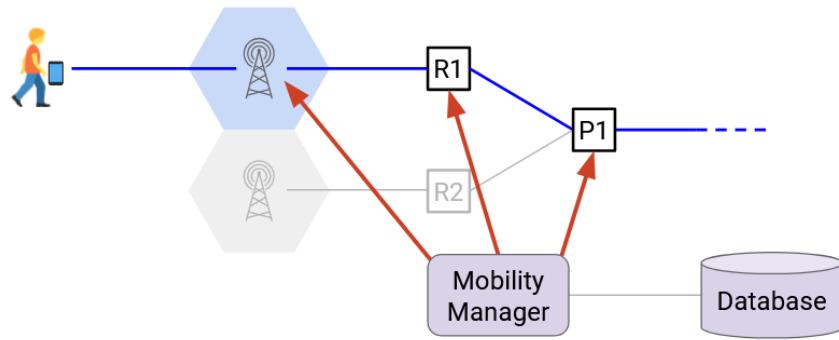
If the authentication succeeds, we know the user is who they say they are. If the database lookup also shows that the user is eligible for service, then the manager approves the attach request.



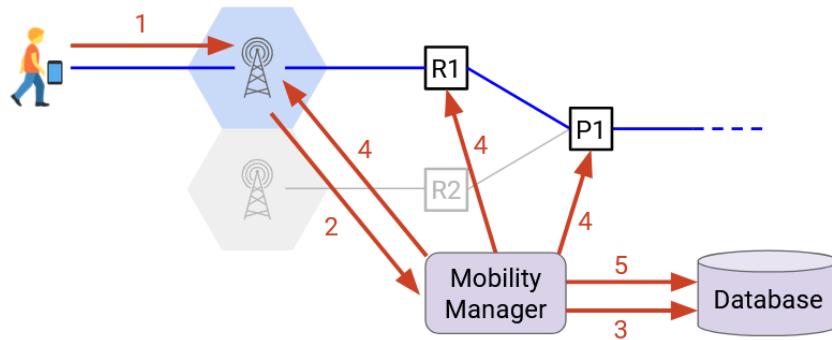
4. After the attach request is approved, the mobility manager now has to configure the data plane to give the

user connectivity. First, the manager assigns an IP address to the device. Then, the manager configures the tower, telling the tower radio controller how many resources to allocate for this user. The manager also configures the tower and the routers to create a path between the device and the Internet. Finally, the manager initializes counters and shapers to keep track of the device's Internet usage.

After setting up the user's connectivity, the manager finishes by recording the user's location information in the database. Specifically, the database maps the user's IMSI to its IP address and the path it's using (which tower, which gateways).

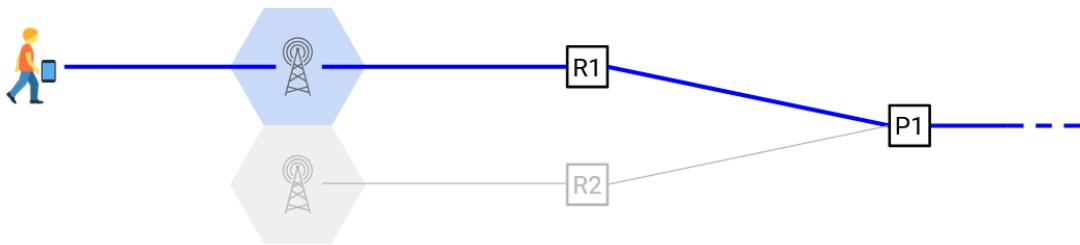


Note that the entire attachment process occurs over control channels. We haven't assigned any frequencies to the user yet, so the user has to use dedicated control channels to communicate.



