CS 6250: Computer Networking

Lecture 1: Introduction

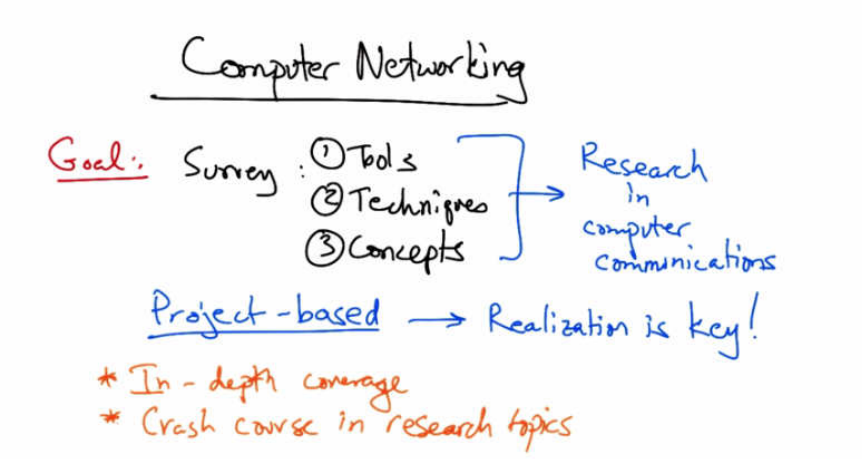
Welcome to Computer Networking

Welcome to networking. I'm Nick and I'll be your teacher in this course.

And I'm Josh. Nick and I have prepared a set of fun projects for you to tackle.

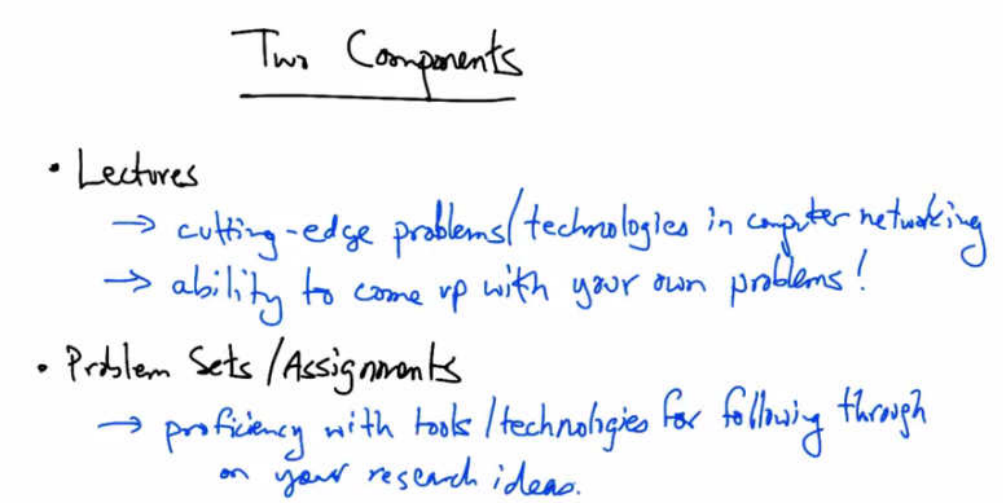
We have an awesome course prepared for you. We'll be covering advanced concepts in networking such as software defined networking (SDN), data center networking (DCN) and content distribution. You'll complete projects using a state of the art network emulator called mini-net to understand and explore these advanced concepts leading up to a final project replicating actual networking research.

Computer Networking



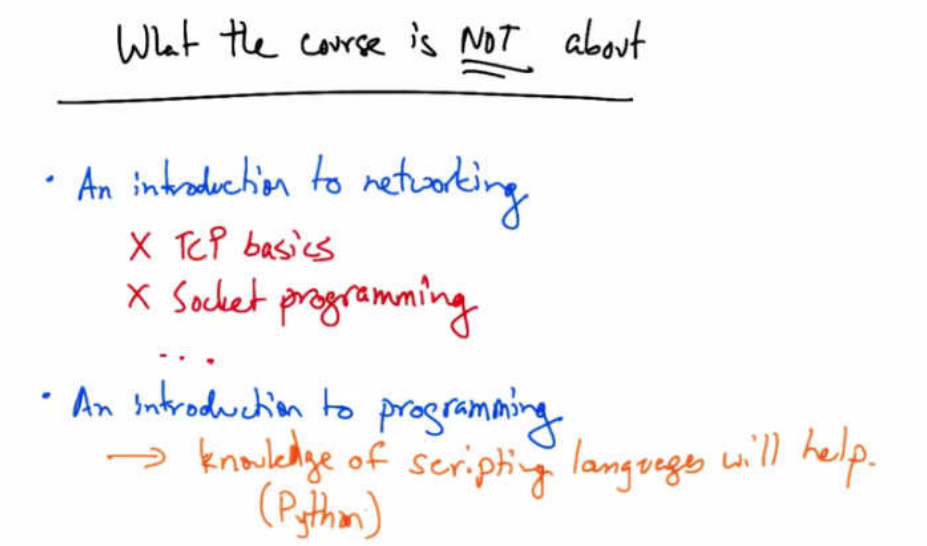
Welcome to the graduate course on computer networking. The primary goal of this course is to provide a survey of the necessary tools, techniques, and concepts to perform research in computer communications. This is a project based course, and there will be significant emphasis on hands-on experience. In networking, perhaps more than many other subjects, realization is key. You can read about concepts or techniques in a textbook, but really the most effective way to learn networking is by doing. So, you'll gain a lot of hands on experience in this course through the assignments. In comparison to an introductory networking course which you may have taken, this course will provide more in depth coverage of networking topics, and it will also offer a crash course in some of the available tools that are now available for performing research in computer networking. You will gain experience with many of these tools through the project based assignments in the course.

Two Components



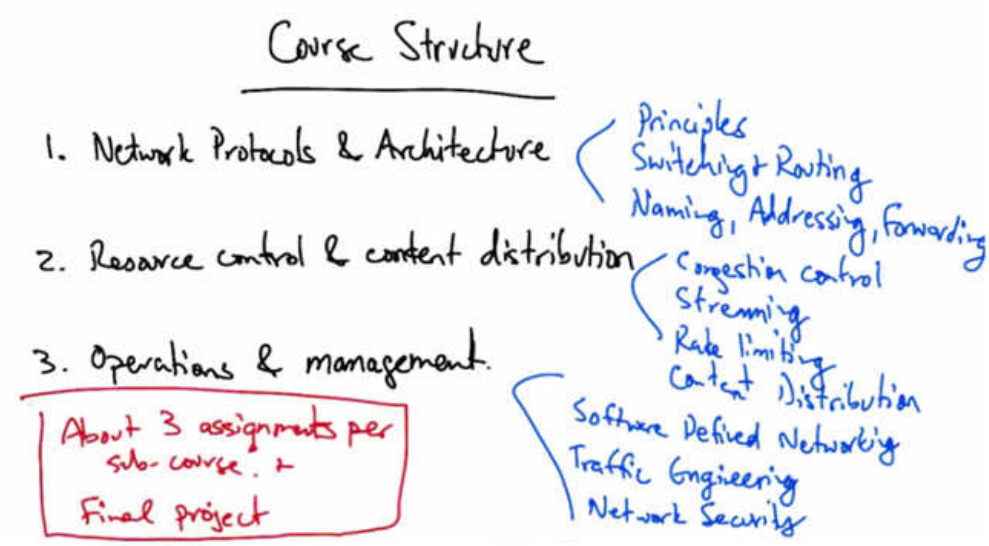
The course has essentially two components. In the lectures you will learn about cutting edge research problems in computer networking and you'll also gain the ability to come up with your own problems. We'll pick up the basics along the way as necessary. In addition to the lectures there are also a number of problem sets or assignments that you will work through as you work your way through the course. The problem sets and assignments in the course will give you proficiency with the tools and technologies that are state of the art in the research community. That will allow you to follow through on the research ideas that you may come up with as we work through various topics in the course. There are tons of exciting tools to use, and the problem sets and assignments will help you gain proficiency with them.

What the Course is NOT About



It's also worth bearing in mind what this course is not about. The course is not an introduction to networking, so there are a number of basic topics that won't be covered in this course. In particular, we'll assume that you're already familiar with the basics of things like TCP, Socket programming, and so forth. Anything that you might have picked up in an introductory networking course, we are just going to assume as a prerequisite for this course. So before you proceed, it may be worth revisiting some of your old undergraduate networking course material. The course is also not providing any introduction to programming. However, many assignments in the course will make use of some amount of programming. So some knowledge of scripting languages like Ruby, Python or Perl will certainly be helpful in some assignments. We'll be making a lot of use of a network emulation toolkit called Mininet, and to use that tool most effectively, you will certainly want to learn some Python if you don't already know it. Don't worry if you don't know these languages already, though. There's plenty of time to learn in the course since the deadlines are fairly spread out. And the assignments aren't focused on knowledge of programming per say, but rather, the concepts that you are going to realize in the programming languages.

Course Structure



The course is broken into three smaller sub-courses. The first course will cover topics including architectural principles, switching, routing, naming, addressing, and forwarding. The second part of the course will cover congestion control, streaming, rate limiting, and content distribution. And the third part of the course will have modules on software defined networking, traffic engineering, and network security. There will be about three assignments per sub course, plus a final project.

CS 6250: Computer Networking

Lecture 2: Architecture & Principles

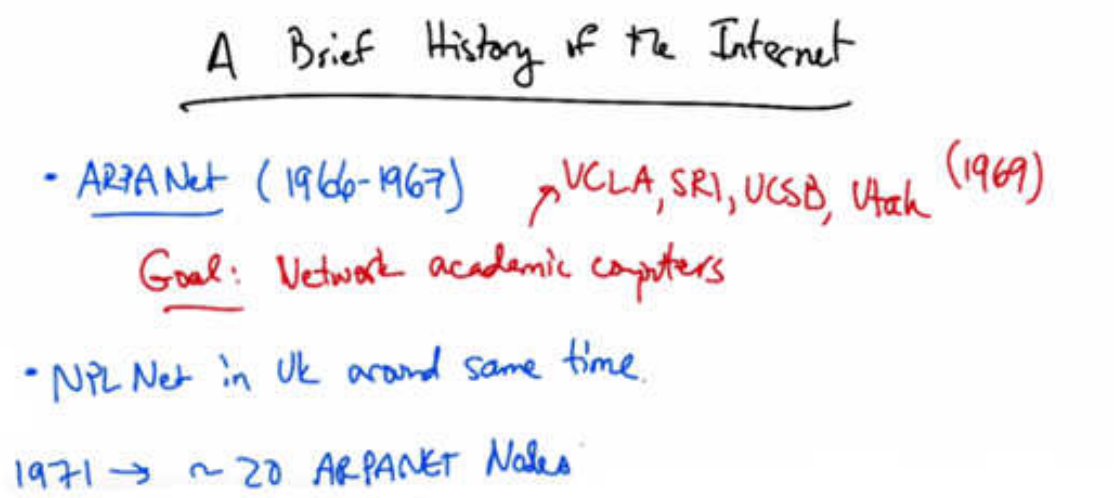
Lesson 2 Intro

We'll begin our foray into networking by reviewing the history of the internet and its design principles. Networking today is an eclectic mix of theory and practice in large part because the early internet architects set out with clear goals and allowed flexibility in achieving them.

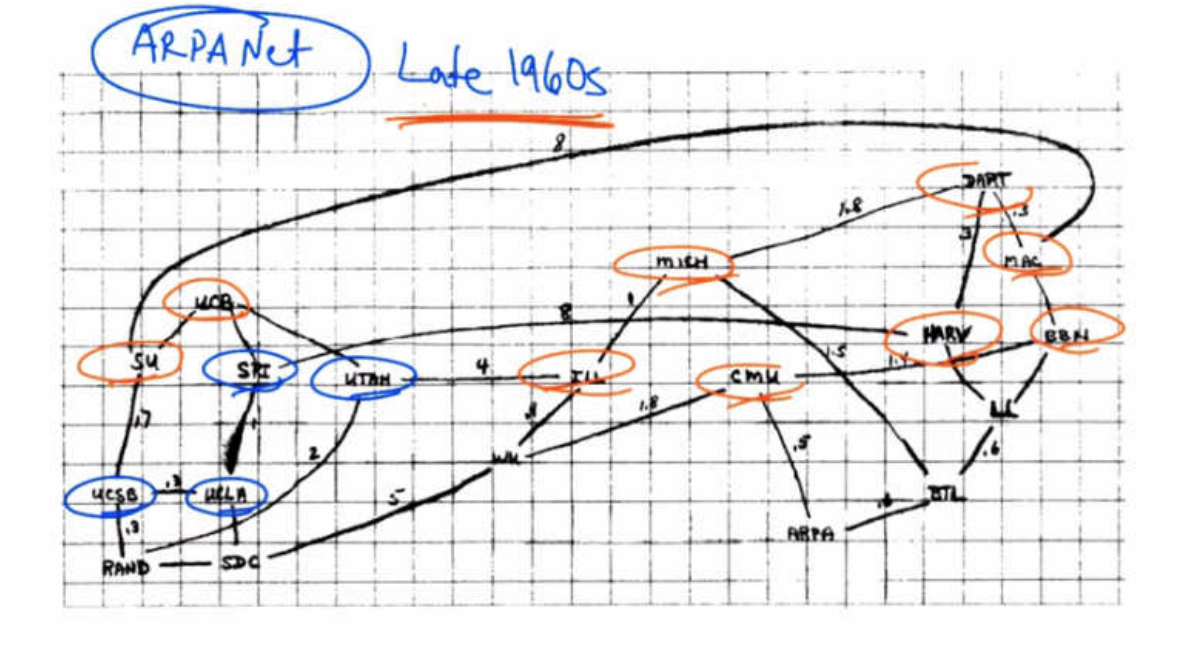
With all that flexibility, does that mean we'll see the rollout of IPv6 soon?

Only in your dreams.

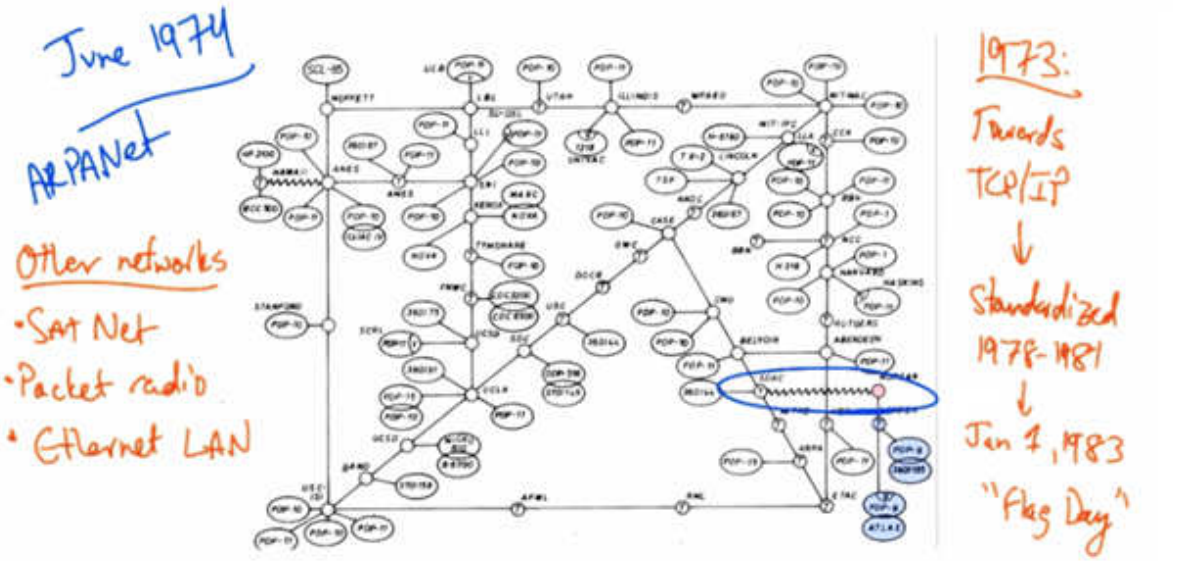
A Brief History of the Internet



In this lesson we will cover a brief history of the internet. The internet has its roots in the ARPA Net which was conceived in 1966 to connect big academic computers together. The first operational ARPA Net nodes came online in 1969 at UCLA, SRI, UCSB, and Utah. Around the same time, the National Physical Laboratory in the UK also came online. By 1971 there were about 20 ARPANet Nodes and the first host-to-host protocol. There were two cross country links, and all of the links were at 50 KBPS.

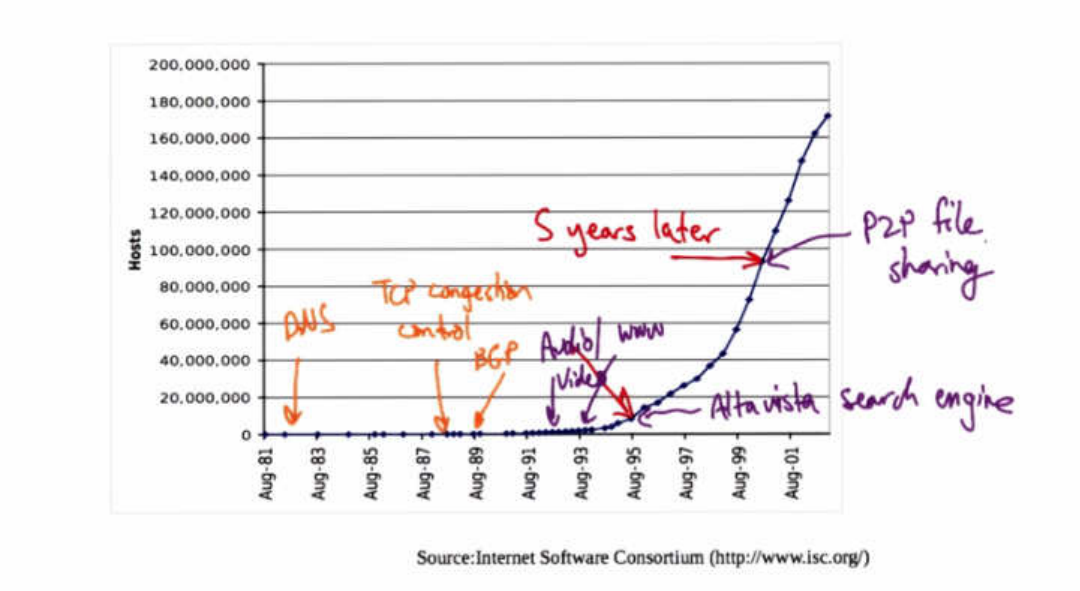


Here is a rough sketch of the ARPANet as drawn by Larry Roberts in the late 1960s. You can see the four original Nodes here, as well as some other well known players such as Berkeley, the MAC project at MIT, BBN, Harvard, Carnegie-Mellon, Michigan, Illinois, Dartmouth, Stanford, and so forth. This is what the ARPANET looked like in the late 1960s.



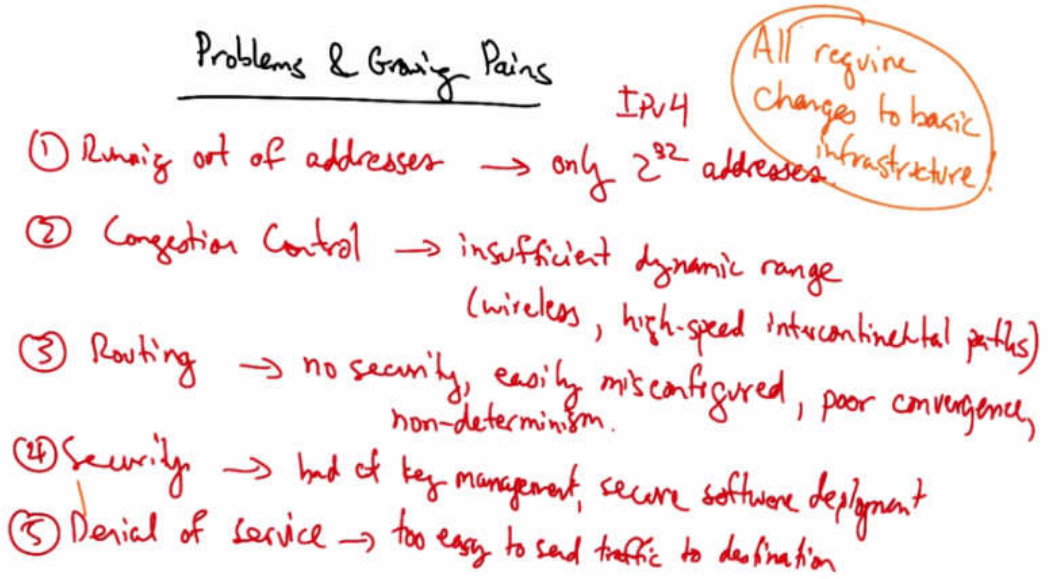
Here's a picture of the ARPANET in June 1974. And you can see not only some additional networks that have come online, but also a diagram of the machines that are connected at each of the universities. You can also see a connection here between the ArpaNet and MPLnet. Of course, the ArpaNet wasn't the only network. There were other networks at the time. Sat Net

operated over satellite. There were packet radio networks, and there were also Ethernet local area networks. Work started in 1973 on replacing the original network control protocol with TCP/IP where IP was the Internetwork Protocol and TCP was the Transmission Control Protocol.



TCP/IP was ultimately standardized from 1978 to 1981 and included in Berkley UNIX in 1981. And on January 1st, 1983 the internet had one of its flag days, where the ArpaNet transitioned to TCP/IP. Now the internet continued to grow, but the number of computers on the internet really didn't start to take off until the mid 90s. You can see here that around August 1995 there were about 10 million hosts on the internet, and five years later there was an order of magnitude more hosts on the internet—more than 100 million. During this period the Internet experienced a number of technical milestones. In 1982 the internet saw the rollout of the domain name system which replaced the host.txt file containing all the world's machine names with a distributed name lookup system. 1988 saw the rollout of TCP Congestion Control after the net suffered a series of congestion collapses. 1989 saw the NSF net and BGP inter-domain routing including support for routing policy. The 90s, on the other hand, saw a lot of new applications. In approximately 1992 we started to see a lot of streaming media including audio and video . Web was not soon after, in 1993, which allowed users to browse a mesh of hyperlinks. The first major search engine was Altavista, which came online in December of 1995, and peer to peer protocols and applications including file sharing, began to emerge around 2000.

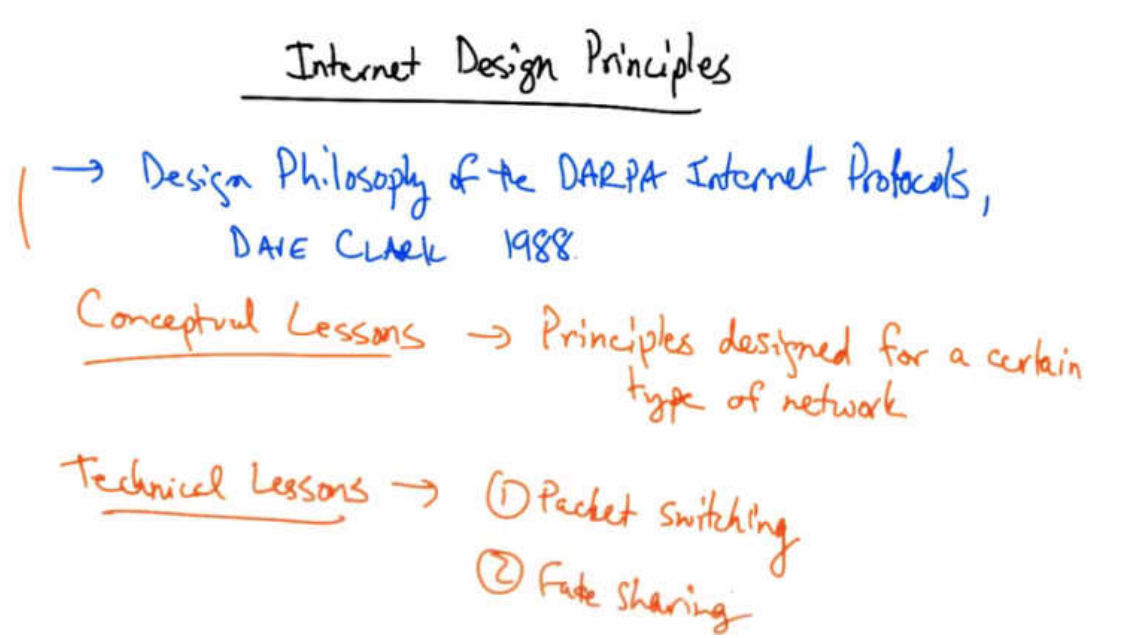
Problems and Growing Pains



Now, today's internet is experiencing considerable problems and growing pains, and it's worth bearing some of these in mind and thinking about them, as many of them give rise to interesting research problems to think about as we work through the material in the course. One of the major problems is that we're running out of addresses. The current version of the internet protocol, IPV4, uses 32-bit addresses, meaning that the IPV4 internet only has 2 to the 32 IP addresses, or about 4 billion IP addresses. Furthermore, these IP addresses need to be allocated hierarchically and many portions of the IP address space are not allocated very efficiently. For example, the Massachusetts Institute of Technology has one two fifty sixth of all the Internet address space. Another problem is congestion control. Now congestion control's goal is to match offered load to available capacity. But one of the problems with today's congestion control algorithms is that they have insufficient dynamic range. They don't work very well over slow and flaky wireless links and they don't work very well over very high speed intercontinental paths. Now, some solutions exist but change is hard and all solutions that are deployed must interact well with one another. And deployment in some sense requires some amount of consensus. A third major problem is routing. Routing is the process by which those on the internet discover paths to take to reach another destination. Today's interdomain routing protocol, BGP, suffers a number of ills, including a lack of security, ease of misconfiguration, poor convergence, and non-determinism. But it sort of works and it's the most critical piece of the internet infrastructure in some sense because it's the glue that holds all of the internet service providers together. Another major problem in today's internet is security. Now while we're reasonably good at encryption and authentication, we are not actually so good at turning these mechanisms on. And we're pretty bad at key management, as well as deploying secure software and secure configurations. The fifth major problem is denial of service. And the internet does a very good job of transmitting packets to a destination even if the destination doesn't want those packets. This makes it easy for an attacker to overload servers or network links to prevent the victim from doing useful work. Distributed denial of service attacks are particularly commonplace on today's Internet. Now, the thing that all of those problems have in common is that they all require changes to the basic

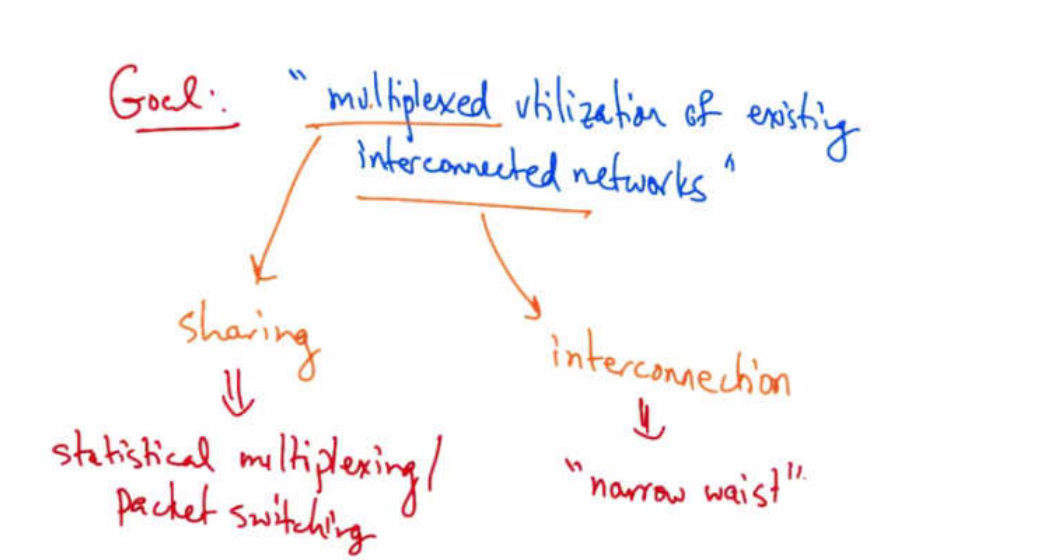
infrastructure, and changing basic infrastructure is really difficult. It's not even clear what the process is to achieve consensus on changes. So as we work our way through the course, it will be interesting to see the problems that we encounter in each of these areas, various solutions that have been proposed, and also to think about ways in which new protocols and technologies can be deployed. In later parts of the course we'll learn about a new technology called software defined networking, or SDN. That makes it easier to solve some of these problems by rolling out new software technologies, protocols, and other systems to help manage some of these issues.

Architectural Design Principles



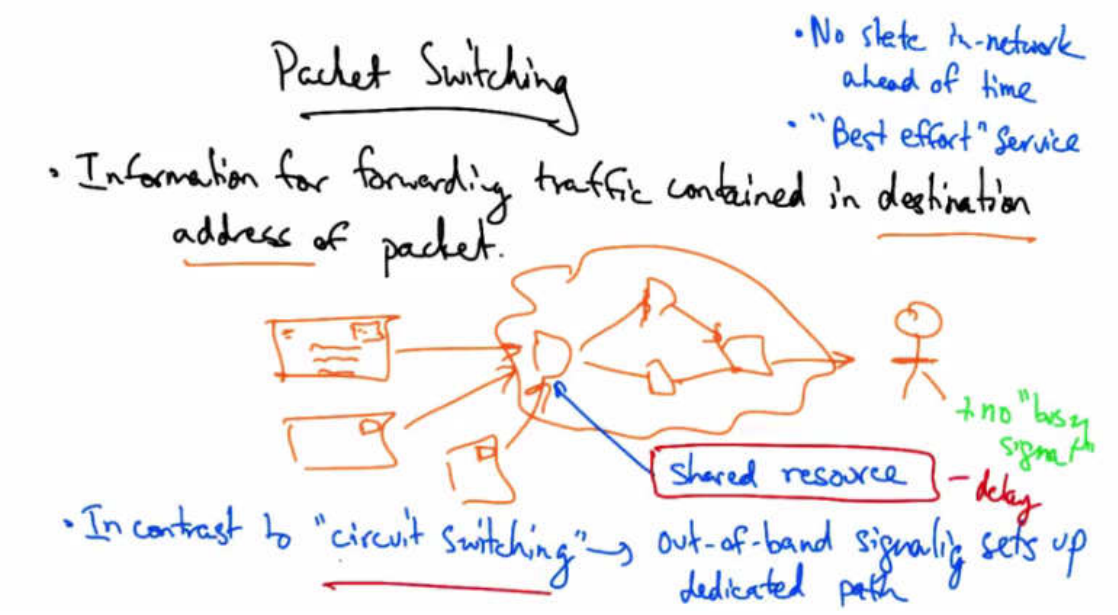
In this lecture we will talk about the Internet's original design principles. These design principles were discussed in the paper reading for today, the Design Philosophy of the DARPA Internet Protocols, by Dave Clark, dated 1988. The paper has many important lessons, and we will go through many of them as we revisit many of the design decisions. Before we jump into any details let's talk about some of the high level lessons. One of the most important conceptual lessons is that the design principles and priorities were designed for a certain type of network. And as the internet evolves, we are feeling some of the growing pains of some of those choices. In the last lesson we talked about a number of the problems and growing pains of the internet. And it's worth bearing in mind that many of the problems that we are seeing now, are a result of some of the original design choices. Now that's not to say that some of these design choices are right or wrong, but rather that they simply reflect the nature of our understanding at the time, as well as the environment and constraints that the designers faced for the particular network that existed at that time. Now needless to say, some of the technical lessons from the original design have turned out to be fairly timeless. One concept is packet switching, which we will discuss in this lesson. And another is the notion of fate sharing, or soft state, which we will discuss in a subsequent lesson in the course.

Goal



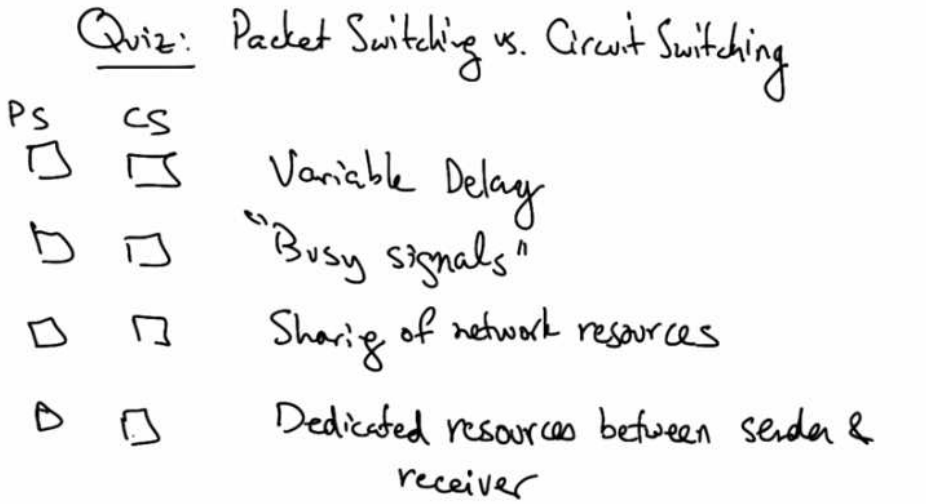
The fundamental design goal of the internet was multiplexed utilization of existing interconnected networks. There are two important aspects to this goal. One is multiplexing or sharing. So one of the fundamental challenges that the internet technologies needed to solve was the shared use of a single communications channel. The second major part of this fundamental goal is the interconnection of existing networks. These two sub problems had two very important solutions. Statistical multiplexing, or packet switching, was invented to solve the sharing problem, and the narrow waist was designed to solve the problem of interconnecting networks. Let's talk about each of these now in turn. We'll first talk about packet switching

Packet Switching



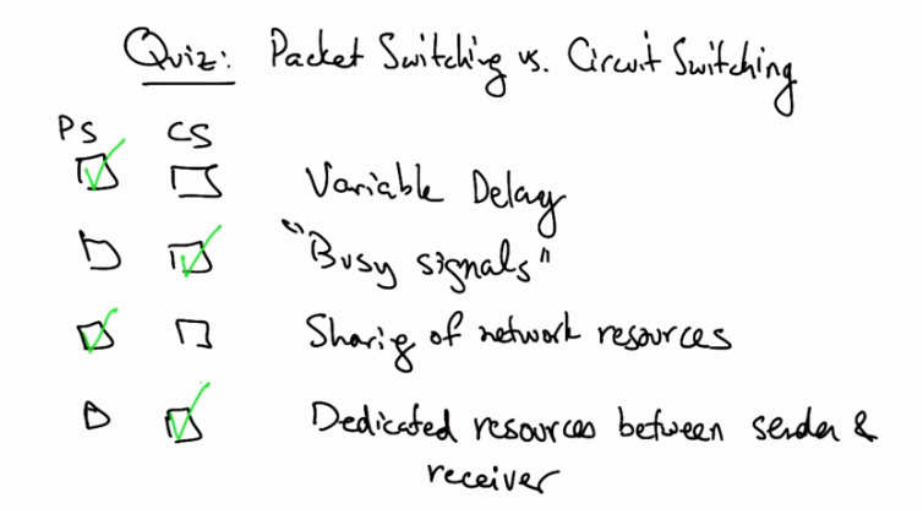
In packet switching, the information for forwarding traffic is contained in the destination address of every datagram or packet. Similar to how you would write a letter and specify the destination to where you want the letter sent, and that letter might wend its way through multiple intermediate post offices en-route to the recipient, packet switching works much the same way. There is no state established ahead of time, and there are very few assumptions made about the level of service that the network provides. This assumption about the level of service that the network provides, is sometimes called best effort. So how does packet switching enable sharing? Just as if you were sending a letter, many senders can send over the same network at the same time, effectively sharing the resources in the network. A similar phenomenon occurs in packet switching when multiple senders send network traffic or packets over the same set of shared network links. Now this is in contrast to the phone network, where if you were to make a phone call, the resources for the path between you and the recipient are dedicated and are allocated until the phone call ends. The mode of switching that the conventional phone network uses is called circuit switching, where a signaling protocol sets up the entire path, out-of-band. So this notion of packet switching and statistical multiplexing, allowing multiple users to share a resource at the same time, was really revolutionary. And it is one of the underlying design principles of the internet that has persisted. Now, an advantage of statistical multiplexing of the links and the network means that the sender never gets a busy signal. The drawbacks include things like variable delay and the potential for lost or dropped packets. In contrast, circuit switching provides resource control, better accounting and reservation of resources, and the ability to pin paths between a sender and receiver. Packet switching provides the ability to share resources and potentially better resilience properties.

Packet Switching vs Circuit Switching Quiz



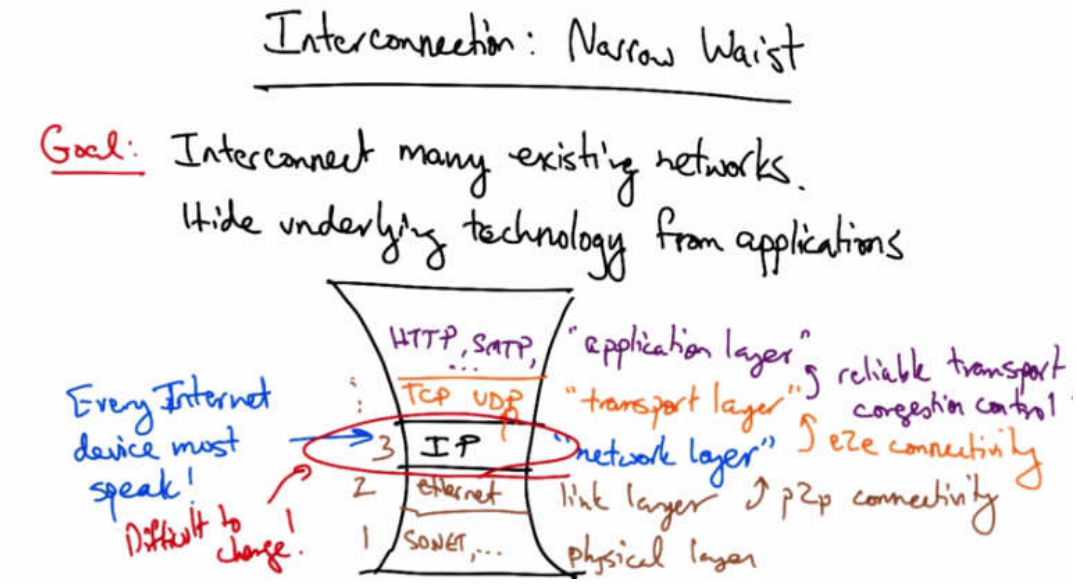
Let's take a quick quiz on packet switching versus circuit switching. Which of the following are characteristics of packet switching and circuit switching: variable delay, busy signals, sharing of network resources like an end-to-end path among multiple recipients, and dedicated resources between the sender and receiver? Each of these options only has one correct answer.

Packet Switching vs Circuit Switching Solution



Variable delay is a property of statistical multiplexing, or packet switching. Circuit switch networks can have busy signals. Packet switch networks share network resources. And circuit switch networks typically have dedicated resources along a path between the sender and receiver

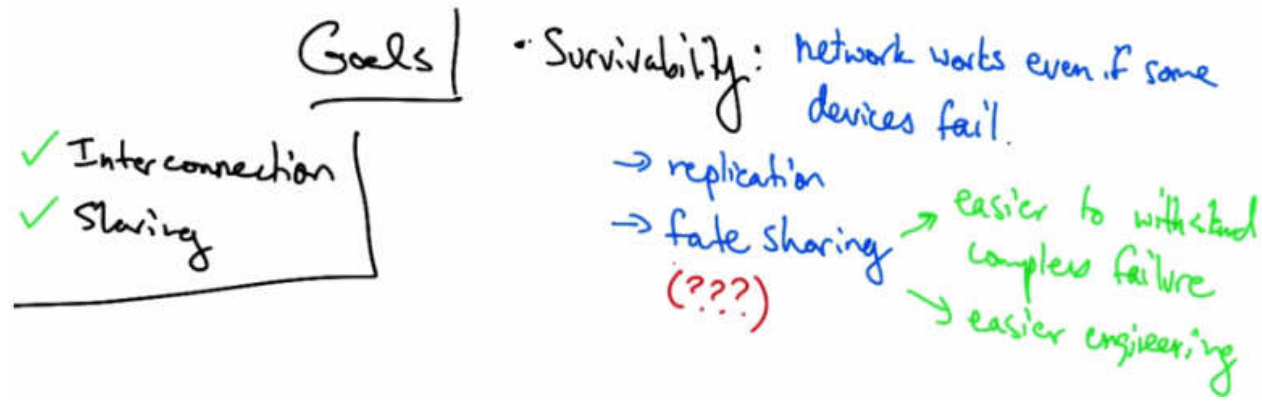
Narrow Waist



Let's now take a look at the second important fundamental design goal on the internet, interconnection, and how interconnection is achieved with the design principle called the Narrow Waist. Let's keep in mind that one of the main goals was to interconnect many existing networks, and to hide the underlying technology of interconnection from applications. This design goal was achieved using a principle called the narrow waist. The internet architecture has

many protocols that are layered on top of one another. At the center is an interconnection protocol called IP, or the internet protocol. Now every internet device must speak IP or have an IP stack. Given that a device implements the IP stack, it can connect to the internet. This layer of the network is sometimes called the network layer. Now this layer provides guarantees to the layers above. On top of the network layer sits the transport layer. The transport layer includes protocols like TCP and UDP. The network layer provides certain guarantees to the transport layer. One of those guarantees is end to end connectivity. For example, if a host has an IP address, then the network layer, or IP, provides the guarantee that a packet with that host destination IP address should reach the destination with the corresponding address with best effort. On top of the transport layer sits the application layer. The application layer includes many protocols that various internet applications use. For example, the web uses a protocol called the hypertext transfer protocol or HTTP. And mail uses a protocol called SMTP or simple mail transfer protocol. Transport layer protocols provide various guarantees to the application layer including reliable transport or congestion control. Now below the network layer we have other protocols. The link layer provides point-to-point connectivity, or connectivity on a local area network. A common link layer protocol is Ethernet. Below that, we have the physical layer, which includes protocols such as sonnet or optical networks and so forth. The physical layer is sometimes called layer 1. The link layer is sometimes called layer 2 and the network layer is sometimes called layer 3. We tend to not refer to layers above the network layer by number. The most critical aspect of this design is that the network layer essentially only has one real protocol in use, and that's IP. That means that every device on the network must speak IP, but as long as the device speaks IP it can get on the internet. This is sometimes called IP over anything, or anything over IP, now the advantage to the narrow waist, as I mentioned, is that it is fairly easy to get a device on the network if it runs IP, but the drawback is that because every device is running IP, it's very difficult to make any changes at this layer. However, people are trying to do so, and later in the course, when we discuss software defined networking, we will explore how various changes are being made to both the IP layer, and other layers that surround it.

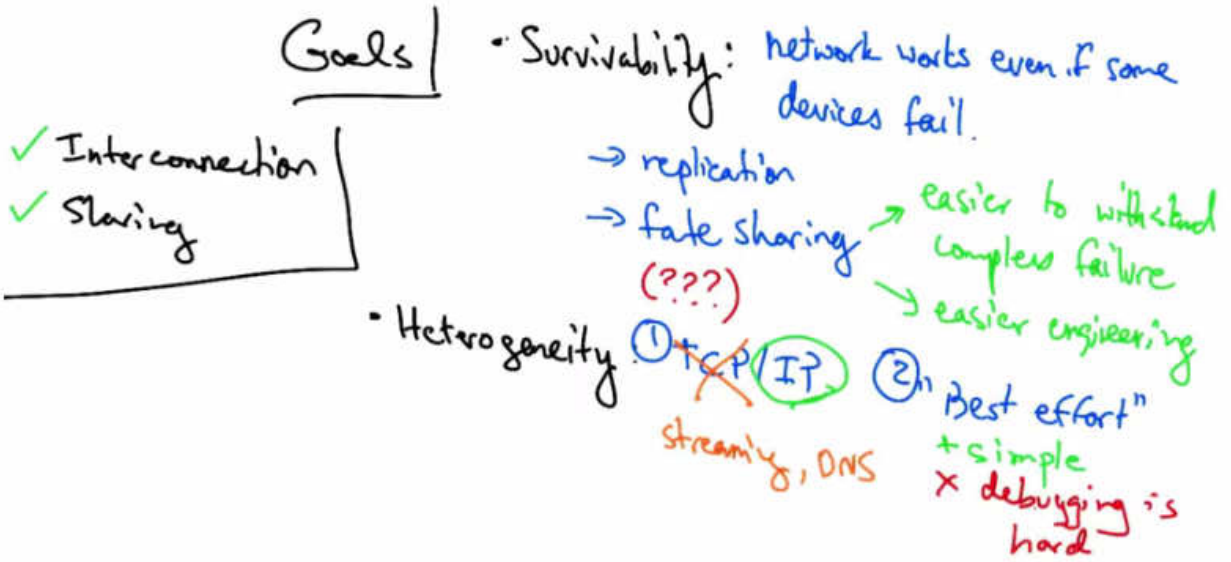
Goals Survivability



So we talked about how the internet satisfies the goals of sharing and interconnection and now let's talk about some of the other goals that are discussed in the DARPA Design Philosophy Paper. As we discuss some of these other goals it's worth considering and thinking about how well the current internet satisfies these other design goals in the face of evolving applications,

threats, and other challenges. One of the goals discussed is survivability, which states that the network should continue to work if even some devices fail, are comprised, and so forth. There are two ways to achieve survivability. One is to replicate. So one could keep state at multiple places in the network, such that when any node crashes there's always a replica or hot standby waiting to take over for the failure. Another way to design the network for survivability is to incorporate a concept called fate sharing. Fate sharing says that it's acceptable to lose state information for some entity, if that entity itself is lost. For example, if a router crashes all of the state on the router, such as the routing tables, are lost. If we can design the network to sustain these types of failures, where the state of a particular device shares the fate of the device itself, then we can withstand failures better. So fate sharing makes it easier to withstand complex failure scenarios and engineering is also easier. Now it's worth asking whether the current internet still satisfies the principle of fate sharing. In a subsequent lesson, we'll talk about network address translation and how it violates the notion of fate sharing. There are other examples where the current internet's design violates fate sharing and it's worth thinking about those.

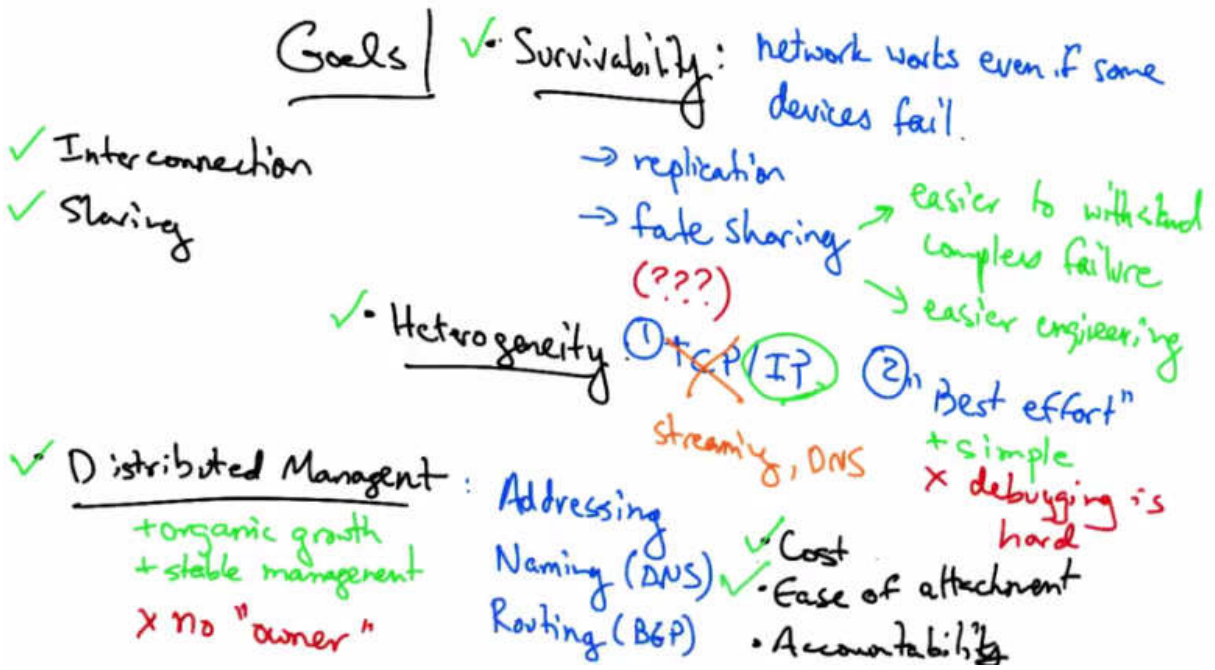
Goals Heterogeneity



The internet supports heterogeneity through the TCP/IP protocol stack. TCP/IP was designed as a monolithic transport, where TCP provided flow control and reliable delivery, and IP provided universal forwarding. Now it became clear that not every application needed reliable, in-order delivery. For example, streaming voice and video often perform well, even if not every packet is delivered. And the domain name system, which converts domain names to IP addresses, often also doesn't need completely reliable, in-order delivery. Fortunately, the narrow waste of IP allowed the proliferation of many different transport protocols, not just TCP. The second way that the internet's design accommodates Heterogeneity is through a best-effort service model, whereby the network can lose packets, deliver them out of order, and doesn't really provide any quality guarantees. It also doesn't provide information about failures, performance, et cetera. On

the plus side, this makes for a simple design, but it also makes certain kinds of debugging and network management more difficult.

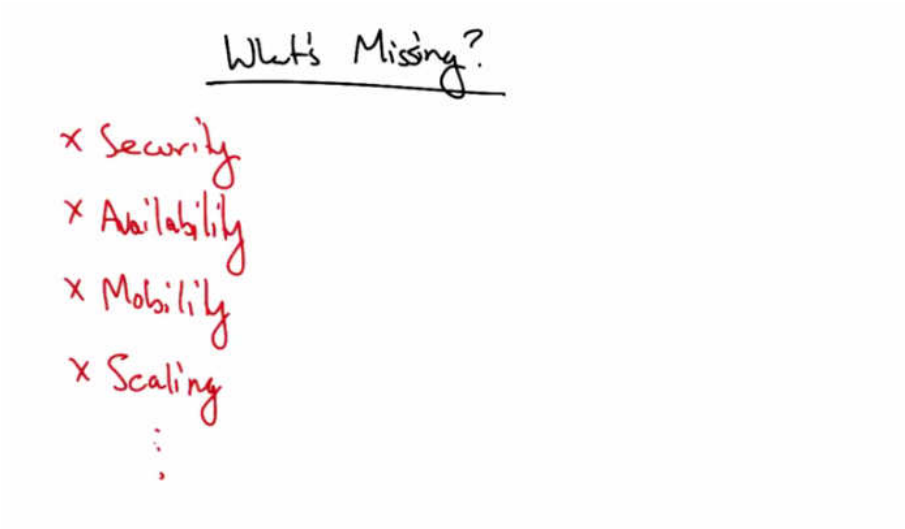
Goals Distributed Management



Another goal of the internet was distributed management. And there are many examples where distributed management has played out. In addressing, we have routing registries. For example, in North America we have ARIN, or the American Registry for Internet Numbers. And in Europe that same organization is called RIPE. DNS allows each independent organization to manage its own names and BGP allows each independently operated network to configure its own routing policy. This means that no single entity needs to be in charge and thus allows for organic growth and stable management. On the downside, the internet has no single owner or responsible party. And as Clark said, some of the most significant problems with the internet relate to the lack of sufficient tools for distributed management, especially in the area of routing. In such a network where management is distributed it can often be very difficult to figure out who or what is causing a problem, and worse, local action such as misconfiguration in a single local network can have global effects. The other three design goals that Clark discusses are cost effectiveness, ease of attachment, and accountability. It's reasonable to argue that the network design is fairly cost effective as is and current trends are aiming to exploit redundancy even more. For example, we will learn about content distributions and distributed web caches that aim to achieve better cost effectiveness for distributing content to users. Ease of attachment was arguably a huge success. IP is essentially plug and play. Anything with a working IP stack can connect to the internet. There's a really important lesson here, which is that if one lowers the barrier to innovation, people will get creative about the types of devices and applications that can run on top of the internet. Additionally, the narrow waist of IP allows the network to run on a wide variety of physical layers ranging from fiber, to cable, to wireless and so forth. Accountability, or the ability to essentially, bill, was mentioned in some of the early papers on

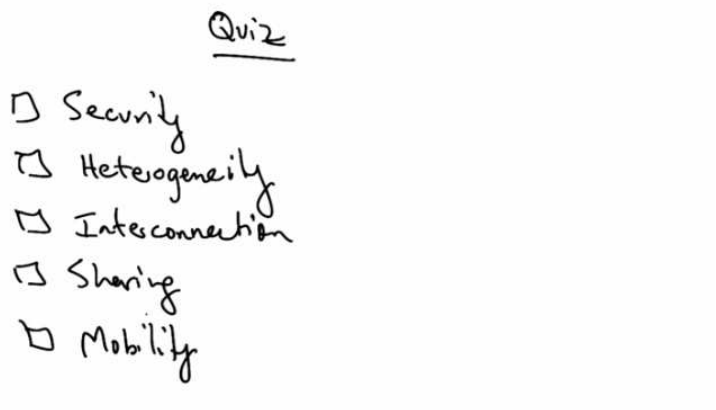
TCP/IP but it really wasn't prioritized. Datagram networks can make accounting really tricky. Phone networks had a much easier time figuring out how to bill users. Payments and billing on the internet are much less precise, and we'll talk about these more in later lectures.