Step 3: Data Exchange

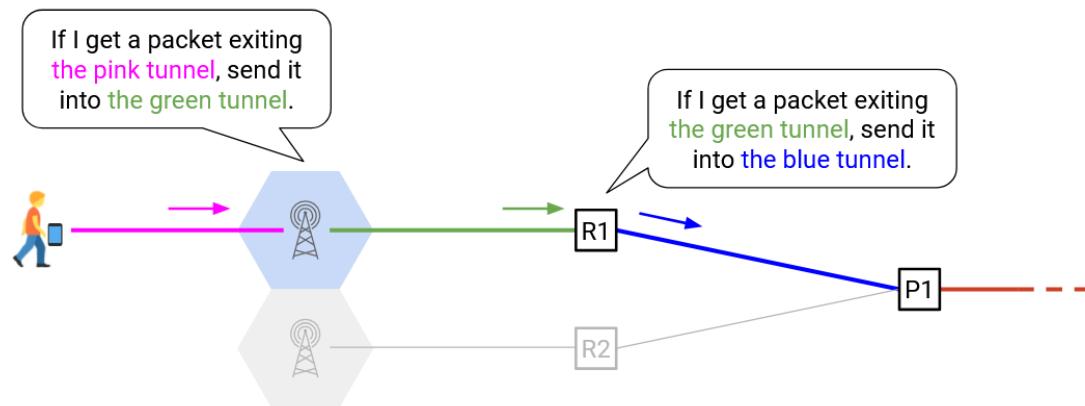
At this point, the network is configured so that the device can use its IP address to send and receive messages.



How does the cellular network (tower, radio gateway, tower gateway) know how to forward packets? Users are constantly moving, so if we ran a traditional routing algorithm like distance-vector, routes would never converge.

Instead, the manager will create a path between the device and the Internet using tunnels. Remember, the packet's path is from the device, to the tower, to the radio gateway, to the packet gateway.

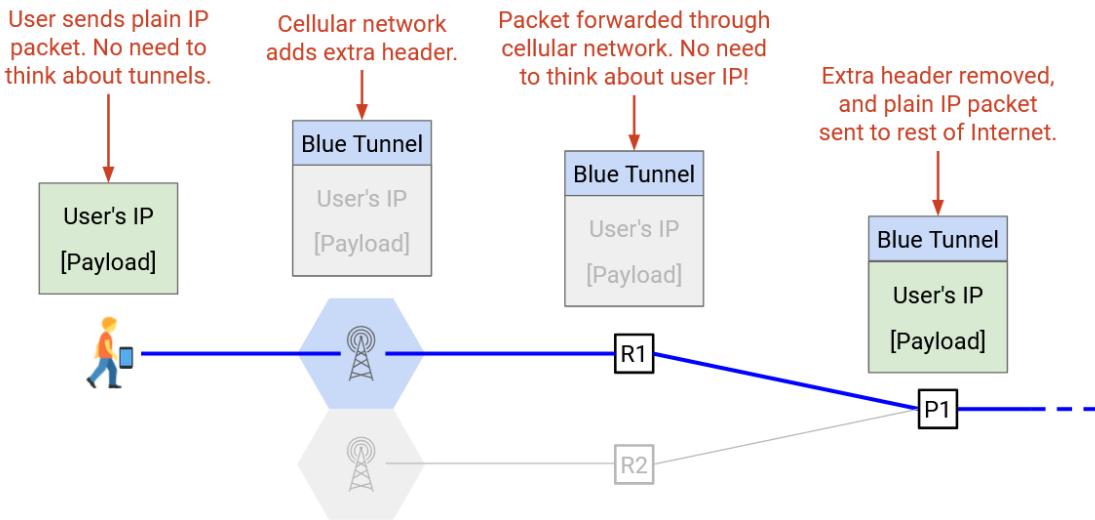
Conceptually, to implement the tunnel, we'll tell the tower: If you get a packet from the user, send it this way (into the blue tunnel). On the other side of the wired link, the packets will exit the blue tunnel and arrive at the radio gateway. We'll then tell the radio gateway: If you get a packet exiting the tunnel, send it this way (into the green tunnel). Packets then travel through the green tunnel and arrive at the packet gateway, who can forward the packet into the Internet.