What's Missing



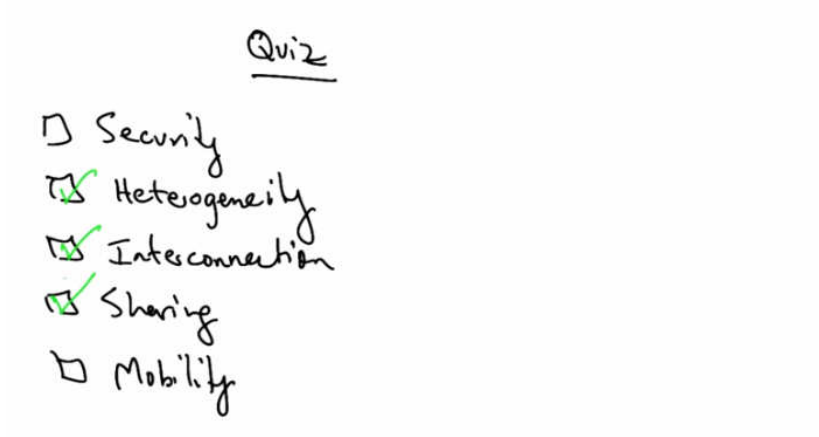
It's also worth noting what's missing from Clark's paper. There's no discussion of security. There's no discussion of availability. There's no discussion of mobility or support for mobility. And there's also no mention of scaling. There are probably a lot of other things that are missing and it's worth thinking about on your own, some of the other things that current internet applications demand, that are not mentioned in Clark's original design paper.

DARPA Paper Quiz



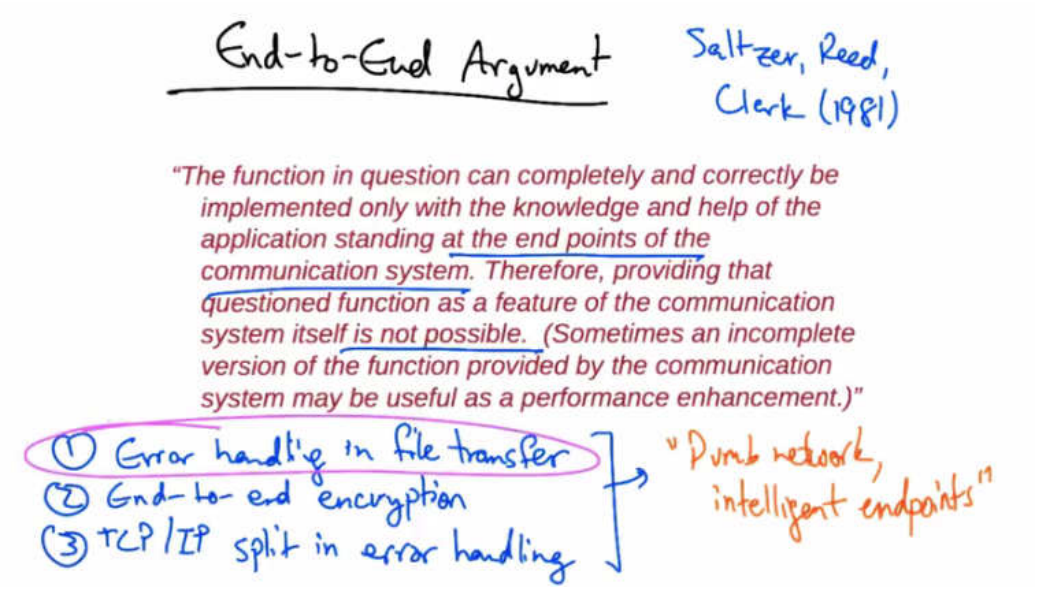
So as a quick quiz, can you quickly check all of the design goals in the list that were mentioned in Clark's original design goals paper? Security, support for heterogeneity, support for interconnection, support for sharing and support for mobility.

DARPA Paper Solution



Clark's original design goals, paper, mentions the need to support heterogeneity, interconnection and sharing.

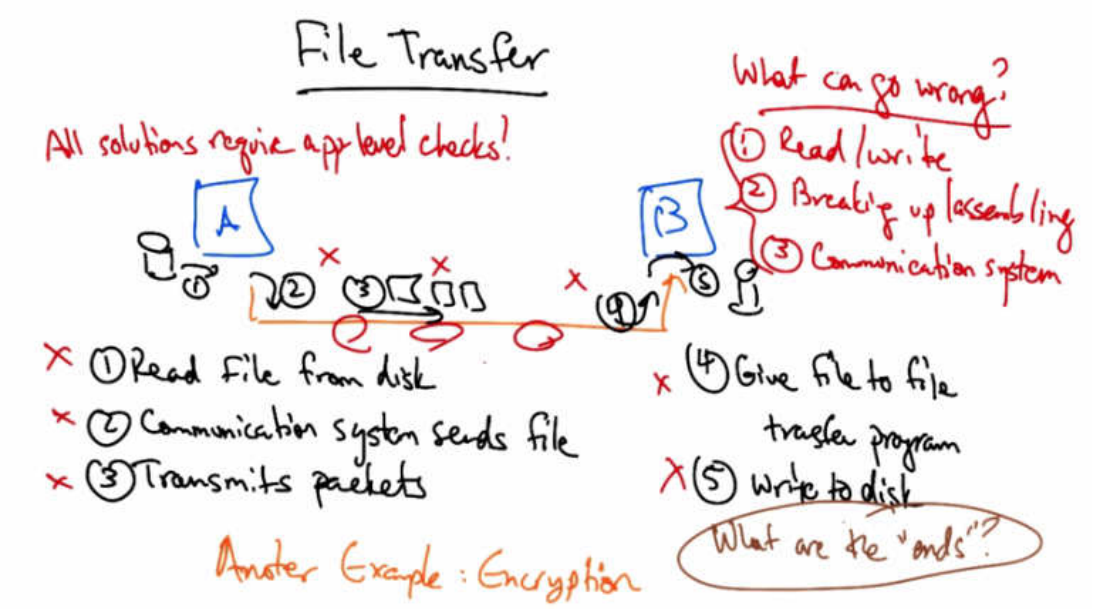
End-to-End Argument



In this lesson, we'll cover the End-to-End Argument as discussed in the paper, End-to-End Arguments in System Design by Saltzer, Reed, and Clark in 1981. In a nutshell, the End-to-End Argument reads as follows, "The function in question can completely and correctly be implemented only with the knowledge and application standing at the end points of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible." Essentially, what the argument says is that the intelligence required to implement a particular application on the communication system should be placed at the endpoints, rather than in the middle of the network. Commonly used examples of the end-to-end argument include error handling and file transfer, encrypting end-to-end versus

hop-by-hop in the network, and the partition of TCP and IP of error handling, flow control, and congestion control. Sometimes the end-to-end argument is summarized as, "the network should be dumb and minimal and the end points should be intelligent." Many people argue that the end-to-end argument allowed the internet to grow rapidly, because innovation took place at the edge in applications and services, rather than in the middle of the network, which can be hard to change sometimes. Let's look at one example of the end-to-end argument, error handling in file transfer.

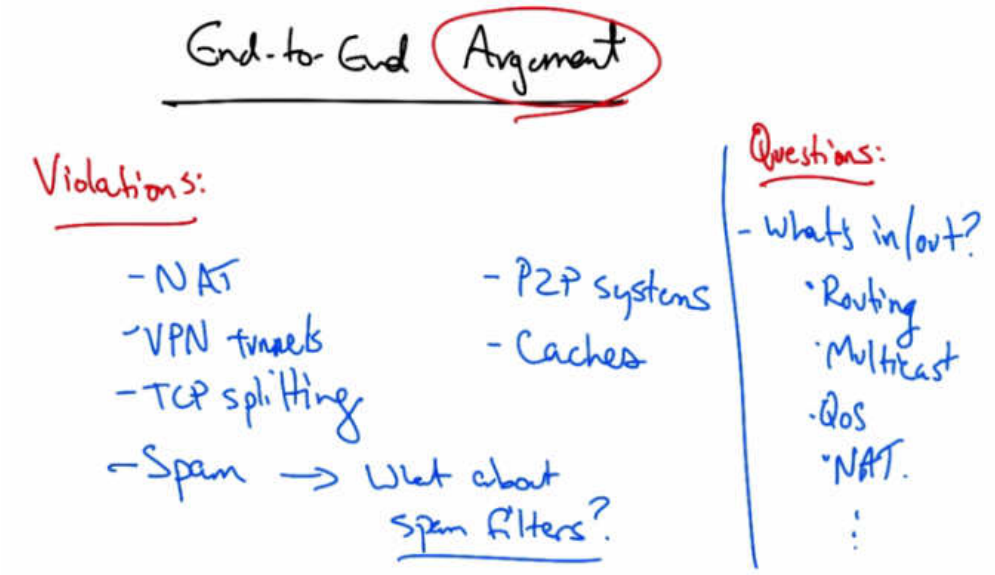
File Transfer



Let's suppose that computer A wants to send a file to computer B. The file transfer program on A asks the file system to read the file from the disk. The communication system then sends the file, and finally the communication system sends the packets. On the receiving side, the communication system gives the file to the file transfer program on B, and that file transfer program asks to have the file written to disk. So what can go wrong in this simple file transfer setup? Well, first, reading and writing from the file system can result in errors. There may be errors in breaking up and reassembling the file. And, finally, there may be errors in the communication system itself. Now, one possible solution is to ensure that each step has some form of error checking, such as duplicate copies, redundancy, time out and retry, so forth. One might even do packet error checking at each hop of the network. One could send every packet three times. One might acknowledge packet reception at each hop along the network. But the problem is that none of these solutions are complete. They still require application level checking. Therefore it may not be economical to perform redundant checks at different layers and at different places of this particular operation. Another possible solution is an end-to-end check and retry where the application commits or retries based on the check sum of the file. If errors along the way are rare, this will most likely finish on the first try. Now, this is not to say that we shouldn't take steps to correct errors at any one of these stages. Error correction at lower levels can sometimes be an effective performance booster. And the trade off here is based on

performance, not correctness. So whether or not one should implement additional correctness checks at these layers depends on whether or not the amount of effort put into the reliability gains are worth the extra trouble. Another example where the intend argument applies is with encryption, where keys are maintained by the end applications, and cipher text is generated before the application sends the message across the network. Now one of the key questions in the end-to-end argument is identifying the ends. The end-to-end argument says that the complexity should be implemented at the ends but not in the middle, but the ends may vary depending on what the application is. So for example, if the application or protocol involves Internet routing, the ends may be routers, or they might be ISPs. If the application or protocol is a transport protocol, the ends might be end hosts. So, identifying the ends in the end-to-end argument is always a thorny question that you have to answer first.

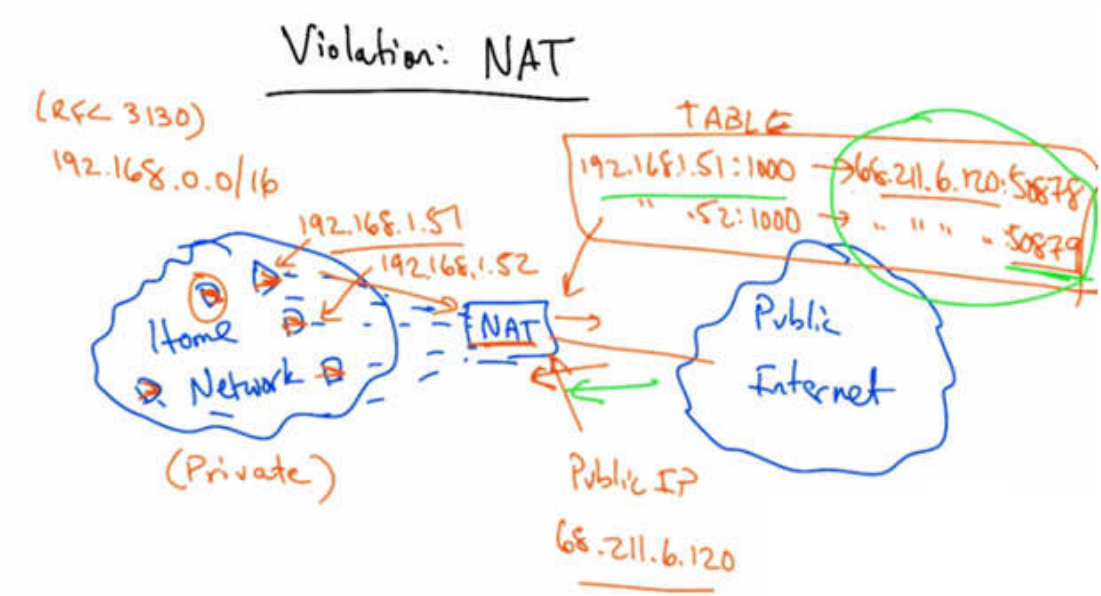
End-to-End Argument Violations



Now, when talking about the end-to-end argument, it is worth remembering that the end-to-end argument is just that. It's an argument. Not a theorem, or a principle, or a law. And there are many things that have come to violate the end-to-end principle. Network address translators, which we'll talk about in the next lesson, violate the end-to-end argument. VPN tunnels, which tunnel traffic between intermediate points on a network, violate the end-to-end argument. Sometimes TCP connections are split at an intermediate node along an end-to-end path, particularly when the last hop of the end-to-end path is wireless. This is sometimes done to improve the performance of the connection because loss on the last hop lossy wireless hop may not necessarily reflect congestion, and we don't necessarily want TCP to react to losses that are not congestion related. Even spam, in some sense, is a violation of the end-to-end argument. For e-mail the end user is generally considered to be a human, and by the end-to-end argument, the network should deliver all mail to the user. Does this mean that spam control mechanisms are in violation of end-to-end, and if so are these violations appropriate? What about peer to peer systems where files are exchanged between two nodes on the Internet but are assembled in

chunks that are often traded among peers? What about caches, and in-network aggregation? So, when considering the end-to-end argument, it's worth asking whether or not the argument is still valid today and in what cases. There are questions about what's in versus out, certainly, and what functions belong in the dumb minimal network. For example, routing is currently in the dumb minimal network. Do we really believe that it belongs? What about multicast? Mobility quality of service? What about NAT's? And it's worth considering whether the end-to-end argument is constraining innovation of the infrastructure by preventing us from putting some of the more interesting or helpful functions inside the network. In the third course, we will talk about software defined networking, which in some sense reverses many aspects of this end-to-end argument.

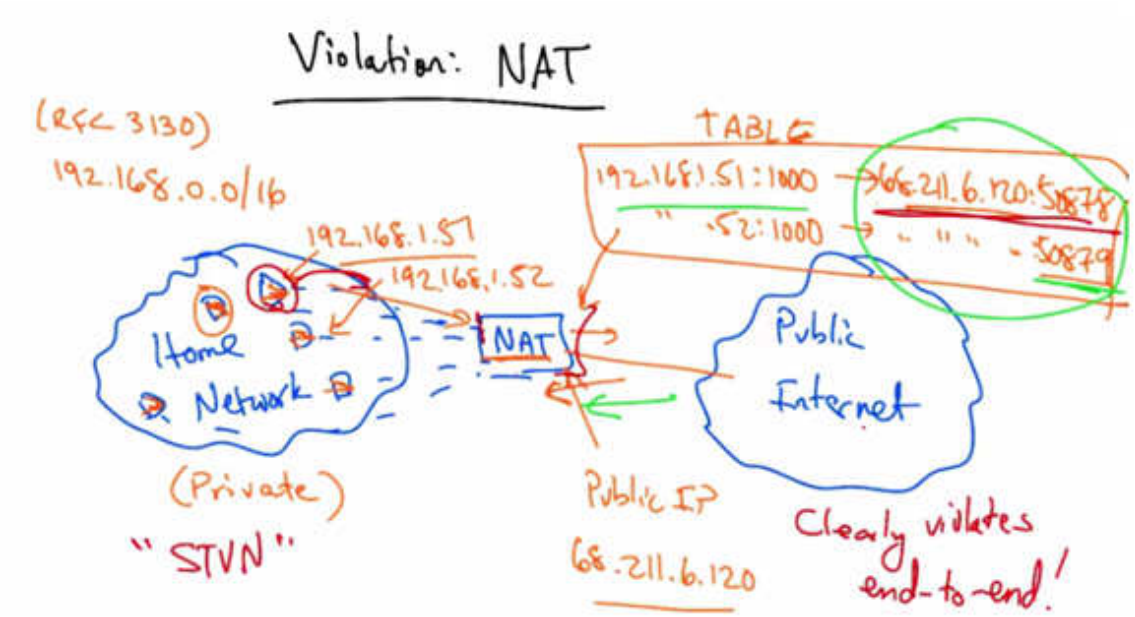
Violation NAT Part 1



A fairly pervasive violation of the end-to-end argument are home gateways, which often perform something called network address translation. Now on a home network we have many devices that connect to the network, but when we buy service from our internet service provider we're typically only given one public IP address. And yet we have a whole variety of devices that we may want to connect. Now the idea behind network address translation is that we can give each of these devices a private IP address and there are designated regions of the IP address space that are for private IP addresses. One of those is 192.168.0.0/16 and there are others, which you can go read about in RFC 3130. Each one of these devices in the home gets its own private IP address. The public internet, on the other hand, sees a public IP address which typically is the IP address provided by the internet service provider. When packets traverse the home router, which is often running a network address translation process, the source address of every packet is rewritten to the public IP address. Now when traffic comes back to that public IP address, the network address translator needs to know which device behind the NAT the traffic should be sent to. So it uses a mapping of port numbers to identify which device the return traffic should be sent to in the home network. So the NAT or the network address translator maintains a table that

says packets with the source IP address of 192.168.1.51 and source port 1000 should be rewritten to a source address of the public IP address and a source port of 50878. Similarly, packets with a source IP address of 192.168.1.52 and source port of 1000 should be rewritten to the public IP address and a source port of 50879. Then when traffic returns to the NAT to one of these addresses the NAT knows that it needs to rewrite the destination address on the return traffic to the appropriate destination IP address and port that's in the private network. So for outbound traffic, the NAT device creates a table entry mapping the computer's local IP address and port number to the public IP address at a different port number and replaces the sending computer's non-routable IP address with the gateway or the NAT public IP address. It also replaces the sender's source port with a different source port that allows it to de-mutiplex the packets sent to this return address and port. For inbound traffic to the home network, the NAT checks the destination port on the packet, and based on the port, it rewrites the destination IP address and port to the private IP address in the table before forwarding the traffic to a local device in the home network.

Violation NAT Part 2



Now the NAT clearly violates the end-to-end principle, because machines behind the NAT are not globally addressable, or routable, and other hosts on the public Internet cannot initiate inbound connections to these devices behind the NAT. Now there are ways to get around this, there're various protocols. One is called STUN, or signaling and tunneling through UDP-enabled NAT devices. And in these types of protocols, the device sends an initial outbound packet somewhere, simply to create an entry in the NAT table and once that entry is created we now have a globally routable address and port to which devices on a public Internet can send traffic. Now these devices somehow have to learn that public IP address and port that corresponds to that service and this might be done using DNS for example. It's also possible to statically configure these tunnels or mappings on your NAT device at home. Needless to say, even with

these types of hacks and workarounds for NAT, it's clear that network address translation is a violation of the end-to-end principle because by default two hosts on the Internet, one on the home network and one on the public Internet, cannot communicate directly by default.

CS 6250: Computer Networking

Lecture 3: Switching

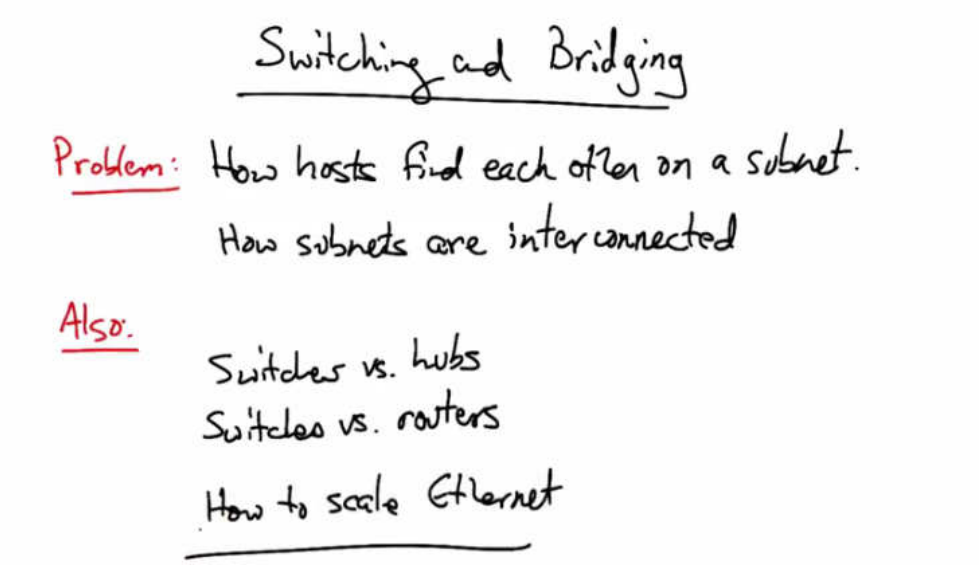
Lesson 3 Intro

At its core, a network serves to route packets between machines on the network. Let's take a look at how packets are moved across networks. It's more complicated than it sounds at first, but quite fascinating.

That's right. To even reach your screen, the packets that make up this video likely traveled across at least four or five networks, if not more.

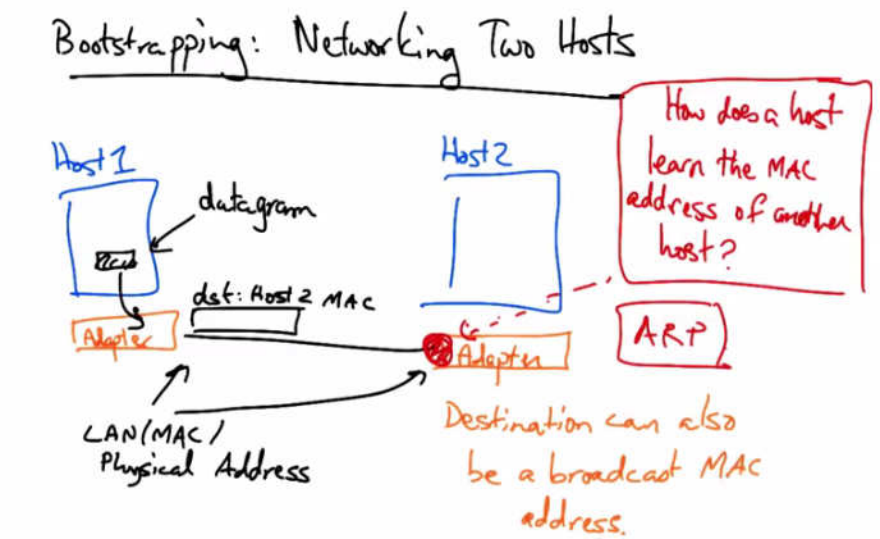
You'll learn how that works in the routing videos on BGP, a routing protocol.

Switching and Bridging



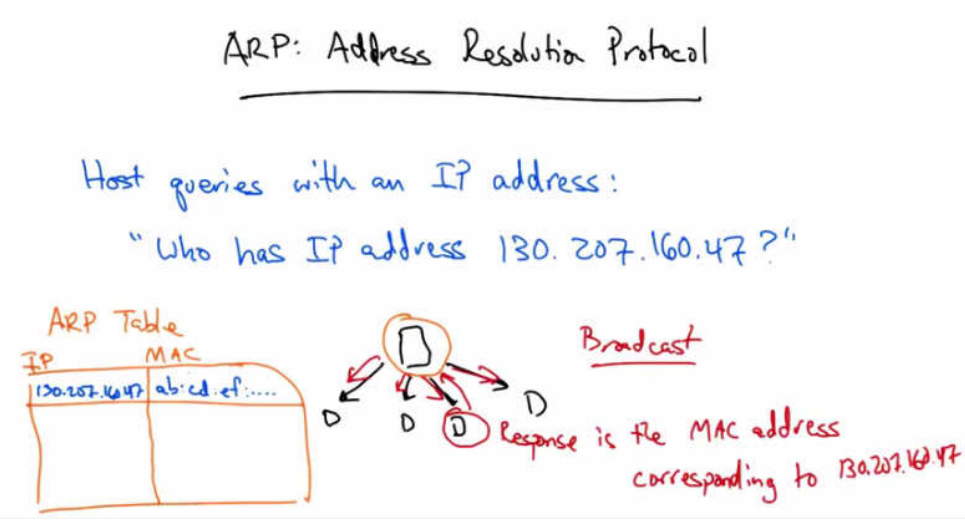
In this lesson, we will learn about switching and bridging. In particular, we will learn about how hosts find each other on a subnet and how subnets are interconnected. We will also learn about the difference between switches and hubs, and the difference between switches and routers. And we'll talk about the scaling problems with Ethernet and mechanisms that can be used to allow it to scale better.

Bootstrapping Networking Two Hosts

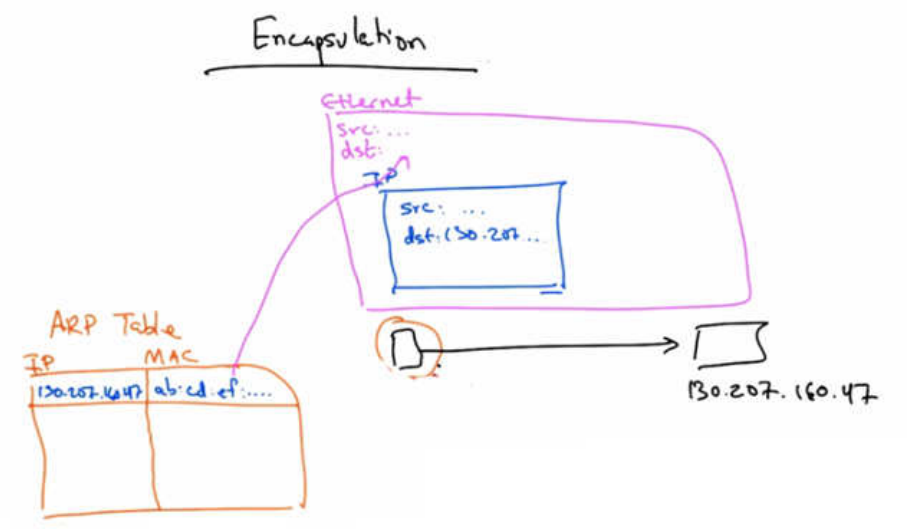


To start let's talk about how you would network two machines, each with a single interface, to each other. So Host 1 and Host 2 would be connected by two Ethernet adapters or network interfaces. And each of these would have a LAN, or physical, or MAC address. Now a host that wants to sent a datagram to another host can simply send that datagram via its Ethernet adapter with a destination MAC address of the other host that it wants to receive the frame. Frames can also be sent to a broadcast destination MAC address which would mean that the datagram would be sent to every host that it was connected to on the local area network. Now, of course, typically what happens is a host knows a DNS name or an IP address of another host, but it may not know the hardware or MAC address of the adapter on the host that it wants to send it's datagram to. So we need to provide a way for a host to learn the MAC address of another host. The solution to this is a protocol called ARP or the address resolution protocol.

ARP Address Resolution Protocol

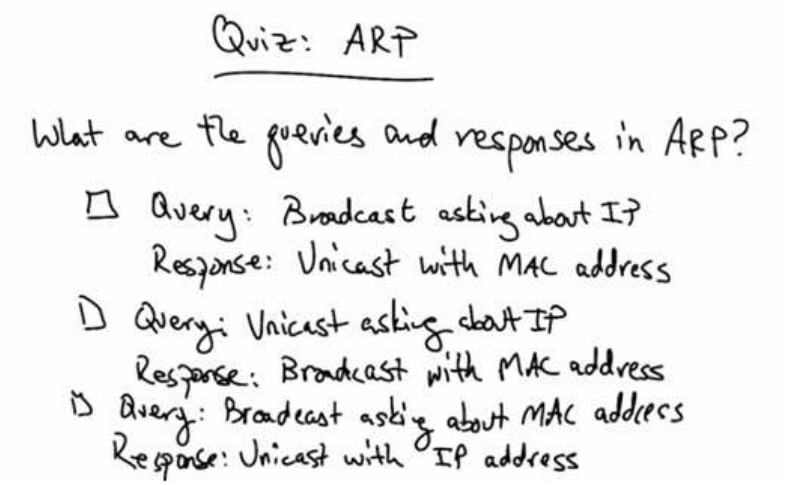


In ARP, a host queries with an IP address, broadcasting that query to every other node on the network. That query will be of a form, "who has a particular IP address," such as 130.207.160.47, and that particular host who has that IP address on the LAN will respond with the appropriate MAC address. So the ARP query is a broadcast that goes to every host on the LAN from the host that wants the answer to the query and the response is a unicast response with the MAC address as the answer. That's returned to the host that issued the query. When the host that issues the query receives a reply, it starts to build what's called an ARP table. It's ARP table then maps each IP address on the local area network to the corresponding MAC address. Now, instead of broadcasting a ARP query to discover the MAC address corresponding with this IP address, the host can simply consult its local ARP table.



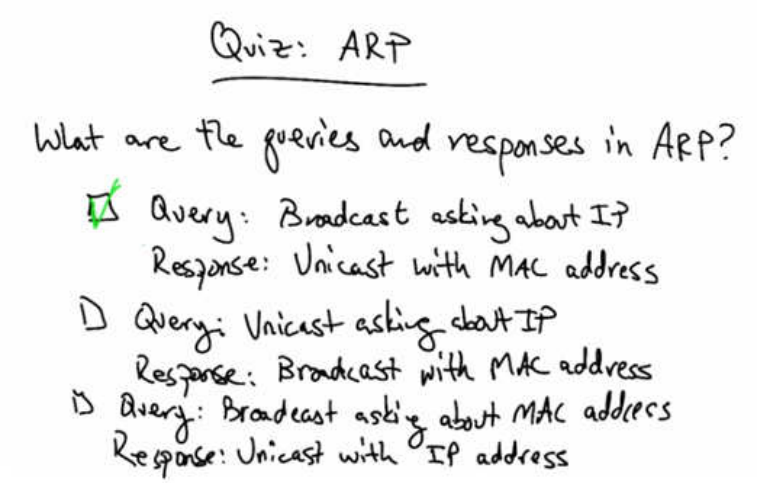
Let's now take a look at what the host does with this information. When the host wants to send a packet to the destination with a particular IP address. It takes that IP packet and encapsulates it in an Ethernet frame with the corresponding destination MAC address. Essentially, it puts that IP packet inside of an Ethernet frame. So before it sends the IP packet with that destination IP address, it first puts the packet inside a larger Ethernet frame with its own source MAC address and the destination MAC address from its local ARP table.

ARP Quiz



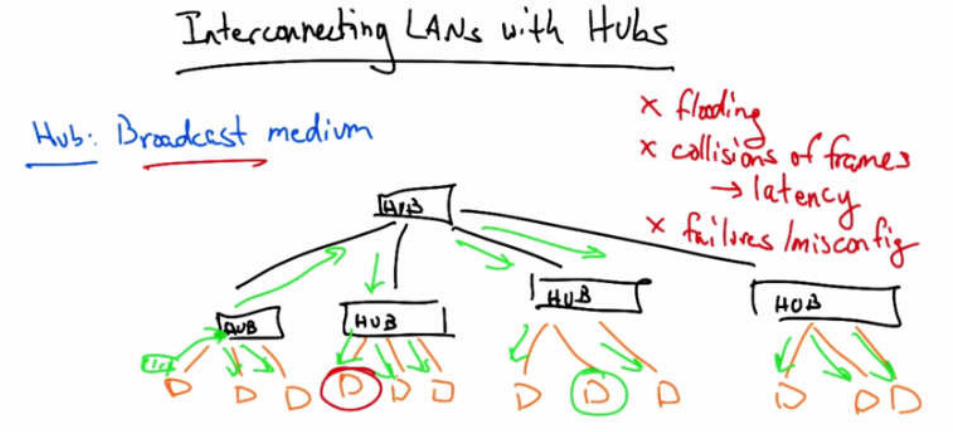
So let's consider what we learned about ARP. So what are the formats of the queries and responses in ARP? Is the query a broadcast where a host is asking about an IP address, and the response is a unicast with a MAC address? Is the query a unicast message asking about an IP address and the response is broadcast with a MAC address? Or is the query a broadcast asking about a particular MAC address, where the response is a unicast with the response of a particular IP address?

ARP Solution



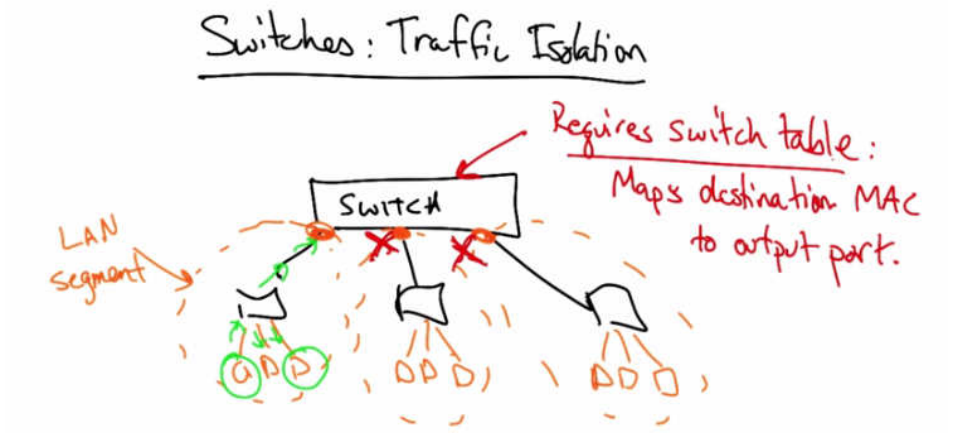
The purpose of ARP is to allow a host to discover the MAC address corresponding to a particular IP address. And the host doesn't know which host on the LAN owns that particular MAC address. So, ARP allows the host to send a broadcast query asking about who owns a particular IP address. And the response comes from the owner of that particular IP address and the response is the MAC address.

Interconnecting LANs with Hubs



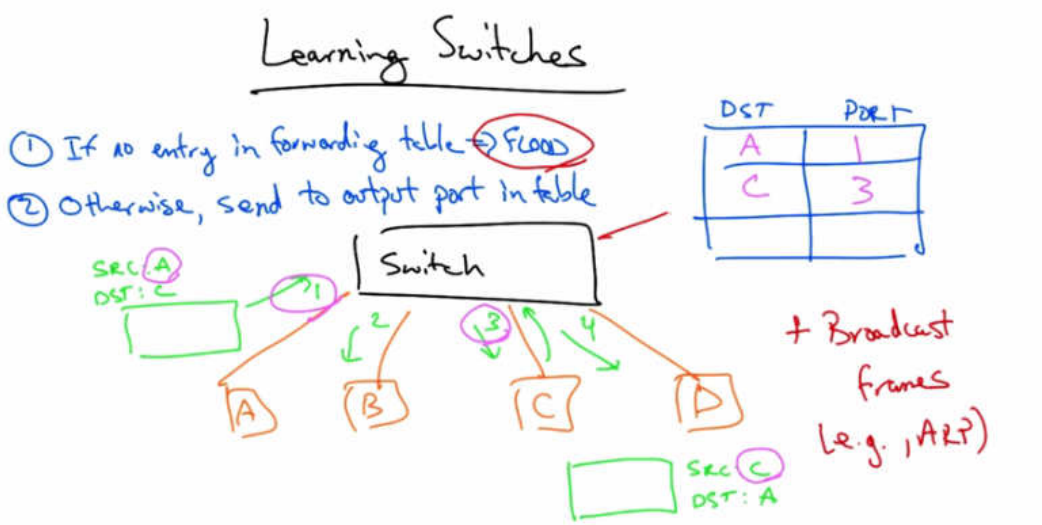
The simplest way that a LAN can be connected is with something called a hub. Hubs are the simplest form of interconnection and in some sense they don't even exist in networks anymore today, because you can build a switch for essentially the same price. But for the sake of example, let's just take a look at how a LAN would be connected with a Hub. Now, a hub essentially creates a broadcast medium among all of the connected hosts where all packets on the network are seen everywhere. So if a particular host sends a frame that's destined for some other host on the LAN, then a hub will simply broadcast that frame that it receives on an incoming port out every outgoing port. So all packets are seen everywhere. There is a lot of flooding and there are many chances for collision. The chance of collision of course, introduces additional latency in the network because collisions require other hosts or senders to back off and not send as soon as they see the other senders trying to send at the same time. LANs that are connected with hubs are also vulnerable to failures or misconfiguration because even one misconfigured device can cause problems for every other device on the LAN. Suppose that you had a misconfigured device that was sending a lot of rogue or unwanted traffic. Well, on a network that's connected with hubs, every other host on the network would see that unwanted traffic. So, we need a way to improve on this broadcast medium by imposing some amount of isolation.

Switches Traffic Isolation



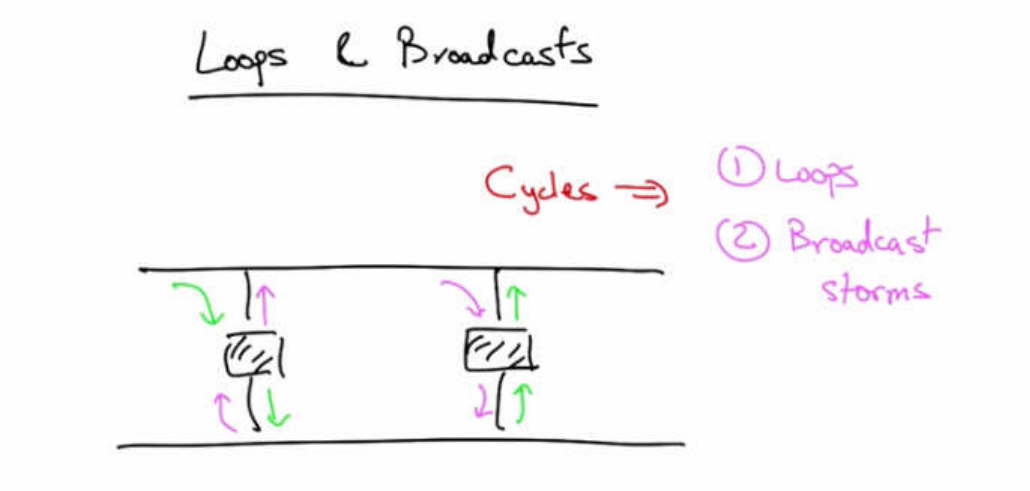
So in contrast, switches perform some amount of traffic isolation so that the entire LAN doesn't become one broadcast medium. But instead, we can partition the LAN into separate broadcast domains or collision domains. Now a switch might break the subnet into multiple LAN segments. Typically a frame that is bound for a host in the same part or segment of the LAN is not forwarded to other segments. So, for example if we had a network with three hubs, all connected by a switch, then each of these would be its own broadcast domain. And if a host here wanted to send a frame to another destination in the same segment, well that frame would be broadcast within that domain. But the switch would recognize that the destination was in the same segment and would not forward the packet on output ports destined for other LAN segments where the destination was not. Now enforcing this kind of isolation, requires constructing some kind of switch table, or state, at the switch, which maps destination MAC addresses to output ports.

Learning Switches



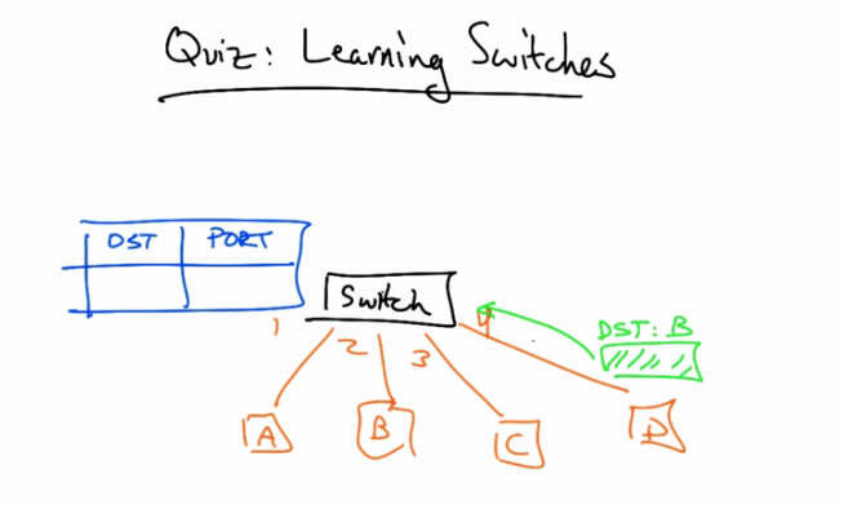
Let's take a quick look at how learning switches work. A learning switch maintains a table between destination addresses and output ports on the switch, so that when it receives a frame destined for a particular place it knows what output port to forward the frame. Initially the forwarding table is empty, so if there's no entry in the forwarding table the switch will simply flood. Let's look at a quick example. If host A sends a frame destined for host C, then initially the switch has nothing in its table to determine where that frame should be sent, so it will flood the frame on all of its outgoing ports. On the other hand, because the frame has a source address of A, and arrived on input port one, the switch can now make an association between address A and port one. In other words, it knows that the host with address A is attached to port one, so that in future, when it sees frames destined for host A, it no longer needs to flood, but can instead send the frames directly to port one. So, for example, when C replies with a frame destined for A, the switch now has an entry that tells it that it doesn't need to flood that packet. But instead, can simply send the packet directly to the output port. Note also that when C replies, the switch learns another association between address C and port three. So future frames destined for host C, no longer need to be flooded, either. They can simply be forwarded to output port three. So, in

summary, if a learning switch has no entry in the forwarding table, it must flood the frame on all outgoing ports. But otherwise, it can simply send that frame to the corresponding output port in the table. Note that learning switches do not eliminate all forms of flooding. The learning switch must still flood in cases where there is no corresponding entry in the forwarding table, and also, these switches must forward broadcast frames, such as ARP queries. Now because learning switches still sometimes need to flood, we still have to take care when the network topology has loops. Now most underlying physical topologies have loops for reasons of redundancy. If any particular link fails, you'd still like hosts on the LAN to remain connected.



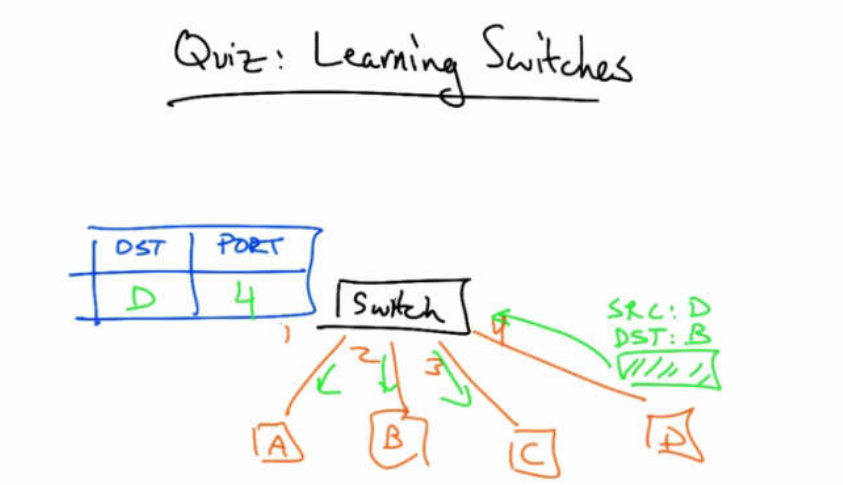
But let's see what happens when the underlying physical topology has a loop. Let's suppose a host on the upper LAN broadcasts a frame. Each learning switch will hear that frame and broadcast it on all of its outgoing ports. When that broadcast occurs, the other learning switches that are in the topology that contains a loop will hear the rebroadcast. They in turn will not know that they shouldn't rebroadcast the packet that they just heard. So each of those switches will in turn rebroadcast the packet on their outgoing ports. And, of course, this process will continue, creating both packet loops and what are known as broadcast storms. So, cycles in the underlying physical topology can create the potential for learning switches to introduce forwarding loops and broadcast storms. So we need some kind of solution to ensure that even if the underlying physical topology has cycles, which it often needs for redundancy, that the switches themselves don't always flood all packets on all outgoing ports. In other words, we need some kind of protocol to create a logical forwarding tree on top of the underlying physical topology.

Learning Switches Quiz



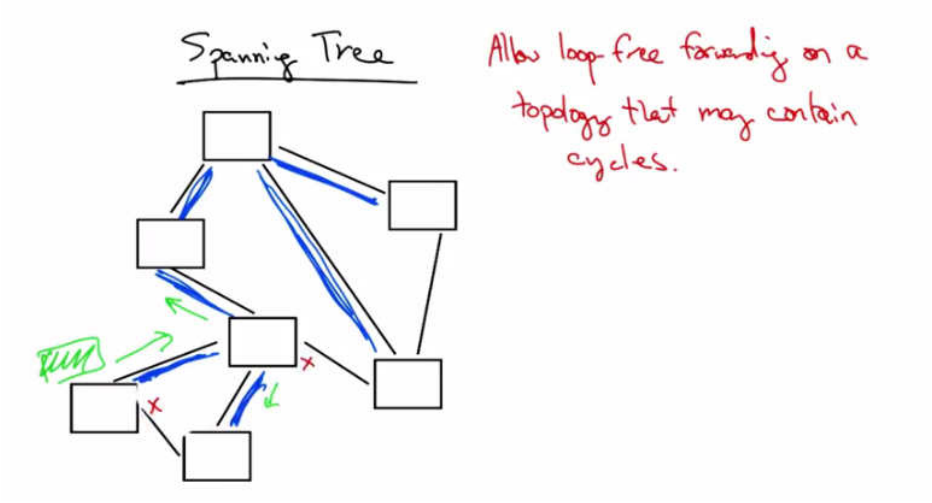
So, as a quick quiz about learning switches, let's suppose that initially the switch forwarding table is empty and host D sends a frame that is destined for host B. Fill out the entry in the switch forwarding table that is populated as a result of this message.

Learning Switches Solution

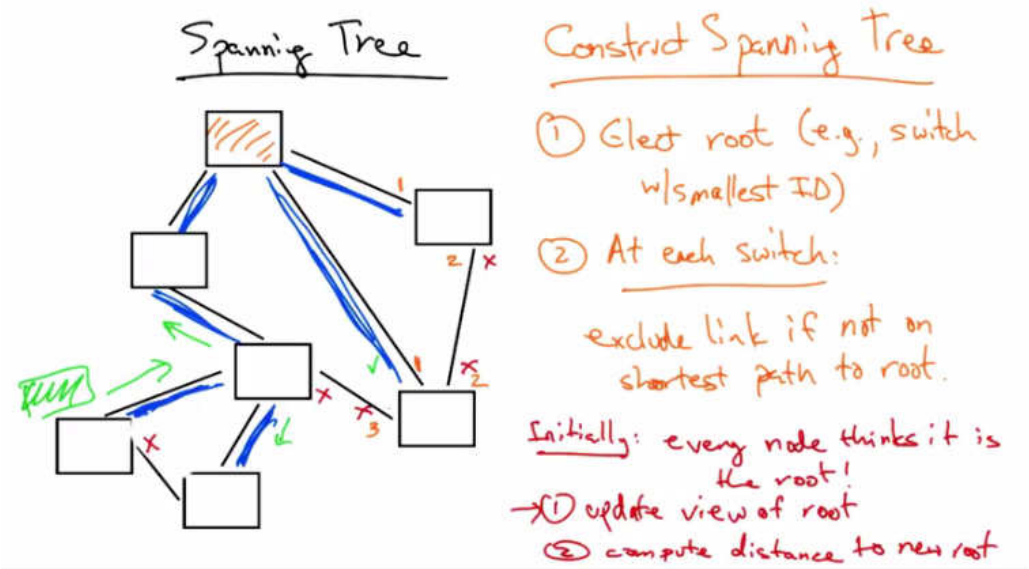


When the switch sees the frame from host D, destined for host B, it doesn't know what to do with the frame, so it forwards that frame on all of its output ports. However, because it sees a frame arrive from source D, it knows that future frames that are destined for source D should be output on port four.

Spanning Tree



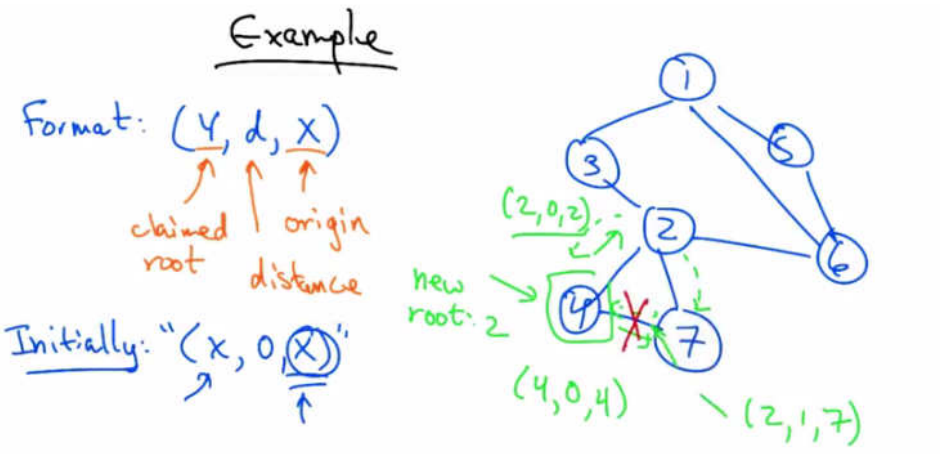
The solution to this problem is to construct what's called a Spanning Tree, which is a loop-free typology that covers every node in the graph. The set of edges, shown in blue, constitutes what's known as a Spanning Tree. The collection of edges in the blue typology covers every node in the underlying physical typology, and yet, there are no loops in the blue topology. Now, instead of flooding a frame, a switch in this topology would simply forward packets along the spanning tree. So for example, this switch would only send a frame along the port corresponding to the blue edge and would not forward the frame out any edges that were not part of the spanning tree. Other switches that receive the frame, would flood in the same fashion, along all edges that were part of the spanning tree, while omitting edges that were not members of the spanning tree.