Incoming packets also travel through the tunnels. We tell the packet gateway: If you get a packet bound for User A, send it into the green tunnel (toward the radio gateway). We also tell the radio gateway: If you get a packet exiting the green tunnel, send it into the blue tunnel (toward the tower).

Notice that none of the network components are running a routing protocol to find paths. Instead, the manager is telling the routers how to forward packets. Each user will need their own set of tunnels, so the network is storing per-user state (e.g. one table entry for each connected user).

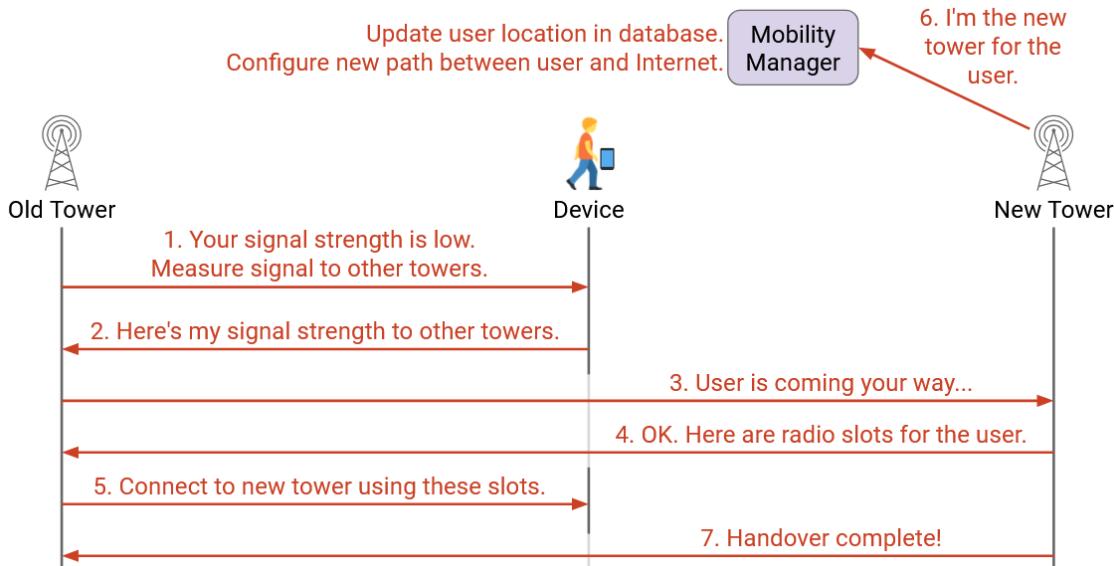
How do we actually implement these rules? For example, how does the radio gateway know when an incoming packet is coming out of the blue tunnel? We can use encapsulation. When entering a tunnel, we can add a new header, indicating that the packet is traveling through that tunnel (e.g. "this packet is traveling through the blue tunnel"). On the other end, when the packet exits the tunnel, the gateway looks at the extra header and knows which tunnel the packet came from. The gateway can then use this information to decide where to forward the packet next.



Notice that with tunnels and encapsulation, the routers are never forwarding based on the user's IP. The user is always moving around, so we can't use their IP to determine their location. Instead, we have to use these pre-configured tunnels to decide where to forward the packet.

Step 4: Handover

What happens if the user moves from one tower to another? Let's look at a (slightly simplified) protocol. We'll call the towers old and new, and move from the old tower to the new tower.



1. Your device is constantly measuring its signal strength to the old tower, and reporting that strength to the old tower. At some point, the old tower will say: Your signal strength is too low. Here are some

nearby towers (owned by the same operator) and their corresponding control channel frequencies. Can you measure your signal strength to these nearby towers?

2. Your device measures the signal strength to the nearby towers, and reports those values to the old tower. The old tower will pick the best new tower, based on whatever policy the operator wants.
3. The old tower tells the new tower: The user is coming your way. This causes the new tower to allocate some frequency resources to the user.
4. The new tower tells the old tower which frequency resources have been allocated.
5. The old tower tells the user: Connect to the new tower, using these frequencies.
6. The new tower reports to the mobility manager: I am the new tower for the user. The manager updates its database with the user's new location. The manager also updates the tunnels to create a new path between the user and the Internet (via a new tower, and also possibly via new radio and packet gateways).
7. Finally, the new tower tells the old tower that handover is complete.