Let's take a look at how to construct the spanning tree. First the collection of switches must elect a root, since every tree must have a root. Typically this is the switch with the smallest ID. In this case, the switch at the top of the topology is the root. Then each switch must decide which of its

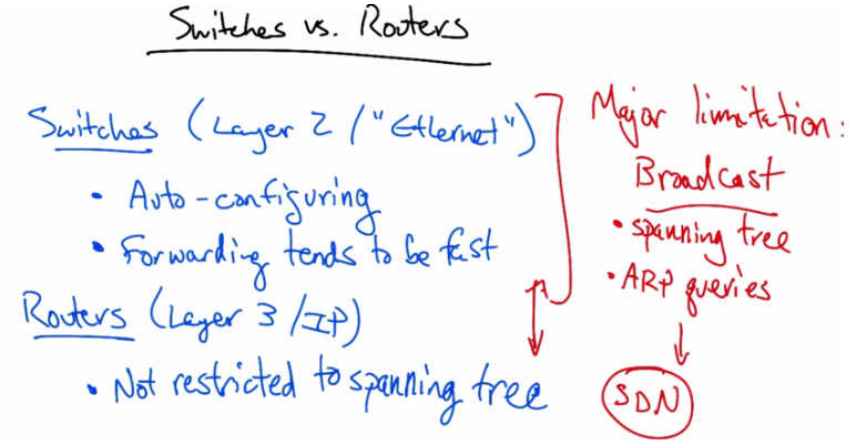
links to include in the spanning tree. And it excludes any link if that link is determined to be not on the shortest path to the root. For example, let's consider the switch in the lower right. It has three lengths, this length takes it on a path that's three hops from the root. This length takes it on a path that's two hops to the root, and this length takes it on a path, that's one hop to the root. Any link that's not on a shortest path to the route is excluded and any link that's on a shortest path on a route is included. Similarly here, this edge is on a path that's one hop away from the route and this edge is on a path that's two hops away. So this node will include this link from the spanning tree. Now, each switch repeats this process to exclude links from the underlying topology. And ultimately, this yields a forwarding topology that looks like the blue graph. And of course there is an issue which is how do we determine the root in the first place? Well initially, every node think it's the root. And the switches run an election process to determine which switch has the smallest ID. And if they learn of a switch with a smaller ID, they update their view of the root, and they compute the distance to the new root. Whenever a switch updates its view of the root, it also determines how far it is from that root. So that when other neighboring nodes receive those updates they can determine their distance to the new root simply by adding one to any message that they receive.

Spanning Tree Example



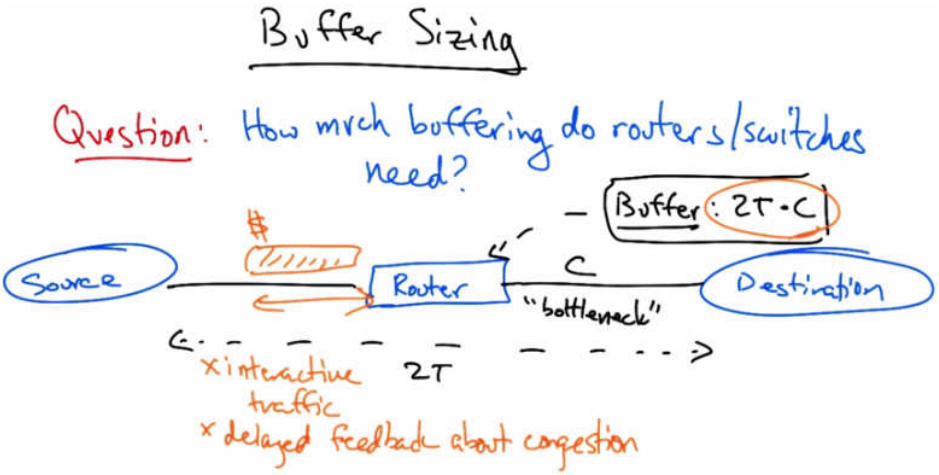
Let's take a quick example. Suppose the message format is as follows. Y, d and x, where x is the origin of the message, Y is the node being claimed as root, and d is the distance of the particular node sending this message, x, from a claimed root. So, initially every switch in the network broadcasts a message like x,0,x to indicate that the node that it thinks itself is the root. When other switches hear this message, they compare the ID of the sender to their owner ID, and they update their opinion of who the root is based on the comparison of these IDs. Let's suppose that we have the following graph and switch number 4 thinks it's the root. So we will send a message 4, 0, 4 to nodes 2 and 7. But 2 also thinks it is the root, so 4 is going to receive the message 2,0,2 from node 2, and then it's going to realize that 4 is just one hop away from node 2. So node 4 will update its view of the root to be node 2. Eventually 4 will also hear a message 2,1,7 from node 7; indicating that node 7 thinks it is one hop away from its view of the root, which is node 2. It will realize that the path through node 7 is a longer path to the root, and it will remove the link 4-7 from the tree. We can repeat this process and ultimately we will end up with a spanning tree.

Switches vs Routers



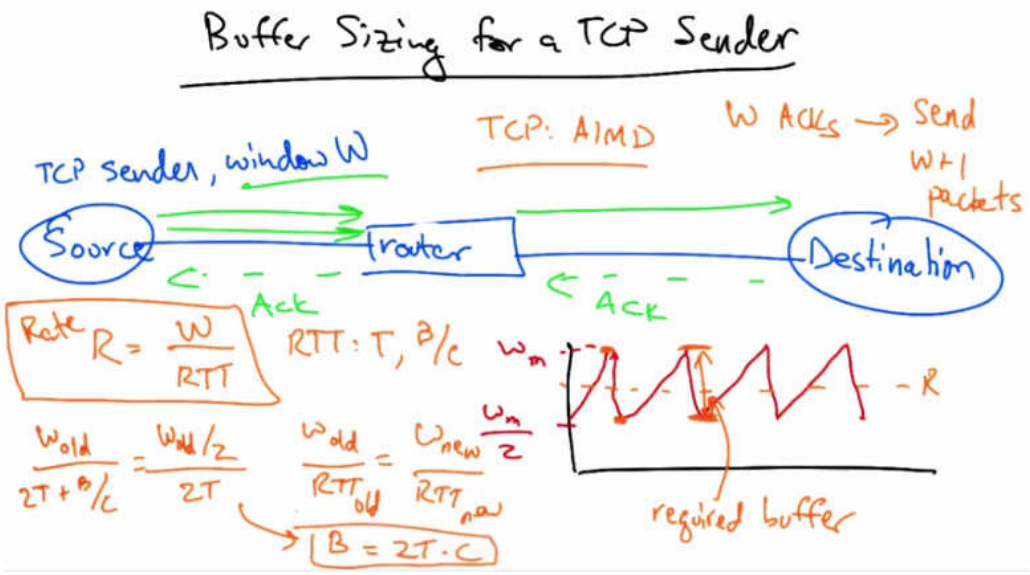
Let's do a quick comparison of switches and routers. Switches typically operate at layer two. A common protocol at layer two is Ethernet. Switches are typically automatically configuring, and forwarding tends to be quite fast since packets only need to be processed through layer two on flat look ups. Routers, on the other hand, typically operate at layer three where IP is the common protocol. And router level topologies are not restricted to a spanning tree. One can even have multipath routing, where a single packet could be sent along one of multiple possible paths in the underlying router level topology. So, in many ways Ethernet, or layer two switching, is a lot more convenient, but one of the major limitations is broadcast. The spanning tree protocol messages and ARP queries both impose a fairly high load on the network. So this raises the question of whether it's possible to get many of the benefits of the auto configuration and fast forwarding of layer two without facing these broadcast limitations. As it turns out, there are ways to strike this balance. And in the third part of the course, when we talk about network management, we will look at some ways to scale Ethernet to very large topologies. For example, in data center networks. We'll also explore how an emerging technology called Software Defined Networking, or SDN, is effectively blurring the boundary between the layer two and layer three.

Buffer Sizing



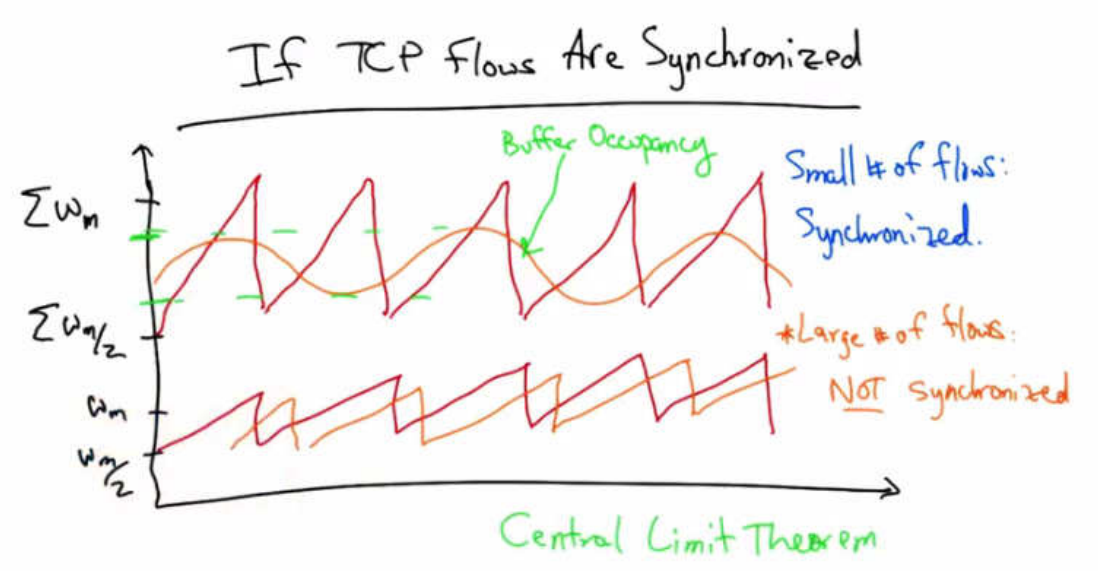
So in this lesson, we'll look at an important question in switch design which is, how much buffering do routers and switches need? It's fairly well known that routers and switches do need packet buffers to accommodate for statistical multiplexing. But it's less clear how much packet buffering is really necessary. Now given that queuing delay is really the only variable part of packet delay on the internet, you'd think we'd know the answer to this question already. And for quite some time there have been some well understood rules of thumb but it turns out that we've recently revisited this question and come up with some different answers. So let's first look at the universally applied rule of thumb. Now for the sake of the examples in this lesson, I'm going to use routers and switches interchangeably because it doesn't really matter. All that matters here is that we have a network device that's a 'store and forward' packet device that has the capability of storing a frame or a packet and then later sending it on. So let's suppose that we have a path between a source and a destination, and the round-trip propagation delay is 2T and the capacity to bottleneck link is C. Now the commonly held view is that this router needs a buffer of 2T times C. It should be clear why this rule of thumb exists. C is the capacity to the bottleneck link in say, bits per second and T is the time of units second, so this works out to bits, and the meaning of this quantity is simply the number of bits that could be outstanding along this path at any given time. It effectively represents the maximum amount of outstanding data that could be on this path between the source and destination at any time. Now this rule of thumb guideline was mandated in many backbone and edge routers for many years. It appears in RFCs and ITF Architectural guidelines and it has major consequences for router design simply because this can be a lot of router memory and memory can be expensive. The other thing of course is that the bigger these buffers, not only the bigger the cost but also the bigger the queuing delay that could exist at any given router. And hence, the more delay the interactive traffic may experience and the more delay that feedback about congestion will experience. The longer these delays are, the longer it will take for the source to hear about congestion that might exist in the network. Now to understand why this guideline is incorrect, let's first re-derive the rule of thumb a bit more formally and then we'll understand why it does not always apply in practice.

Buffer Sizing for a TCP Sender

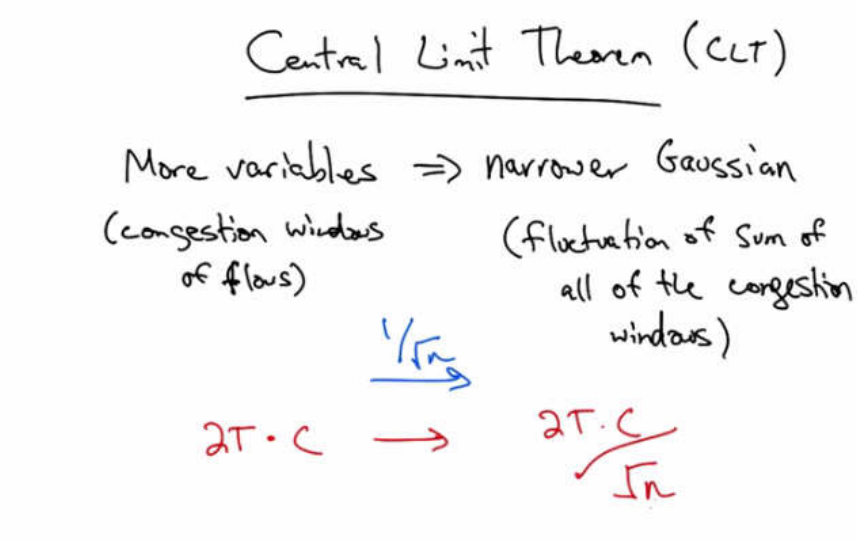


Let's suppose that we have a TCP sender that's sending packets, where the sending rate is controlled by the window W, and it's receiving ACKs (acknowledgements). Now at any time if the window is W, only W unacknowledged packets may be outstanding. So the sender's sending rate, R, is simply the TCP window, W, divided by the round trip time (RTT) of the path. So the rate is W over RTT. Now remember that TCP uses additive increase, multiplicative decrease, or AIMD, congestion control. So for every W ACKs received, we send W plus one packets, and our TCP saw tooth will look something like this. We'll start at a rate W_max over 2, increase the window to W_max and then when we see a drop we will apply multiplicative decrease and reduce the sender's sending rate to W_max over 2 again. So here, right at the point of a packet drop, this represents the maximum number of packets that can be in flight. So again, the required buffer is the maximum number of packets that can be in flight, or simply the height of this TCP saw tooth. Now we know the rate is W over RTT, and we'd like the sender to send at a common rate, R. And if we'd like the sender to be sending at the same rate before and after it experiences a loss, then we know that the rate before the drop must equal the rate after the drop. So then we can set these two rates equal. We know that the RTT is part transmission delay T, and part queuing delay which is the maximum buffer size of the bottleneck link, divided by the capacity of the bottleneck link. We also know that after reducing the window, the queuing delay is zero. So we can replace the term on the left with W_old over 2T plus B over C and we can replace the term on the right with W_old over 2, because the congestion window has been reduced half divided by 2T, simply the propagation delay with no queuing delay. Now if we solve this equation we find that the required buffering is simply 2T times C. Now the rule of thumb makes sense for a single flow, but a router in a typical backbone network has more than 20,000 flows. And it turns out that this rule of thumb only really holds if all of the those 20,000 flows are perfectly synchronized. If the flows are desynchronized, then it turns out that this router can get away, with much less buffering.

If TCP Flows are Synchronized



Now, if TCP flows are synchronized, the dynamics of the aggregate window as shown in the upper part of the graph, would have the same dynamics as any individual flow. The quantities on the Y axis here would simply be different. Specifically, the number of pockets occupying the buffer would be the sum of all of the TCP flows windows, rather than the window of any individual flow. Now if there are only a small number of flows in the network then these flows may tend to stay synchronized, and the aggregate dynamics might mimic the dynamics of any single flow, as shown. But as the network supports an increasingly large number of flows, these individual TCP flows become de-synchronized. So instead of all of the flows lining up with the saw tooth as shown in the bottom part, individual flows might see peaks at different times. As a result, instead of seeing a huge saw tooth that's the sum of a bunch of synchronized flows, the aggregate instead might look quite a bit more smooth, as a result of the individual flows being desynchronized. And we can represent this sum, which is the buffer occupancy, as a random variable. At any given time, it's going to take a particular range of values. The range of values that this buffer occupancy takes can actually be analyzed in terms of the central limit theorem.



The central limit theorem tells us that the more variables that we have, and, in this case the number of variables are the number of unique congestion windows of flows that we have, the narrower the Gaussian will be. In this case, the Gaussian is the fluctuation of the sum of all of the congestion windows. In fact, the width decreases as 1 over root N, where N is the number of unique, congestion windows of flows that we have. And therefore, instead of the required buffering, needing to be 2T times C, we can get away with much less buffering, in particular, 2T times C divided by the square root of N, where N, is the number of flows, passing through the router.

CS 6250: Computer Networking

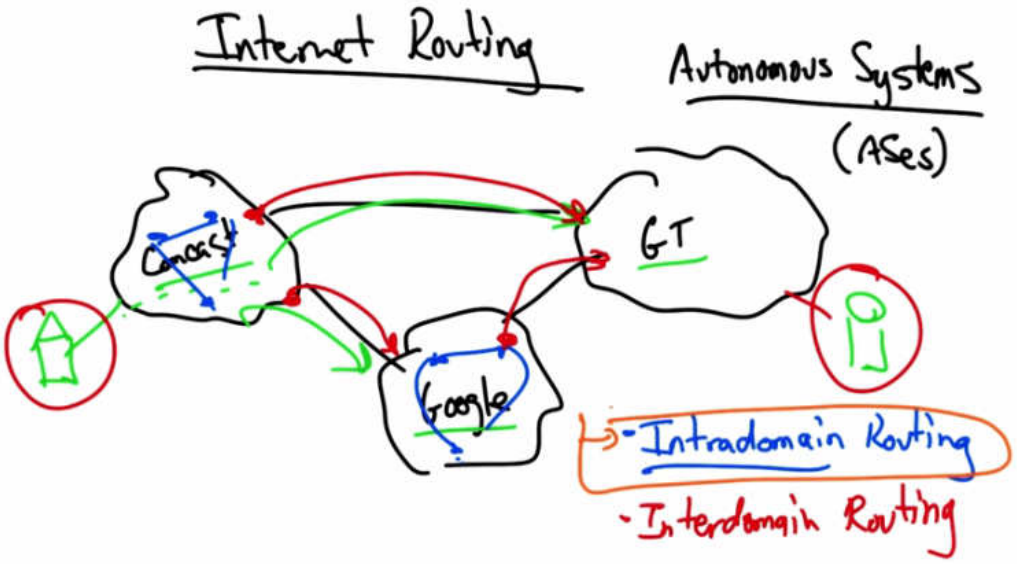
Lecture 4: Routing

Lesson 4 Intro

With your head wrapped around routing we'll now take a look at the nuts and bolts that make routing possible: naming, addressing and forwarding.

And you'll start your first significant Mininet project. In the project you'll investigate switched buffer sizing which can have an important effect on network performance.

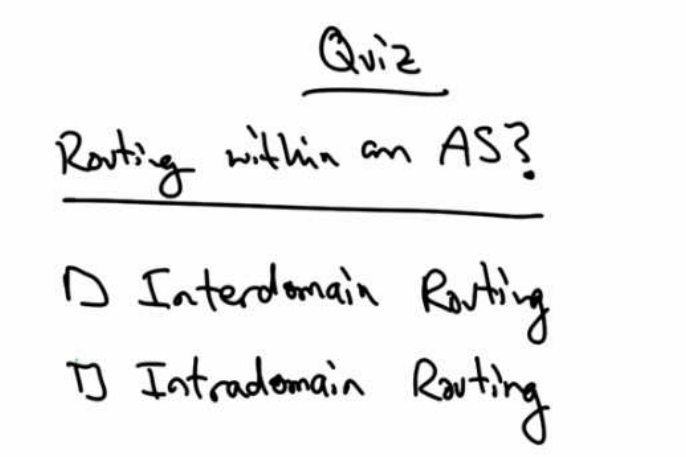
Internet Routing



The next few lessons will cover internet routing. Contrary to what you might think, the internet is not a single network, but rather a collection of tens of thousands of independently operated networks, or autonomous systems, sometimes simply called ASes. Networks such as Comcast, Georgia Tech, and Google, are different types of autonomous systems. An autonomous system might be internet service provider, a content provider, a campus network, or any other independently operated network. Now when you're sitting at home on Comcast and trying to reach content in Google or Georgia Tech, your traffic actually traverses multiple autonomous systems. This process of internet routing actually involves two distinct types of routing. One is intradomain routing, which is the process by which traffic is routed inside any single autonomous system. The other is interdomain routing, which is the process of routing traffic between autonomous systems. So computing a path between a node in an ISP like Comcast and another node in a network like Georgia Tech's involves computation of both intradomain paths

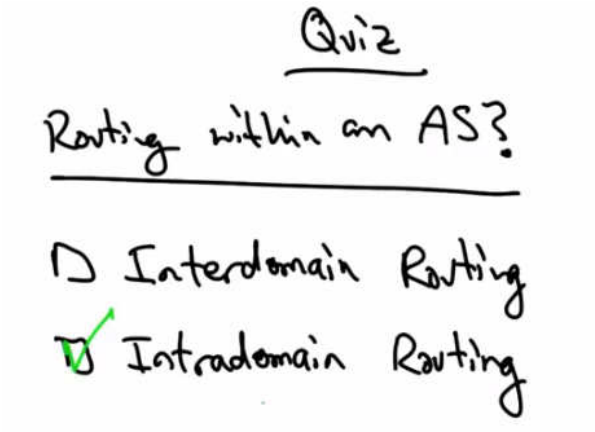
and interdomain paths. In this part of the lesson we'll look at intradomain routing. Then we'll study interdomain routing, as well as the business relationships that make interdomain routing so complicated. So let's jump into our study of intradomain routing and topology.

AS Quiz



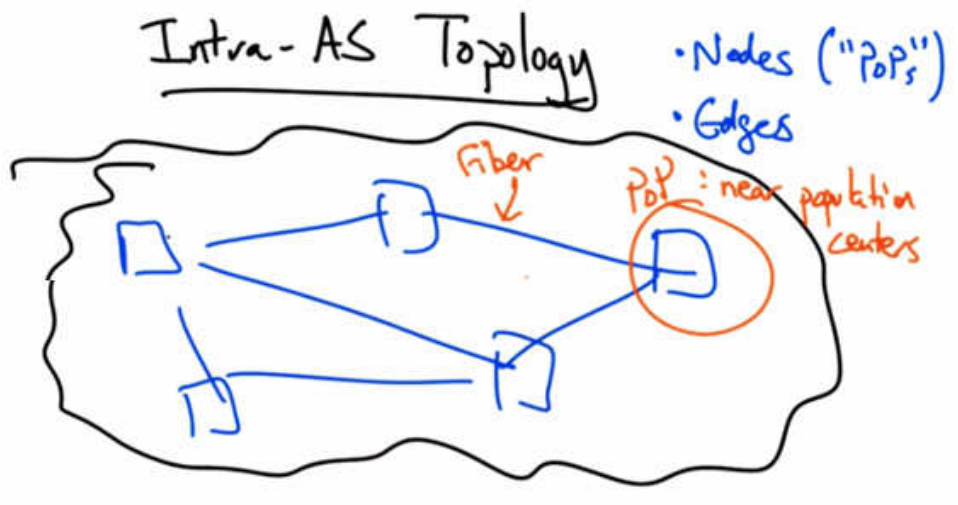
As a quick quiz, which of the following types of routing protocols are responsible for routing within an autonomous system?

AS Solution

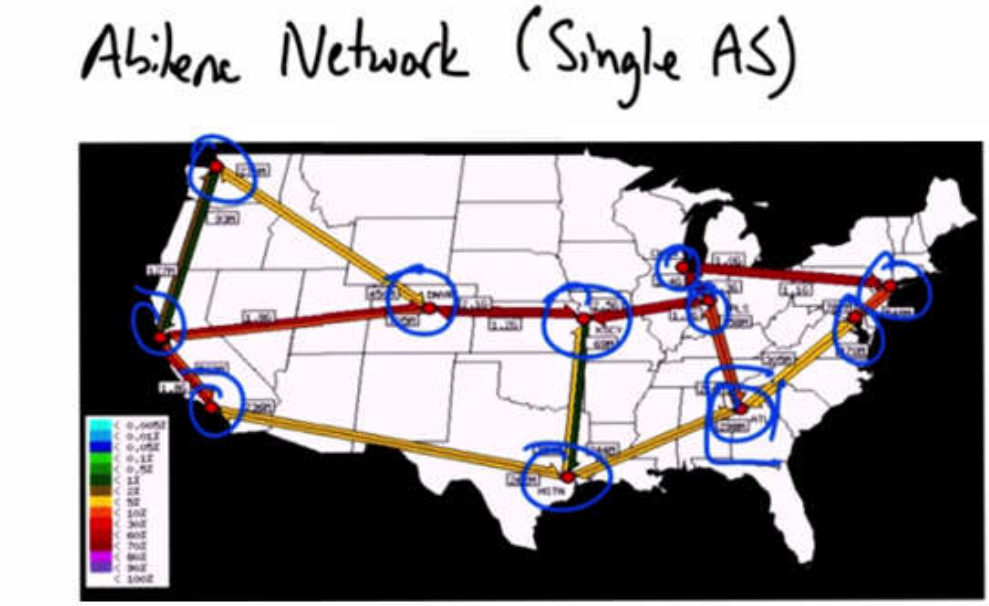


Intradomain routing protocols are responsible for routing within an autonomous system. Interdomain routing protocols, on the other hand, are responsible for routing traffic between autonomous systems.

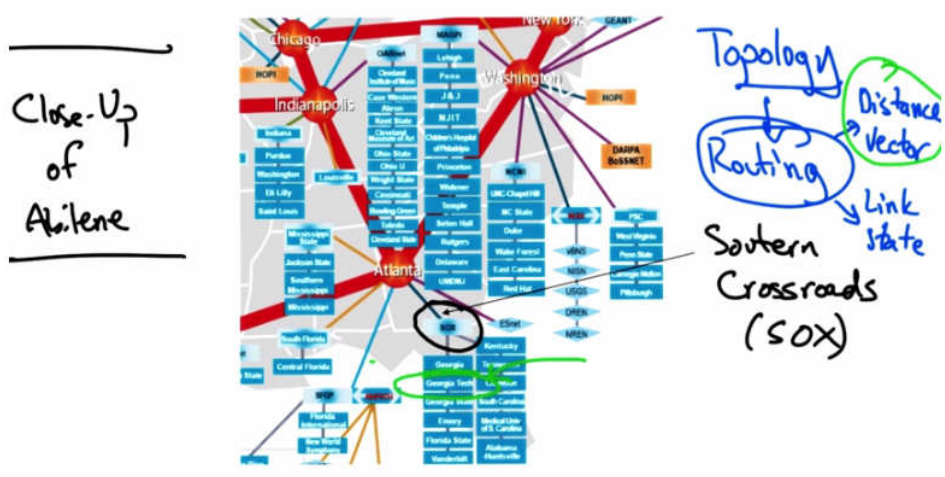
Intra AS Topology



Before we jump into intradomain routing, let's take a look at what a topology might look like inside a single autonomous system. A topology inside an AS consists of nodes and edges that connect them. The nodes are sometimes called points of presence, or PoPs. A PoP is typically located in a dense population center, so that it can be close to the PoPs of other providers for easier interconnection and also close to other customers for cheaper backhaul to customers that may be purchasing connectivity from this particular AS. The edges between pops are typically constrained by the location of fiber paths, which for the sake of convenience typically parallel major transportation routes such as railroads and highways.

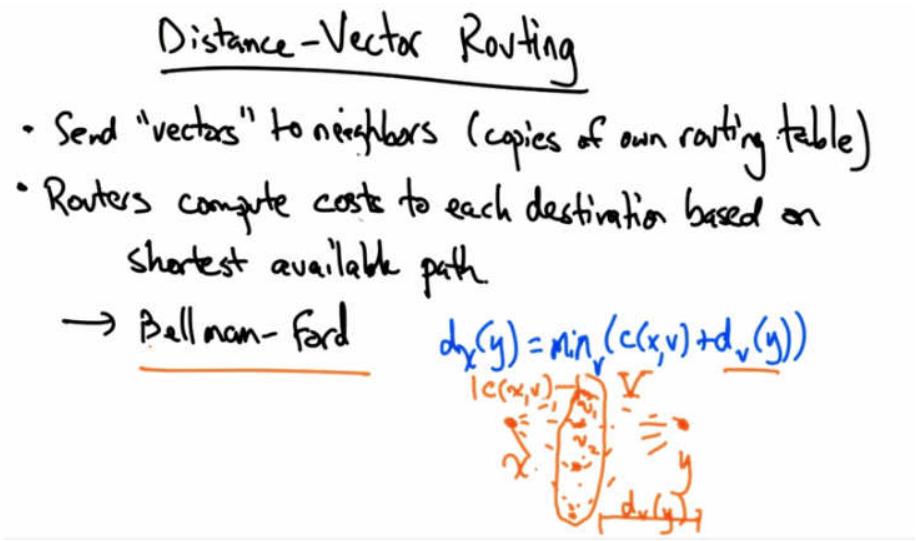


Here's an example of a single AS topology which is the Abilene Network, which is a research network in the United States. Each of these locations would be considered a PoP, and each of these PoPs may have one or more edges between them. Georgia Tech is an autonomous system that connects at the Atlanta PoP of the Abilene Network.



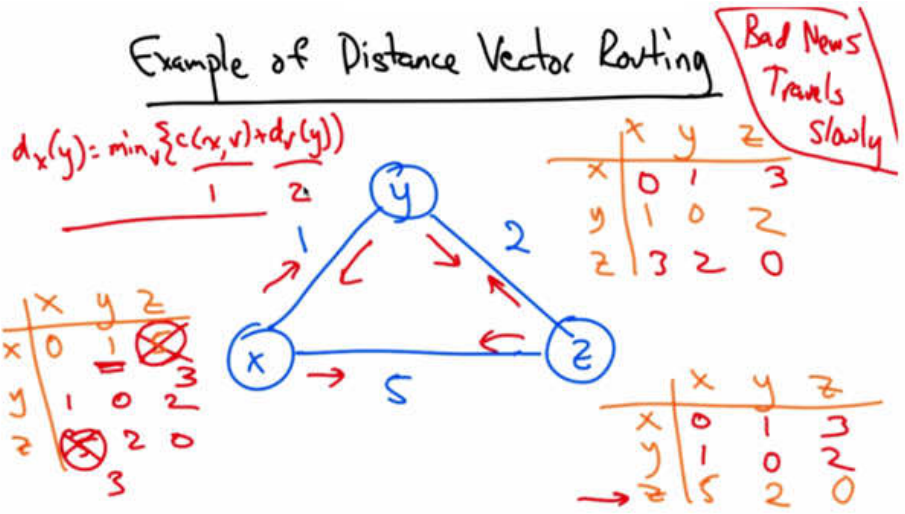
Here's a close up of the Abilene Network in the south eastern U.S. The Abilene network connects to other universities in the southeast near Atlanta and an internet exchange point called SOX, or southern crossroads. Now, thus far we've just talked about the topology of an autonomous system, which essentially defines the graph. The next step is to compute paths over that topology, a process called routing. Routing is the process by which nodes discover where to forward traffic so that it reaches a certain node. There are two types of intradomain routing. One is called distance vector, and the other is called link state. In the rest of this lesson we'll explore the two different types of intradomain routing and the advantages and disadvantages of each of them. Let's first take a look at distance vector routing.

Distance Vector Routing



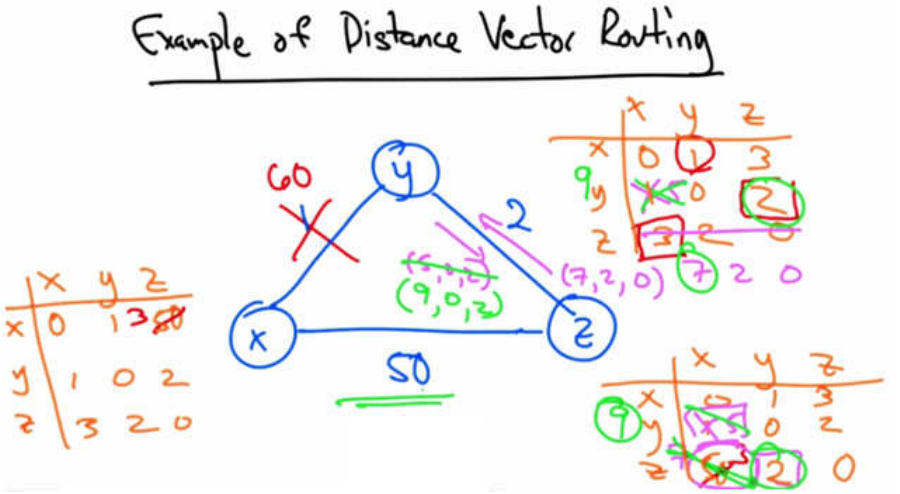
In distance vector routing, each node sends multiple distance vectors to each of its neighbors, essentially amounting to copies of its own routing table. Routers then compute costs to each destination in the topology based on shortest available path. Distance vector routing protocols are based on the Bellman-Ford algorithm. A node X's forwarding table is based on the solution to the following equation. Suppose that node X is trying to find a shortest cost route to node Y. In this case node X is trying to find a path through some intermediate node, V, that minimizes the cost between X and V, and the already known shortest cost path between V and Y. Again, the solution to this equation for all destinations, Y, in the topology is X's forwarding table. Let's now take a look at distance vector routing by way of example.

Example of Distance Vector Routing 1

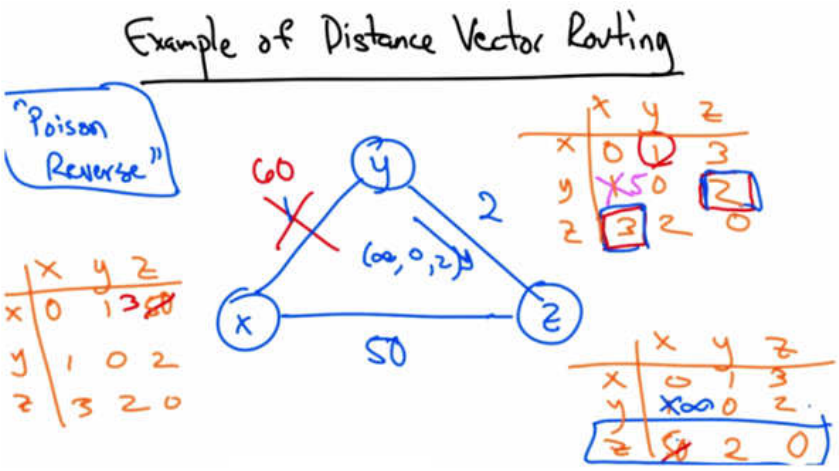


Let's suppose that we have a three node network with the costs on the edges as shown. Initially, each node has a single distance vector representing the shortest path cost to each other incident node in the graph. For example, the distance between x and x is obviously zero. And the shortest known distance between x and y from x's perspective is one, the direct path. Similarly, the shortest known distance between x and z to x at the outset is five because all it knows is the direct path. Note that a shorter path between x and z exists via y, but x simply doesn't know about it yet. Now in distance vector routing, every node send its vectors to every other adjacent node. And each node then updates its routing table according to the Bellman-Ford equation. Let's look at what happens when node x learns of y's distance vectors. Well in this case, the distance from x to z will be computed as the minimum of the sums of all distances to z through any intermediate node. So the cost between x and y is one, and the distance between y and z as discovered by y's distance vector is two. Therefore, x can update its shortest cost distance to z as three. Similarly, x will receive a distance vector from z, five two zero, but of course, when it uses the Bellman-Ford equation to update its distances, again the distance between z and x will be updated from five to three. We can repeat this exercise at other nodes, as they receive distance vectors from other nodes in the topology, and quickly, every node in the network has a complete routing table. Now when costs decrease, the network converges quickly, but one problem is that when failures occurs, bad news can actually travel slowly.

Example of distance Vector Routing 2

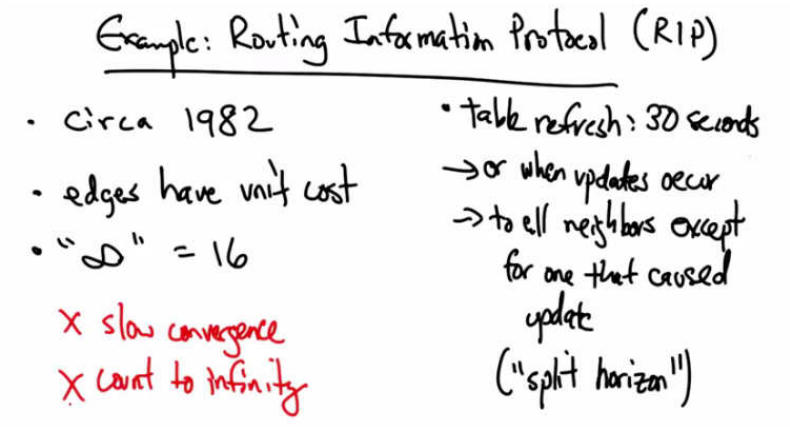


Let's look at a different example. So for the sake of illustration, I've increased the cost between x and z to 50, and now everyone starts with a different set of initial distance vectors. Now eventually, after running the distance vector protocol, we would see the tables converge as such. Let's suppose that the cost of the link between x and y suddenly increased from 1 to 60. Well now in this case, y would need to update its view of the shortest path between y and x. Now it's no longer one, but it's not 60 either. To see why let's go back to our Bellman-Ford equation. We can see that y thinks it can get to z with a cost of two, and that z can get to x with a cost of three. So in fact it's going to update this entry from one to five. Then it will tell it's neighbor z its new distance vector. In other words, that now its distance to x is no longer one but five. At this point, z needs to re-compute it's shortest path to x. Now, it knows that it can get to y with a cost of two but it thinks still that y can get to x with a cost of five. Therefore, this entry is no longer three but seven. And now z sends its new distance vector back to y. Y then updates it's distance vector for z and this process continues. So, then y thinks it is now nine units away from x. So z has to do this all over again and now z thinks that its shortest path is two plus nine or 11. Now this process repeats of course until z finally realizes that it has a shorter path of 50 directly through x after this counting up process exceeds the value of 50.



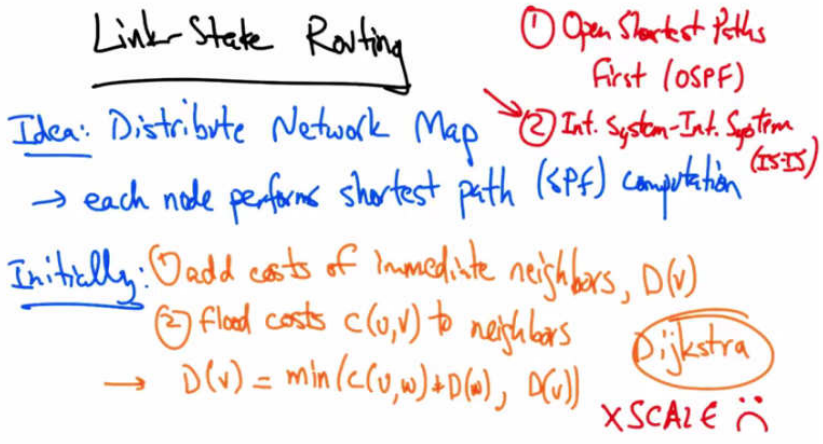
This problem is called the count to infinity problem, and the solution is called poison reverse. The idea here is that if y must route through z to get to x in its table, as it did here, then y advertizes an infinite cost for the destination x to z. So instead of sending five, zero, two, y would send infinity, zero, two. This would thus prevent z from routing back through y, and immediately, it would choose the shortest path to x, of path cost 50.

Routing Information Protocol



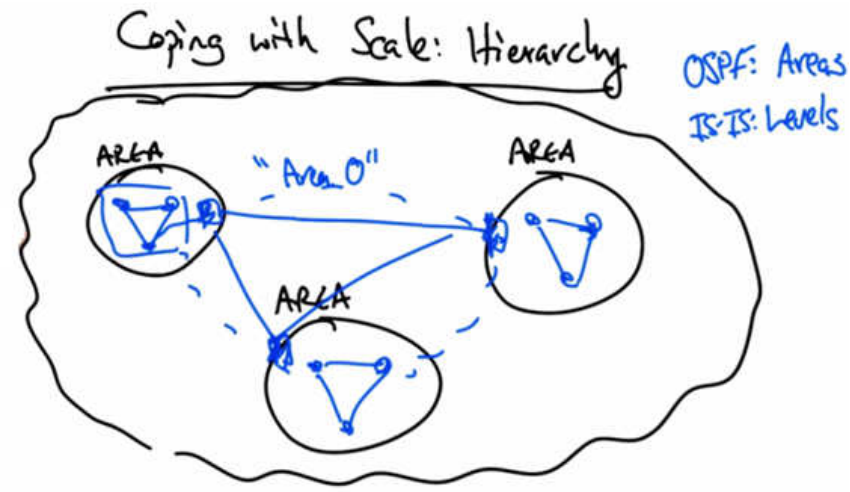
An example of a distance vector routing protocol is the routing information protocol or RIP. The first version of RIP was defined in 1982 where edges had unit cost, and infinity for the count to infinity problem was 16. Table refreshes occur every 30 seconds and when an entry changes, it sends a copy of that update to all of its neighbors except for the one that induced the update. This rule is sometimes called the split horizon rule. The small value for infinity ensures that the count to infinity doesn't take very long and every round has a time out limit of 180 seconds which is basically reached when a router hasn't received an update from a next hop for six 30 second periods. In practice, when a router or link fails in RIP, things can often take minutes to stabilize. So because of problems such as slow convergence and count to infinity, protocol designers look to other alternatives.

Link State Routing



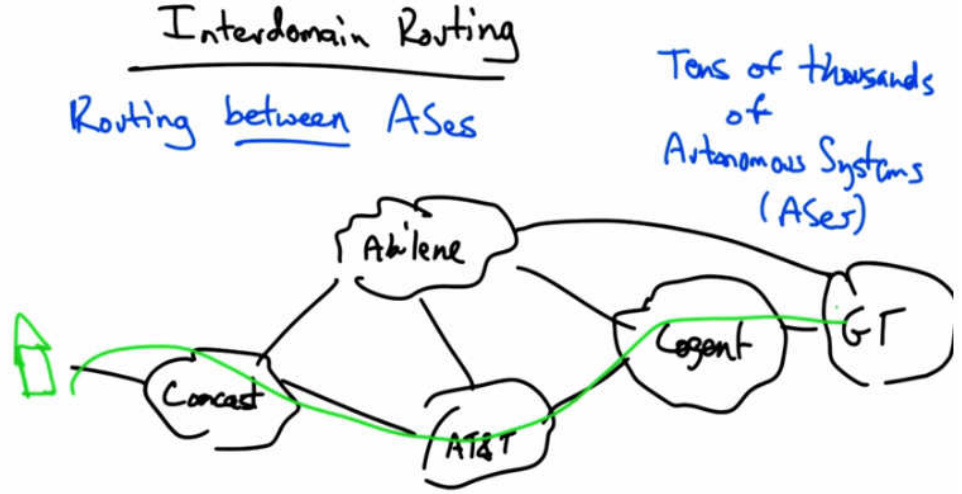
The prevailing alternative and the one that is used in most operational networks today is link state routing. In link state routing, each node distributes a network map to every other node in the network and then each node performs a shortest path computation between itself and all other nodes in the network. So, initially each node adds the cost of its immediate neighbors, D(v), and every other distance to a node that is infinite. Then each node floods the cost between nodes u and v to all of its neighbors. And the distance to any node v becomes the minimum of the cost between u and w plus the cost to w, or the current shortest path to v. The shortest path computation is often called the Dijkstra shortest path routing algorithm. Two common link state routing protocol are open shortest paths first or OSPF and intermediate system- intermediate system or IS-IS. In recent years, IS-IS has gained increasing use in large internet service providers and is the more commonly used link state routing protocol in large transit networks today. One problem with link state routing is scale. The complexity of a link state routing protocol grows as n cubed where n is the number of nodes in the network.

Coping with Scale Hierarchy

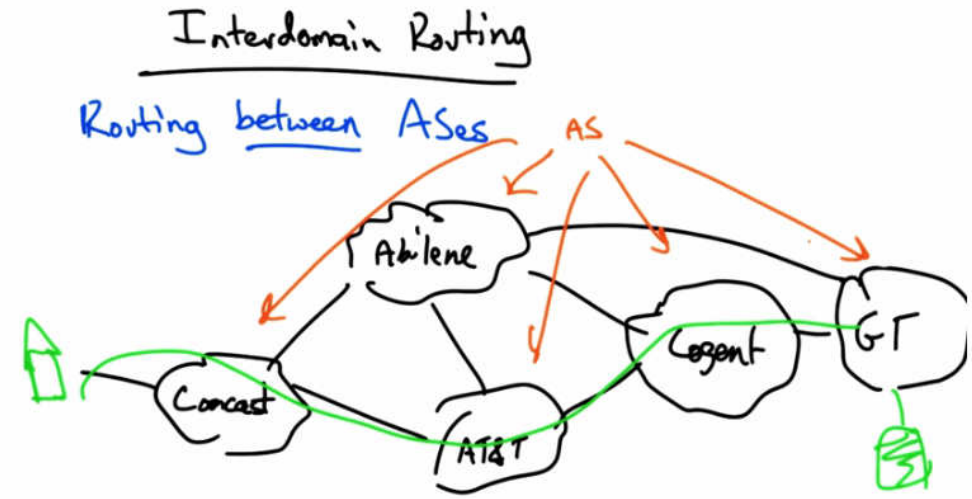


One way of coping with scale is to introduce hierarchy. OSPF has a notion of areas, and IS-IS has an analogous notion of levels. In a backbone network, the network's routers may be divided into levels, or areas, and the backbone itself may have its own area. In OSPF, the backbone area is called area zero, and each area in the backbone that's not in area zero has an area zero router. The area zero routers perform shortest path computations and the routers in each of the other areas independently perform shortest path computations. Now paths are computed by computing the shortest path within an area, or, if the path must leave an area, it's computed by stitching together the shortest path to the area zero backbone router, and then the shortest path across area zero followed by another intra-area shortest path.

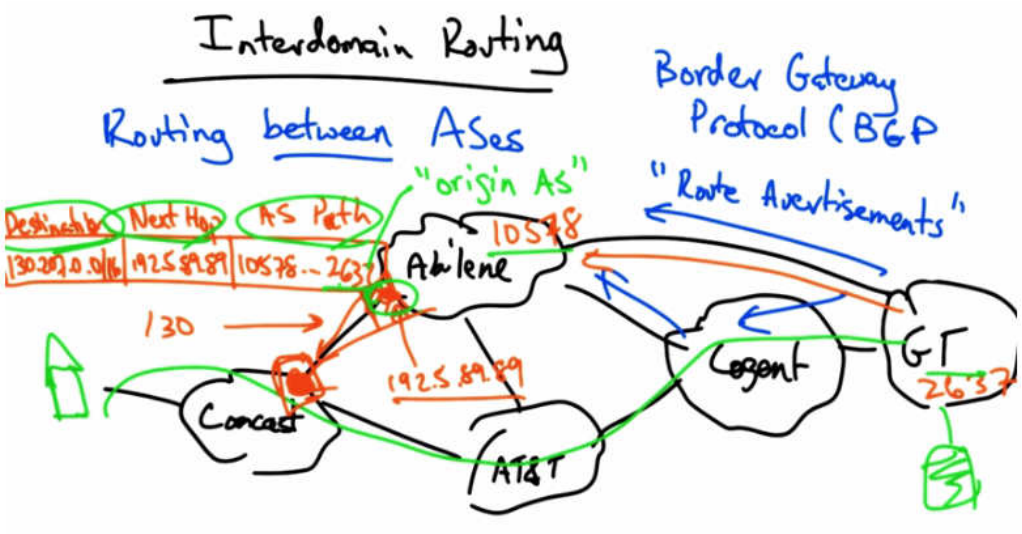
Interdomain Routing



We're now moving on to cover interdomain routing or routing between ASes. Recall that internet routing consists of routing between tens of thousands of independently operated networks, or autonomous systems. Each of these networks operates in their own self-interest and have independent economic and performance objectives, and yet they must cooperate to provide global connectivity so that when you're sitting at home, you can retrieve content that might be hosted at the Georgia Tech network.

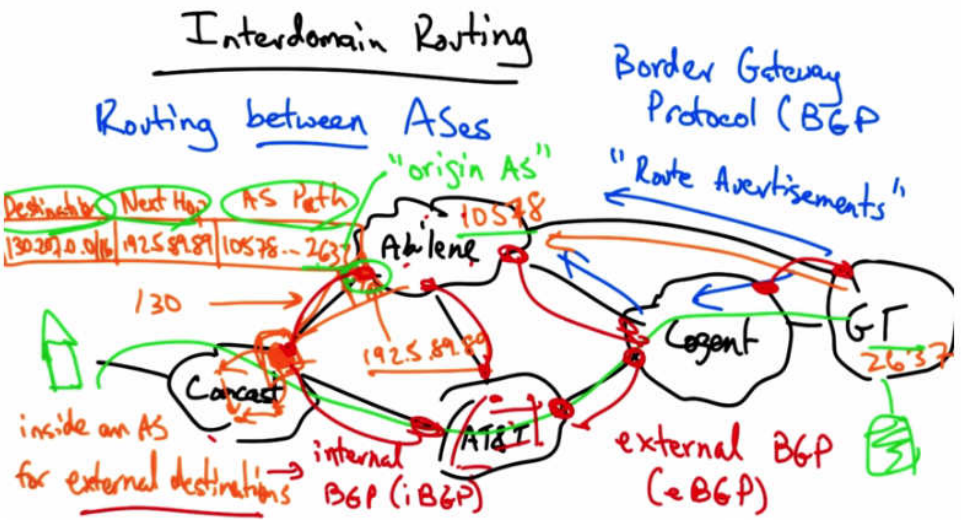


Now, each independently operated network is called an autonomous system, or AS. And each AS advertises reachability to some destination by sending what are called route advertisements or announcements.



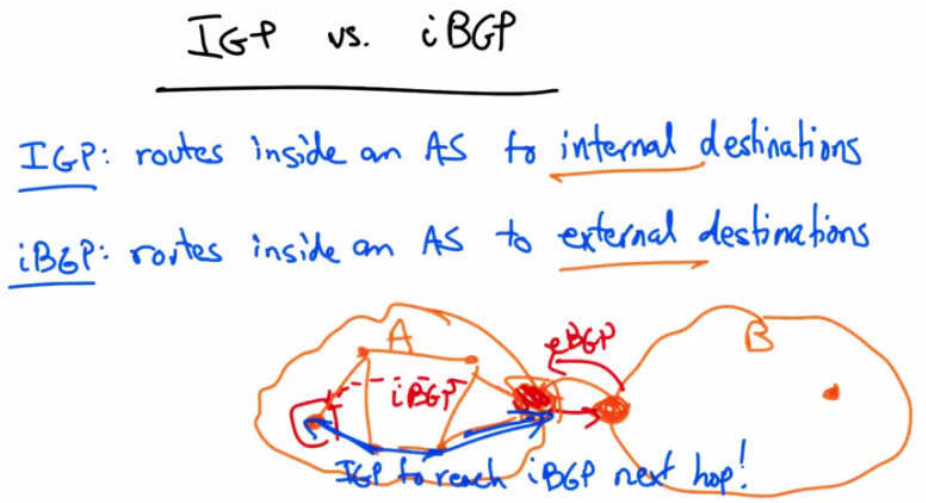
The protocol that ASes use to exchange these route advertisements is called the Border Gateway Protocol, or simply, BGP. A route advertisement has many important attributes, but for now, let's just talk about three. Now a router here, let's say on the Comcast network, might receive a route advertisement, typically from its neighboring AS. That route advertisement might contain a destination prefix, such as the IP prefix for Georgia Tech. Then it might contain what's called a next hop IP address, which is the IP address of the router that the Comcast router must send traffic to, to send traffic along that route. Typically that next hop IP address is the IP address for the first router in the neighboring network. And the Comcast router knows how to reach that next hop IP address because its border router and the border router in the neighboring AS are on the same subnet. Typically this might be a /30 subnet, therefore this IP address is reachable from Comcast's border. A third important attribute is what's called the AS path, which is a sequence of what are called AS numbers that describe the route to the destination. Now strictly speaking, the AS path is nothing more than the sequence of ASes that the route traversed to reach the recipient AS. So for example, Georgia Tech's AS number is 2637 and Abilene's is 10578 so the AS path that Comcast would hear if it received a route advertisement from Abilene for Georgia Tech, would be 10578 followed by 2637. So in the remainder of the lesson we'll look at other BGP route attributes. But these are essentially the three most important because they describe how to stitch together an interdomain path to a global destination. So we have the destination IP prefix for the destination that a router might want to send traffic to; the next hop, which is the IP address for the router for the next hop along the path; and finally, the AS path, which is the sequence of ASes that the route traversed en route to the AS that's hearing the announcement. The last AS number on the AS path is often called the origin AS, because that is the AS that originated the advertisement for this IP prefix. In this case, the origin AS is 2637, or Georgia Tech, because it is the AS that originated the announcement for this prefix.