Why was the handover process so complicated? Remember, we want to give the user seamless communication, with no interruption as they move between towers. This requires cooperation between the user, the old and new towers, the mobility manager, and the gateways.

Seamless communication is difficult because the handover process is not atomic. The user is still sending and receiving data while the handover is ongoing. For example, outside servers replying to the user might have sent a bunch of incoming packets to the old tower. During the handoff, the old tower continues to buffer any data it receives for that user. After the handoff, the old tower can send that buffered data to the new tower, which forwards that data to the user. Notice that traditional TCP/IP networks didn't need to buffer data like this. This type of buffering is a new feature added for seamless handovers as the user moves around.

Notice that the decisions in this handover process are always made by the operator. The device doesn't get to choose the next tower to use. The benefit of this design is, it gives the operator more control. For example, if a tower is overloaded, the operator can load-balance and send the user to a different tower. Or, if some users are prioritized over others, the operator can send less-prioritized users to worse towers. The drawback of this design is, it's a bit slower and requires more round-trips and more complexity.

Notice that the user's IP address remains unchanged during the handoff. We just updated the tunnels so that packets destined for the user's IP go through a different path.

Handovers are complicated and require updating the per-user state in the network. If the number of users increases, or users move around really quickly, this protocol encounters scaling challenges. And yet, the modern cellular network works pretty well at scale, because so much work has gone into optimizing these protocols. That's why the standards specifications are often thousands of pages long!

Roaming

Recall that a user can roam and connect to a different network if they're visiting another country (or any place where their own operator doesn't have coverage).

The connection process (discovery, attachment, handover) in a visiting network is generally pretty similar to connecting in the home network. The main difference is, the mobility manager in the visitor network

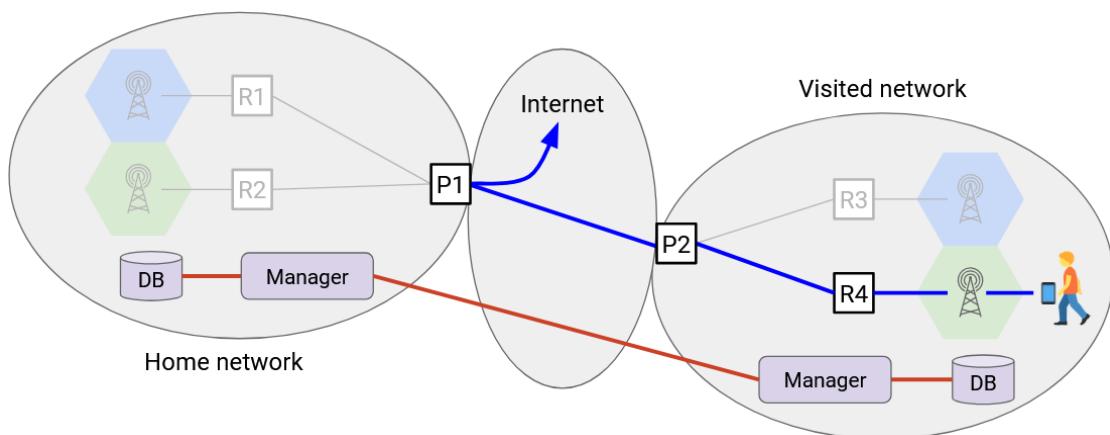
must communicate back to the mobility manager in the home network.

For example, the visitor needs to ask the home for help in authenticating the user (check if the user has paid for roaming). Also, the visitor needs to send tracking data back to the home network, so that the home network knows the user's location.