Interdomain Routing 2



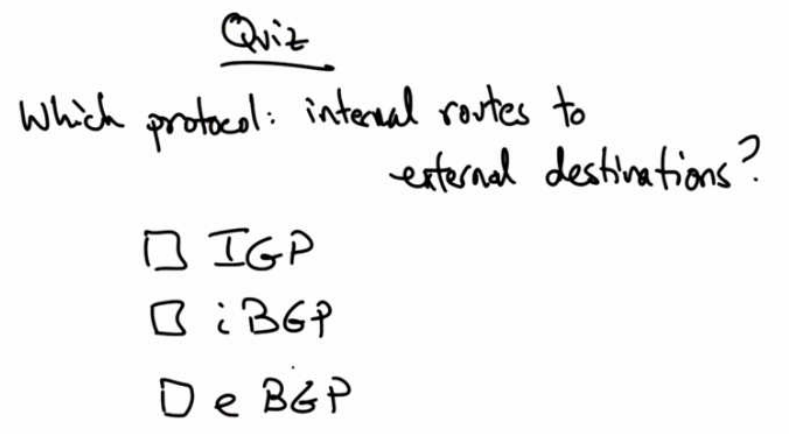
Now thus far, we've talked about interdomain routing BGP, or the border gateway protocol, as consisting of route advertisements solely between border routers of adjacent autonomous systems. In fact, this is a specific type of BGP called external BGP, or eBGP. But in fact, as we know, each one of these autonomous systems has routers of its own, inside. Those routers also need to learn routes to external destinations. The protocol that is used to transmit routes inside an autonomous system for external destinations, is called internal BGP or iBGP. Okay, so to review, external BGP is responsible for transmitting routing information between border routers of adjacent ASes about external destinations. And internal BGP is responsible for disseminating BGP route advertisements about external destinations to routers inside any particular AS. Note the distinction between iBGP and an intra-domain routing protocol or an IGP.

IGP vs iBGP



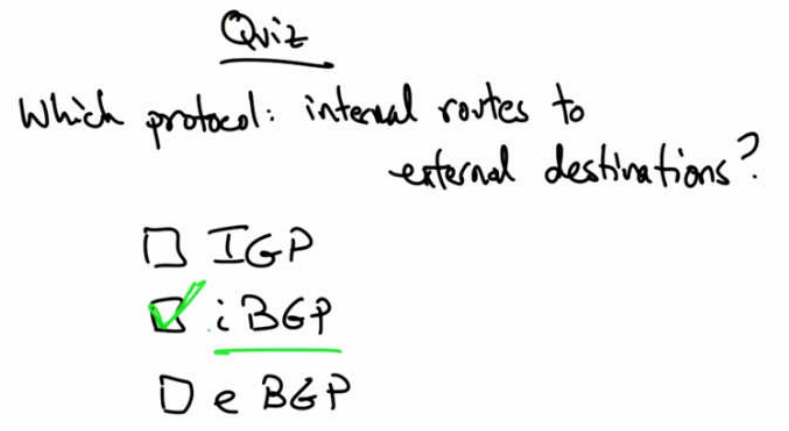
The IGP or the intra-domain routing protocol, disseminates routes inside an AS to internal destinations whereas iBGP or internal- border gateway protocol, disseminates routes inside an AS to external destinations. So let's suppose that a router inside AS A is trying to reach a destination inside AS B. AS A would learn the route via eBGP and the next hop of course, at this router, would be the border router at B. And now a router inside autonomous system A would learn the route to B via iBGP. Now the BGP next stop, would be the border router. And so, this router inside AS A, needs to use the IGP, to reach the iBGP next hop.

Protocol Quiz



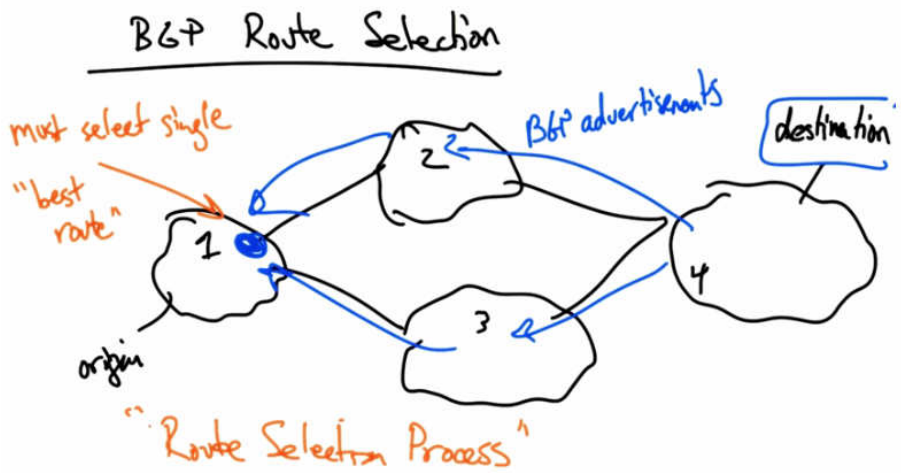
So as a quick quiz, which routing protocol is responsible for disseminating routes inside an AS to external destinations? Is it the IGP? Is it iBGP. Or is it eBGP?

Protocol Solution



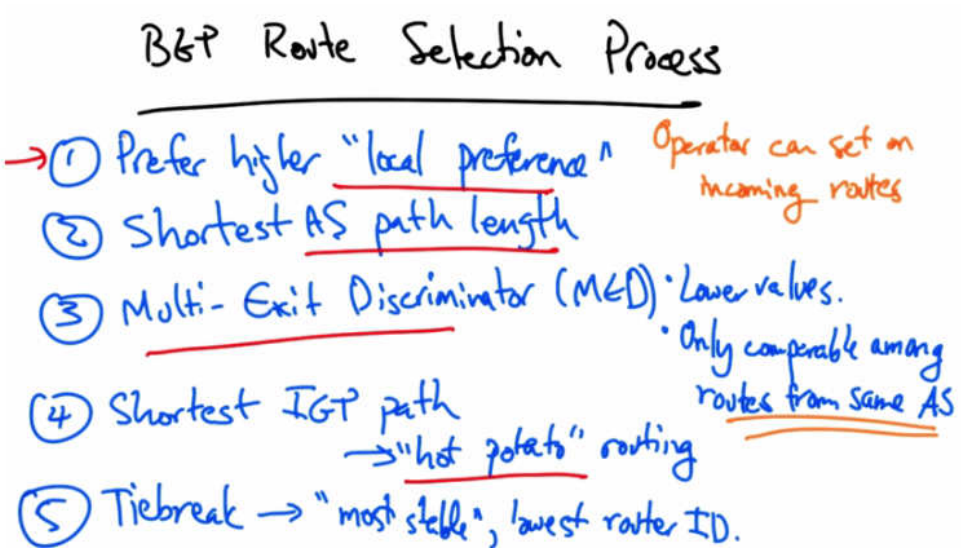
iBGP is responsible for disseminating routes inside an AS about destination IP prefixes that are located outside that AS. The iBGP next hop is typically a next hop IP address that is reachable via the ASes intradomain routing protocol, or IGP.

BGP Route Selection



Let's now take a quick look at BGP route selection. It is often the case that a router on a particular autonomous system might learn multiple routes to the same destination. In this case, a router on autonomous system one, might learn a route to a destination in AS4 via both AS2 and AS3. In this situation. The router in AS one must select a single best route to the destination among the choices. The selection among multiple alternatives is known as the BGP route selection process. Let's now take a quick look at that process.

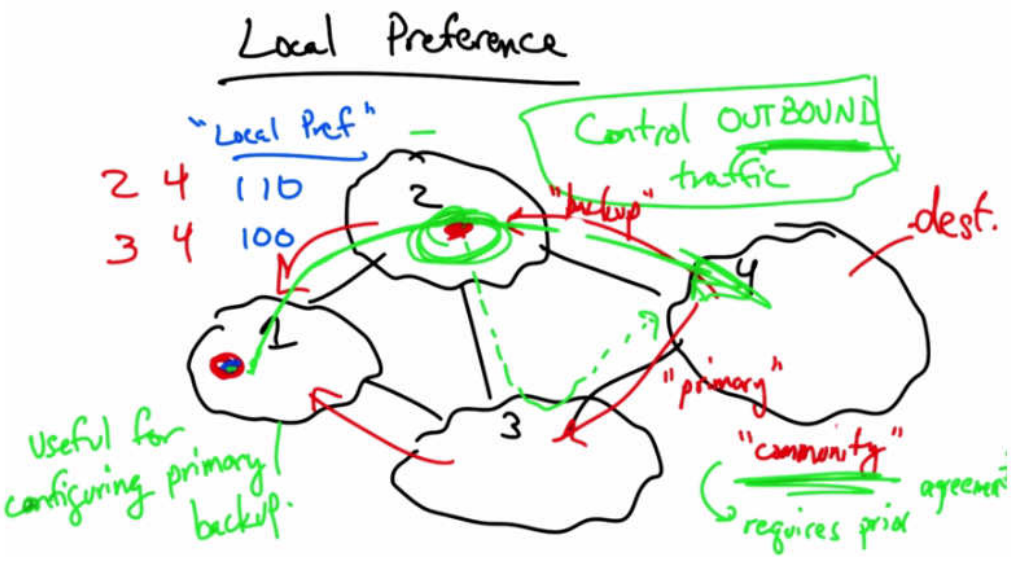
BGP Route Selection Process



The first step in the BGP route selection process is to prefer a route with the higher local preference value. The local preference value is simply a numerical value that a network operator in the local AS can assign to a particular route. This attribute is purely local. It does not get transmitted between autonomous systems, so it is dropped in eBGP route advertisements. But it allows a local network operator the ability to explicitly state that one route should be preferred

over the other. Among routes with equally high local preference values, BGP prefers routes with shorter AS path length. The idea is that a path might be better if it traverses a fewer number of autonomous systems. The third step involves comparison of multiple routes advertised from the same autonomous system. The multi-exit discriminator (MED) value allows one AS to specify that one exit point in the network is more preferred than another. So lower MED values are preferred, but this step only applies to compare routes that are advertised from the same autonomous system. Because the neighboring AS sets the MED value on routes that it advertises to a neighbor, MED values are not inherently comparable across routes advertised from different ASes. Therefore this step only applies to routes advertised from the same AS. Fourth, BGP speaking routers inside an autonomous system will prefer a BGP route with a shorter IGP path cost to the IGP next up. The idea here is that if a router inside an autonomous system learns two routes via iBGP then it wants to prefer the one that results in the shortest path to the exit of the network. This behavior results in what is called "hot potato" routing, where an autonomous system sends traffic to the neighboring autonomous system via a path that traverses as little of its own network as possible. Finally, if there are multiple routes with the highest possible local preference, the shortest AS path and the shortest IGP path, the router uses a tiebreak to pick a single breaking route. This tiebreaking step is arbitrary. It might be the most stable, or the route that's been advertised the longest. But often, to induce determinism, operators typically prefer that this tie breaking step is performed based on the route advertisement from the router with the lowest router ID, which is typically the neighboring router's IP address. Let's now take a closer look into local preference, AS path length, multi-exit discriminator, and hot potato routing. Now as I mentioned, the first step in the router selection process is for routers to prefer routes with higher local preference values. Now an operator can actually set the local preference value on incoming BGP route advertisements to affect which route a router ultimately selects. Let's see how this works.

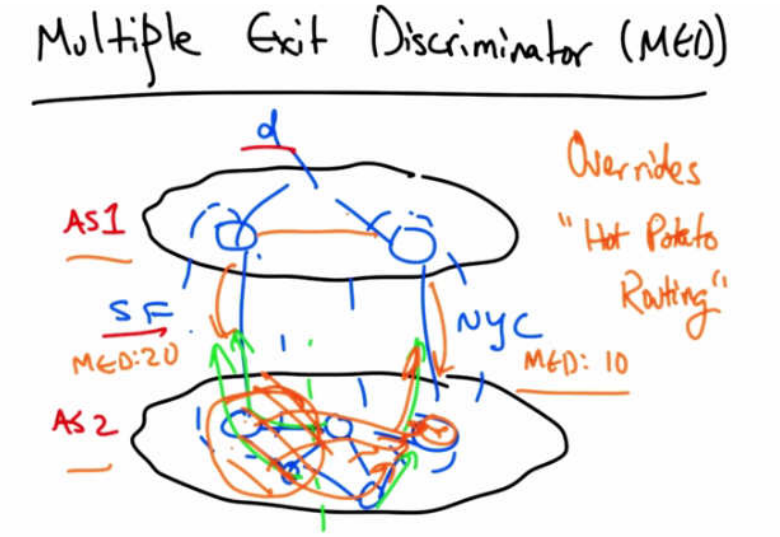
Local Preference



Now, a router in AS1 might learn two routes to a destination, one via the AS path 2-4 and the other via the AS path 3-4. Local preference, or simply, local pref, allows an operator to configure

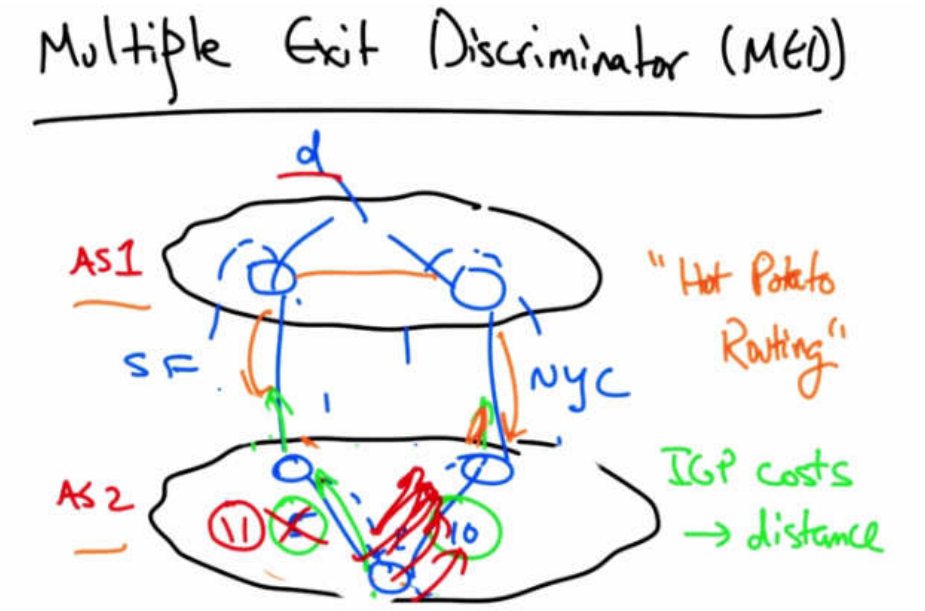
the router to assign different preference values to each of the routes that it learns. The default local preference value is 100. But if the operator prefers that this router select the path through AS two, it can configure the router to set a higher local preference for that route such as 110. This results in this router selecting the route through AS two and sending traffic to the destination in AS four via AS two. In this way an operator can adjust local preference values on incoming routes to control outbound traffic or to control how traffic leaves its autonomous system en route to a destination. This is extremely useful in configuring primary and back up routes. For example, here the route though AS two might be the primary route ,and the route through AS three, is the backup route. Now typically, as I mentioned, local preference is used to control outbound traffic. But sometimes autonomous systems can attach what's called a BGP community to a route to affect how a neighboring autonomous system sets local preference. A community is nothing more but a fancy jargon word for a tag on a route. So let's suppose that AS four wanted to control inbound traffic by affecting how AS two or AS three set local preference. In this case, let's suppose that AS two wanted traffic to arrive via AS three, its primary, rather than by AS two, its backup. In this case, AS two might advertise its BGP routes with primary and backup communities. The backup community value might cause a router in AS two to adjust its local preference value, thus affecting how AS two's outbound traffic choices are made. So, again local preference is used to control outbound traffic, in this case AS two's outbound traffic decision. But the use of a BGP community on the route advertisement can sometimes be used to cause a neighboring AS to make different choices regarding it's outbound traffic, thereby, allowing an AS to specify a primary or back up path for incoming traffic. This type of arrangement requires prior agreement.

Multiple Exit Discriminator



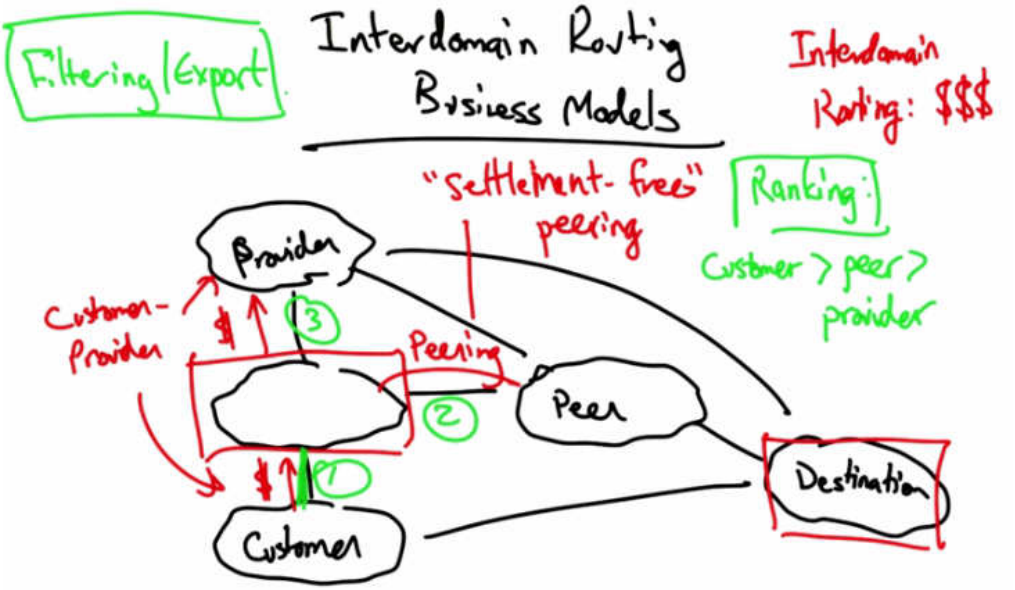
Let's suppose that two autonomous systems connect in two different cities, San Francisco and New York. Let's further suppose that AS 1 wants traffic to destination d to enter via New York City, rather than via the peering link in San Francisco. Well, remember that all things being equal, routers inside AS 2 will select the BGP route with the shortest IGP path cost to the next hop, resulting in hot potato routing. So some routers will select the San Francisco egress, and

other routers might select the New York egress. To override this default hot potato routing behavior, AS1 might advertise its BGP routes to AS2 with MED values. For example, if the MED value on the route learned at the border router in New York was 10, and the MED value from the route learned from the router in San Francisco was 20, then instead of performing hot potato routing, all of these routers that would ordinarily be closer to the San Francisco egress, would instead pick the route learned via the New York egress because the preference for a lower MED value comes before the preference for a next hop with the lower IGP path process. So all of these routes would instead be carried over AS 2's backbone network and exit via New York. Thus MED overrides hot potato routing behavior allowing an AS to explicitly specify that it wants another neighboring AS to carry the traffic on its own backbone network, rather than dumping the traffic at the closest egress and forcing traffic across the neighbor's backbone. MEDs are typically not used in conventional business relationships, but they're sometimes used, for example, if AS 1 does not want AS2 free riding on AS 1's backbone network. So effectively MED allows AS 1 to say, yes, I will connect or peer with you, but it is your job to carry the traffic long distances across the country. This mechanism is sometimes used when a transit provider peers with a content provider, and the transit provider doesn't want the content provider essentially getting free transit through the neighboring AS.

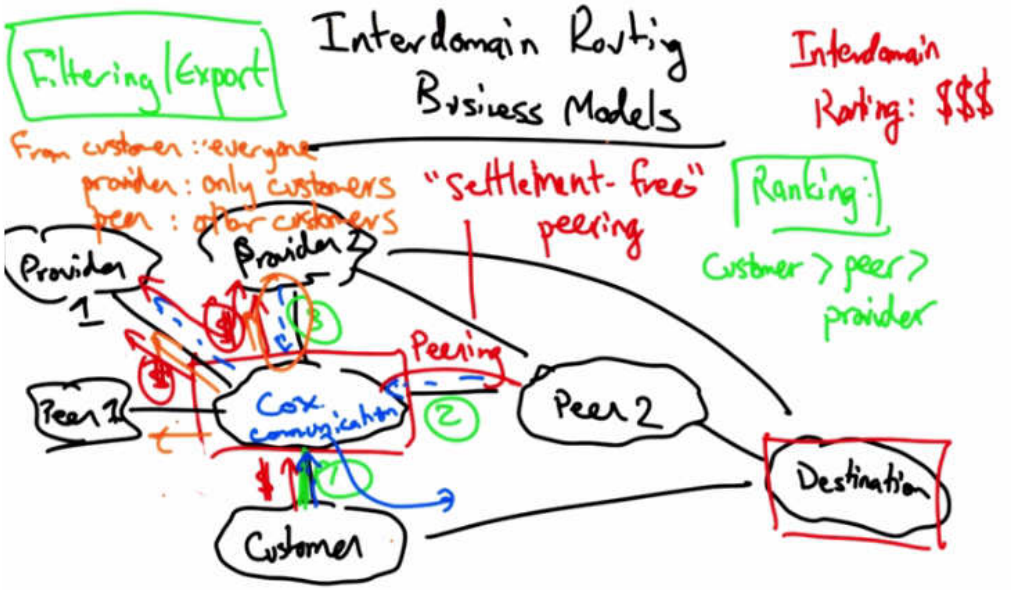


In the absence of MED overriding any behavior, typically what will happen is a router inside AS 2 would learn multiple routes via internal BGP to different egress points for the same destination d, and it would simply pick the next hop, or the egress router with the lowest IGP path cost, in this case, 5. It's very common practice to set these IGP costs in accordance with distance, or propagation delay, thus resulting in routers inside the AS picking shorter paths. Now one problem with this notion of hot potato routing is that a very small change in IGP path cost can result in a lot of BGP routing changes. Remember that it's probably not just one destination that's being routed through the San Francisco egress, but maybe tens of thousands of routes. So a single IGP path cost change can result in rerouting of tens of thousands of IP prefixes in BGP. People have looked at various ways to improve the stability of BGP routing by decoupling the IGP and the BGP in this part of the route selection process.

Interdomain Routing Business Models

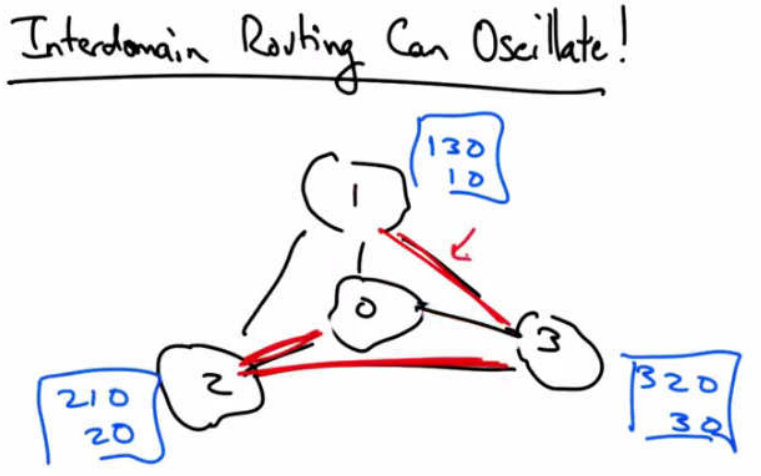


So now we're going to look at Interdomain Routing Business Models. So the one thing to remember about interdomain routing is that it's really all about routing money. Let's consider this AS that wants to send traffic to a particular destination. Well, in the internet there are two different types of business relationships: a customer-provider business relationship, where money flows from customer to provider regardless of the direction that traffic flows; the other type of business relationship is a peering relationship where an AS can exchange traffic with another AS free of charge. This is sometimes also called settlement-free peering. So already you can see given three possible ways to reach the destination. This AS is first going to prefer a route through its customer, because regardless of the direction of traffic on this link, money is always flowing from the customer. The peering link is the second most preferable because it's free. And the least preferable route is through the provider, because the AS has to pay money every time it sends traffic on this link. This leads to the basic rules of preference in interdomain routing, where customer routes are preferred over peer routes, which are in turn preferred over provider routes.