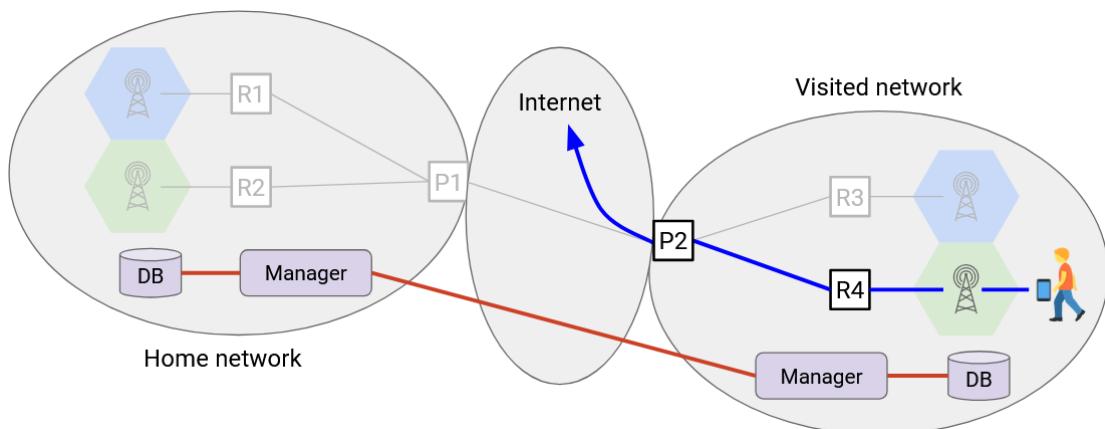
How does the visitor know where the home network is? Remember, during attachment, the device presents its IMSI, and the IMSI contains a Mobile Network Code which identifies the user's operator.

There are two different approaches to set up tunnels between the user and the Internet.

In the home routing approach, traffic is tunneled through the home network's packet gateway. This means that all packets must travel from the visiting network back to the home network, before getting forwarded to the wider Internet. This is beneficial because it lets the home network's packet gateway track the user. One drawback is, if you're a USA-based user, you roam in Germany, and you want to access a website in Germany, your packet must travel from Germany, back to the USA gateway, and then back to Germany.



In the local breakout approach, traffic is tunneled through the visiting network's packet gateway. This can shorten the route between the user and the Internet, since packets don't have to travel all the way back to the home network first. However, this can make accounting for the user's usage more complicated, since the roaming network must now do the accounting and send the data back to the home network.



Additional Operations

We've seen some of the key operations in cellular networks, but other operations exist as well.

Lawful intercept is a legal requirement for all cellular operators. This allows a government with a search warrant to wiretap your connection and listen to the packets you're sending.

Stolen phone registries allow a user to report their phone as stolen. Then, if the thief tries to connect your stolen phone to the network, the operator (manager and database) notice that the phone is stolen, and can try to track down the phone. Here, the operator uses the IMEI (the ID number hard-coded into your phone) to identify the specific phone (regardless of the IMSI, the subscriber ID). Devices need to report their IMEI when they connect, allowing the operator to check if the phone is stolen.

These additional operations are possible because the operator has centralized control, keeping track of all the users and their locations.

Cellular Network Design Reflections

As we noted earlier, cellular networks have different fundamental goals and design choices, compared to the traditional Internet. For example, we saw that authentication and accounting are central goals of the cellular network, even though these were not goals in the traditional Internet. We also saw that allocation is based on reservations, and the network maintains per-user state that is dynamically changing.

Using stateful reservation-based networks increased the complexity of our network. The various components had to constantly reconfigure tunnels as the user moved around.

Let's think about some possible alternate designs. Recall that handover was complicated because we wanted the user to keep the same IP address as they moved around. What if we instead changed the user's IP address on each handover? Now, the IP addresses actually reflect the user's location, and we could use traditional routing protocols again. However, higher-level protocols like TCP and HTTP will break. Remember, TCP relies on the two connecting users keep the same IP address.

Using the same IP address increases complexity, but changing IP addresses breaks TCP. One possible solution is to use a different transport-level protocol that allows changing IP addresses, like QUIC (developed at Google). Then, even though the IP addresses are changing, we can use the flow label field in the IPv6 header to label all the packets in a flow.