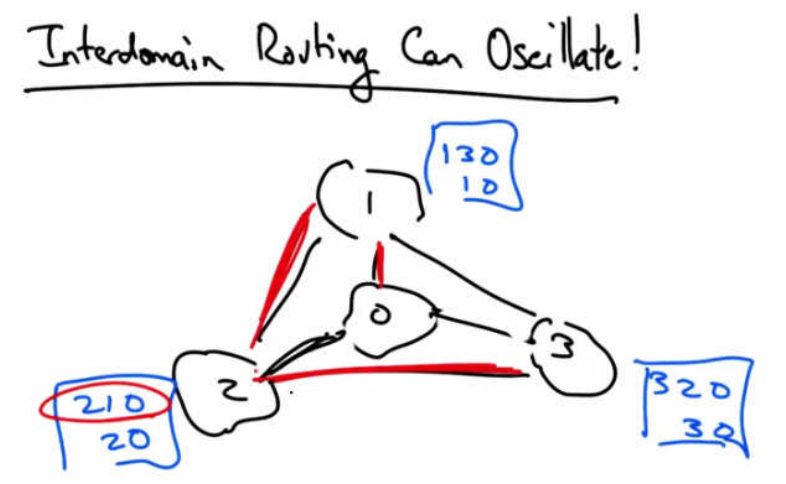


The other consideration that an AS has to make is filtering, or export decisions. In other words, given that an AS learns a route from its neighbor, to whom should it re-advertise that route? To understand filtering and export decisions, let's add a couple more AS's to the graph. Let's add another peer, and let's add another provider. Let's call this AS in the middle of the picture Cox Communications. This ISP might have smaller regional customers and it might also buy transit connectivity from other providers. Now let's suppose that this AS learns routes to a destination via its customer, its peer, and its provider. Now we already have established that it would prefer the customer route, so that it can make money by sending traffic to that destination. But what about filtering decisions? Well, routes that are learned from a customer, Cox of course would want to re-advertise to everyone else, because the more people use that route, the more money Cox makes. Therefore a route that's learned from a customer, gets advertised to everybody else. On the other hand, a route that's learned from a provider, if it were actually selected, would of course, only be advertised to customers. It wouldn't make any sense to take a route like this and advertise it to another provider. The reason, of course, is that money is flowing in the direction of the providers. So any route that's learned from a provider would never be advertised to another provider, because it would result in Cox essentially becoming a transit provider between two of its own providers and paying them both for the privilege of carrying that traffic. So routes learned from a provider would only ever be advertised to other customers. And similarly, routes from peers would only be advertised to other customers, not to other peers or other providers. So to summarize, interdomain routing has both ranking rules, where, given multiple choices, an AS might prefer a customer route over a peer route over a provider route. And then, given that it selected a particular route from either a customer, a provider, or a peer, it makes different decisions about where to re-advertise that route to other neighboring ASes. Now as it turns out, if every AS in the internet followed these rules exactly, then routing stability is guaranteed. Now you might wonder, isn't routing stability guaranteed already? And it turns out that it isn't.

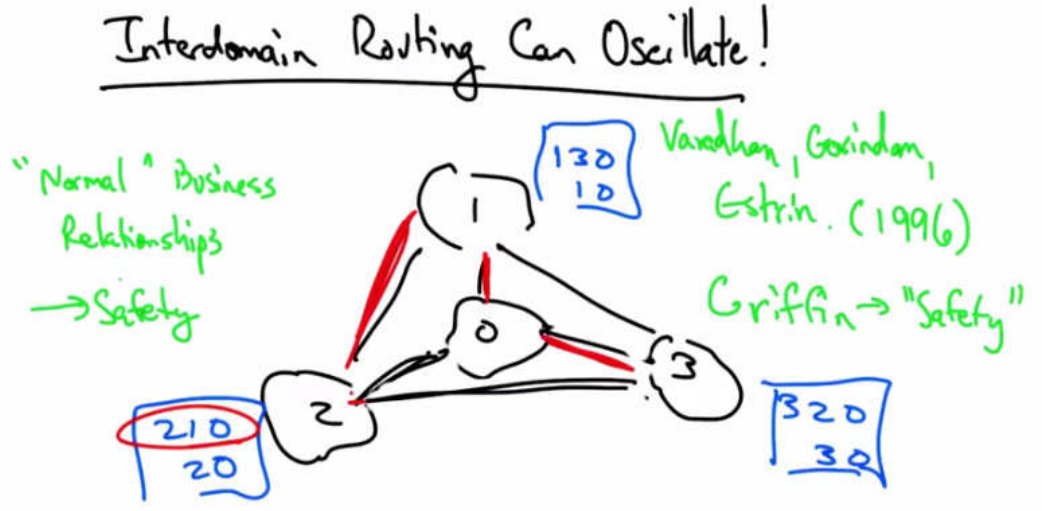
Interdomain Routing Can Oscillate!



In fact, interdomain routing can oscillate indefinitely. To see why, consider the following 4 AS topology, where each AS specifies preferred paths, presumably via local preference. So each AS prefers the AS in the clockwise direction, rather than the shorter, direct path. Now it's pretty easy to see that there's no stable solution. Let's suppose that we started off with everybody selecting the direct path. Well, in this case, any one of these ASes would notice that it has a more preferred path. So for example, AS 1 would see that because AS 3 has picked the direct path, then, in fact, it could prefer a situation where oscillations can occur indefinitely. Similarly, here now AS 3 sees that it has a more preferred path, 3 2 0, so it might switch to that.



In doing so, it breaks AS 1's path. 1 3 0 no longer works. So AS1 has to switch back to its less preferred direct path, but now we're in the same situation all over again because now AS2's preferred path becomes available via 1, so AS 2 now reroutes, and AS 3's most preferred path, 3 2 0, no longer works so it must switch to the direct path.

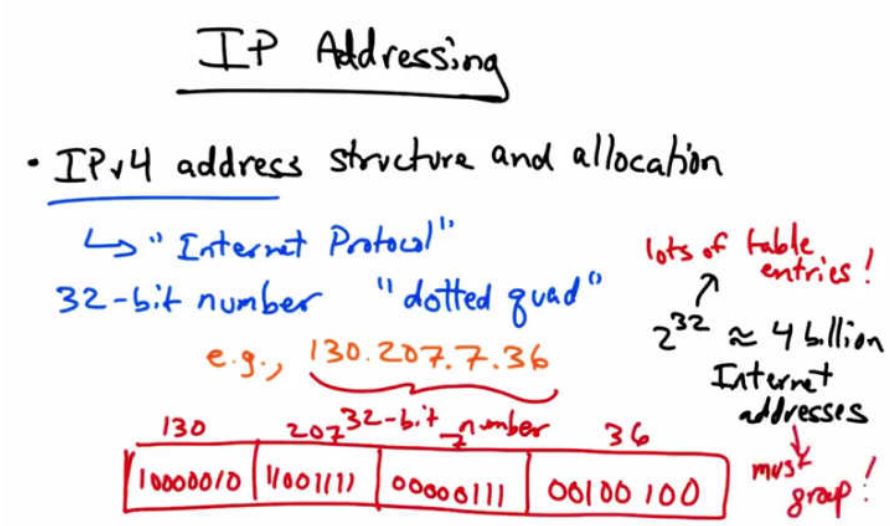


Now, it's very easy to see that this oscillation continues ad infinitum. This particular pathology was first discovered by Varadhan, Govindan, and Estrin, in a paper called persistent route oscillation in interdomain routing, in 1996. Later, Tim Griffin formalized this pathology and derived conditions for stability. Those stability conditions came to be known as a BGP correctness property called safety. It turns out that if ASes follow the ranking and export rules that we discussed, that safety is guaranteed. But, there are various times when those rules are violated. Business relationships, such as regional peering and paid peering, can occasionally cause those conditions to be violated. So as it turns out, to this day, BGP is not guaranteed to be stable in practice, and many common practices result in the potential for this type of oscillation to occur.

CS 6250: Computer Networking

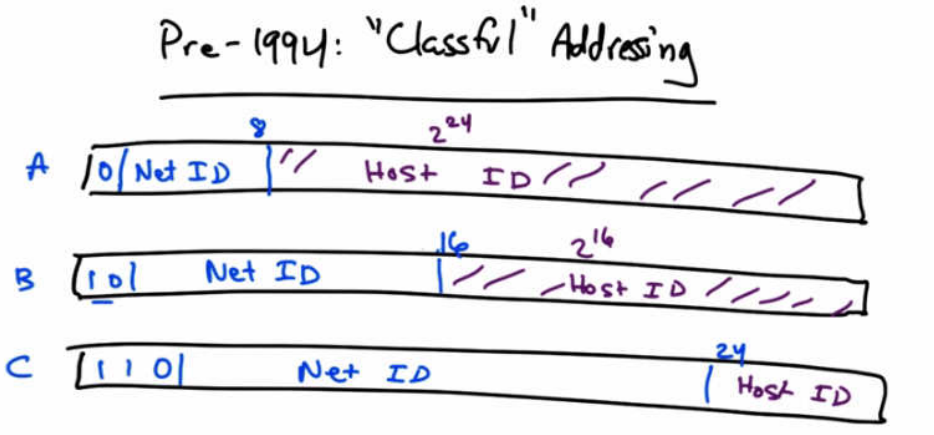
Lecture 5: Naming, Addressing & Forwarding

IP Addressing

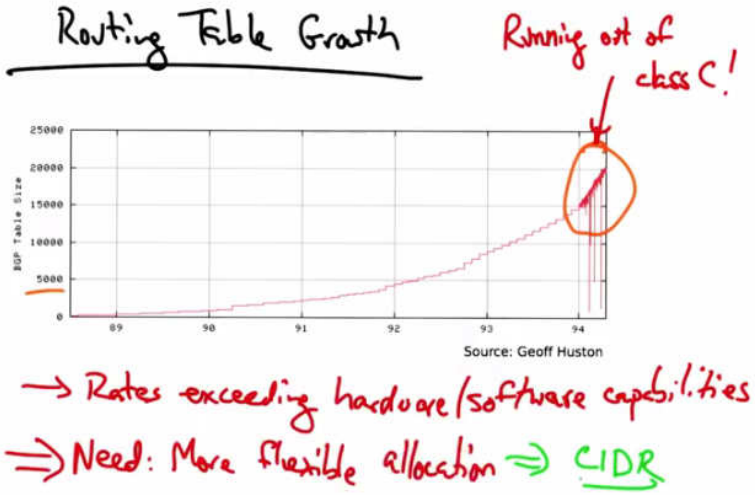


In this lesson we will be covering IP addressing. In particular we will be covering IPv4 address structure and allocation. IP stands for Internet Protocol, and version four is the version of the protocol that is widely deployed on the Internet to date. Each IP address is a 32-bit number. And that 32-bit number is formatted in what is called dotted quad notation. For example, the IP address for www.cc.gatech.edu, is 130.207.7.36. And this is just a convenient way of writing a 32 bit number. So 130 represents 8 bits, and 207 is another eight bit number, 7 is another eight bit number, as is 36. This structure allows for two to the 32, or about 4 billion Internet addresses. Now that sounds like a lot of addresses. As it turns out it's actually not enough, and we're running out of IP addresses, as I'll discuss in a later lesson. But even if we only had to deal with two to the 32 Internet addresses, that's still a lot. Think of it if you have to store every single IP address as an entry in a table. Very quickly that becomes an extremely large table where look-ups can be slow and the memory required to store such a large table might be expensive. So what we need is a more concise way of representing groups of IP addresses. There are different ways to group IP addresses and we'll look at the prevailing method in the next part of the lesson. But first, let's look at how this was done before 1994.

Pre 1994 Classful Addressing



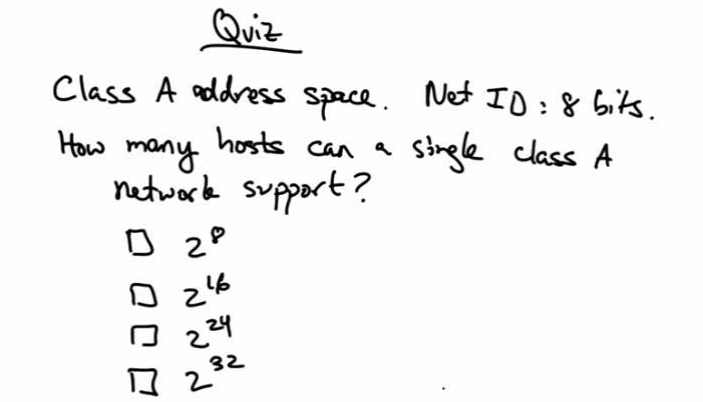
Before 1994, addresses was divided into a Network ID portion and a Host ID portion. So if we take our 32-bits, suppose the first bit is a zero. Note that that's half of all IPv4 address space. Anything that starts with a 0 is going to be known as a class A address and the next 7 bits will represent the network ID or the network that owns this portion of address space. The remaining portion of the address space is dedicated for hosts on that network. In this case, any class A network can support up to two to the twenty-fourth hosts. Addresses that started with one zero were designated as class B networks, where the first 16 bits signified the Network ID and the remaining 16 bits signified the Host ID for that network. Note here that each class B address range represents about one sixty-five thousandth of all internet address space. So, discounting the first two bits which indicate that this is a class B network, we have about 2 to the 14th, unique class B's, each of which can have two to the sixteenth, or 65,000 hosts on each network. Class C's use the first 24 bits for the net ID and the remaining 8 for host ID. So each class Cnetwork essentially can have only 255 hosts on it.



This plot shows the BGP routing table size as a function of the year, starting in 1989, and going up to the internet routing table is quite small. It started at less than 5,000 prefixes. By comparison

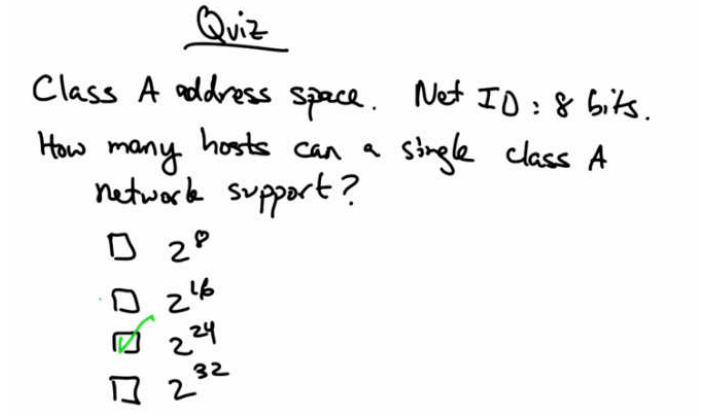
now we have about 500,000 IP prefixes in the internet routing table. But we can see in this period, that internet routing table growth began to accelerate, in particular the growth rates were exceeding the advances in hardware and software capabilities. And, in particular, we began to run out of the class C address allocation. There were far more networks that needed just a handful of IP addresses, such as a class C address space could provide, and yet because only a certain range of the IP address space could be used for class C addresses we began to run out fairly quickly. So there began to be a need for more flexible allocation. The solution to this problem is something called classless interdomain routing, or CIDR. Something that we'll cover in the next lesson.

IP Addressing Quiz



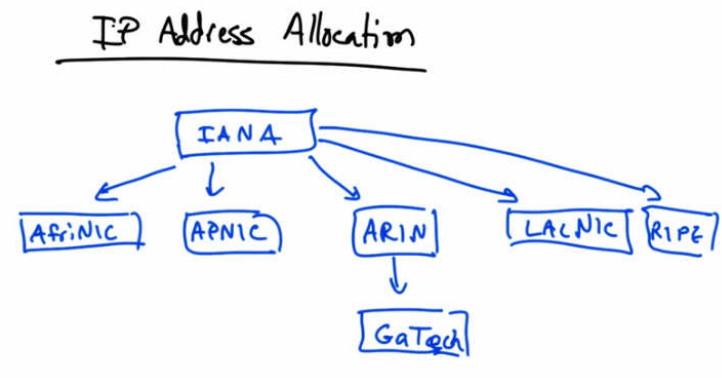
As a quick quiz, suppose you have a class A address space which means that the network ID is eight bits. How many hosts can a single class A network support? Is it to the 2 to the 8th, 2 to the 16th, 2 to the 24th or 2 to the 32nd?

IP Addressing Solution

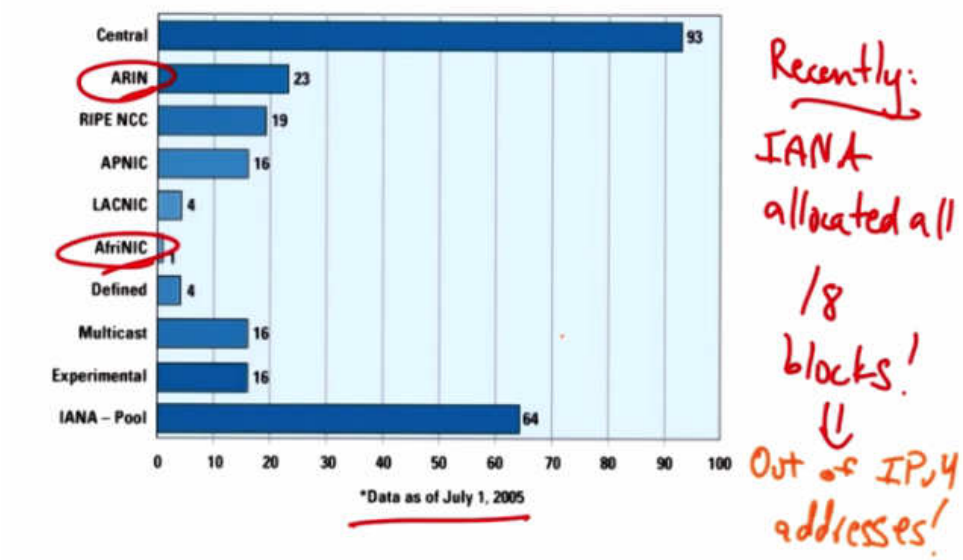


Each Class A address space has a network ID of 8 bits meaning that there are 24 bits that remain for the host ID. This means that each Class A network can support up to 2 to the 24th hosts.

IP Address Allocation



Let's take a look at how IP address space is allocated to Internet service providers. At the top of the hierarchy is an organization called the Internet Assigned Numbers Authority, or IANA. IANA has the authority to allocate address space to what are called regional routing registries. For Africa, the regional registry is called AFRINIC, for Asia and Australia the registry is called APNIC, for North America, ARIN, for Latin America it's LACNIC, and for Europe it's RIPE. ARIN in turn allocates address space to individual networks, like Georgia Tech. The address space across registries is far from even.

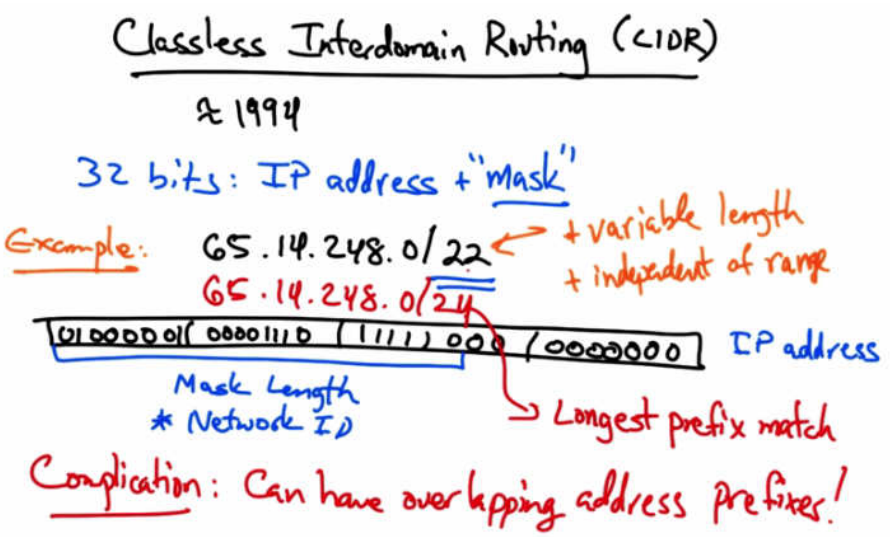


Now, this graph is from a journal article that's a little bit dated now, but it gives you an idea of how many /8 address allocations have been allocated to each of the regional registries. So as of 2005 North America had 23 /8s allocated to it but the entire continent of Africa had only one. And the recent news is that IANA actually finished allocating all remaining /8 Internet address blocks, essentially meaning that we're out of IPv4 address space. So when you hear that we're out of IPv4 addresses, it doesn't mean that you can no longer attach a new device to the Internet. There are various ways for coping with this pressure on address space. What that means is that IANA no longer has any more /8 blocks to give to these regional registries.

```
lawn-143-215-205-52:~ udacity$ whois -h whois.ra.net 130.207.7.36
route:          130.207.0.0/16
descr:          Georgia Tech
origin:         AS2637
mnt-by:         MAINT-AS2637
changed:        scott.friedrich@oit.gatech.edu 20010318
source:         RADB

route:          130.207.0.0/16
descr:          Georgia Tech
admin-c:        SFZ177
tech-c:         SFZ177
origin:         AS2637
remarks:        Georgia Institute of Technology
notify:         scott@gatech.edu
mnt-by:         PEACHNET-MNT
changed:        scott@gatech.edu 20040305
source:         LEVEL3
```

Querying an IP address using Whois and a routing registry, such as ra.net, will tell you the owner of that particular prefix. For example, if we run a Whois query on an IP address at Georgia Tech, it will tell us that that IP address is from a /16 allocation, that Georgia Tech is the owner of the prefix, and it's associated with this autonomous system number. The routing registry entry also gives us some additional information, such as who to contact if we need to contact the owner of this address space.

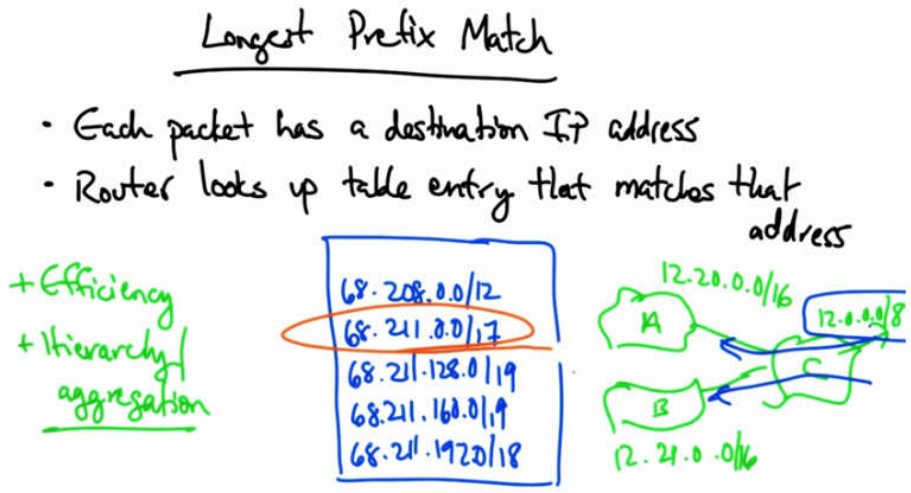
Classless Interdomain Routing



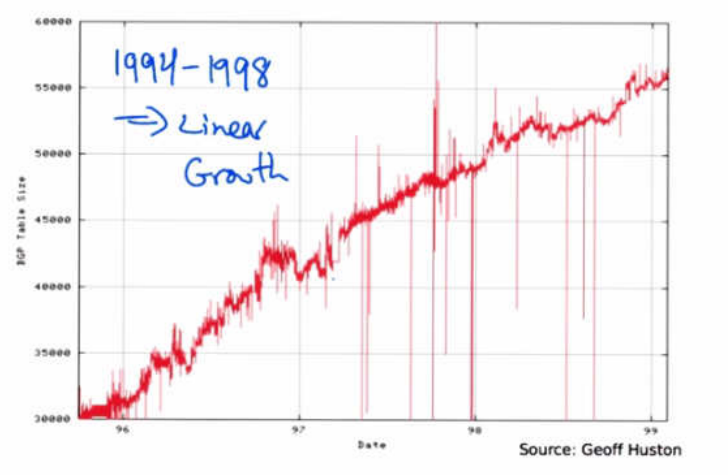
The pressure on address space usage spurred the adoption of classless interdomain routing or CIDR which was adopted in 1994. The idea is that instead of having fixed network ID and host ID portions of the 32 bits, instead, we would simply have an IP address, and what is known as a mask, where the mask is variable length and indicates the length of the network ID. So, for example, suppose we have an IP address like 65.14.248.0/22. Well in this case our 32 bits look like so, but this doesn't tell us how long the network IDand how long the host ID should be. The /22 indicates the mask length, which says that the first 22 bits should represent the network ID.

Now the key is that this mask can be variable length. And the mask length no longer depends on the range of IP addresses that are being used. This allows those allocating IP address ranges, to both allocate a range that's more fitting to the size of the network and also not have to be constrained about how big the network ID should be depending on where in the IP address space the prefix is being allocated from. Of course now the complication is that it's possible to have overlapping address prefixes. For example, 65.14.248.0/24 overlaps with 65.14.248.0/22. The red prefix is actually a subset of the black one. So supposing these two entries both show up in an Internet routing table, what are we supposed to do? The solution is actually to forward on what's called the longest prefix match, meaning that if a routing table has two overlapping entries that it should forward according to the entry that has the longest prefix, or the longest mask length. Intuitively that makes sense because the prefix with the longer mask length is more specific than the prefix with the shorter mask, or the larger prefix.

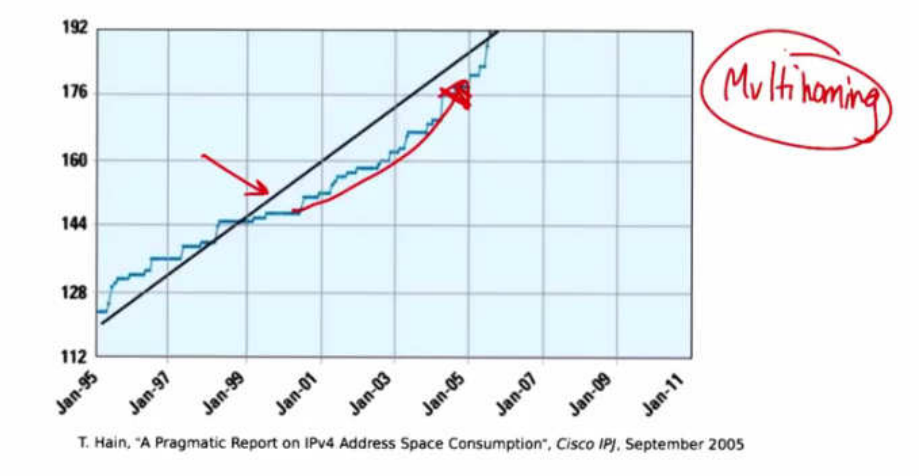
Longest Prefix Match



Let's take a closer look at longest prefix match. So each packet has a destination IP address, which determines where the package should be forwarded next, and a router basically looks up a table entry that matches that address. So, for example, a forwarding table might have a number of prefixes in it, and many of these prefixes might be overlapping. But when we see an IP address, it may match on one or more prefixes in this table, you simply match that IP address to the entry in the forwarding table with the longest matching prefix. So the benefits of cider and longest matching prefix are efficiency, since prefix blocks can be allocated on a much finer granularity than with classful inter-domain routing, and the opportunity for aggregation if two downstream networks with more specific or longer prefixes, should be treated in the same way by an upstream network, who might simply aggregate two contiguous shorter prefixes into one forwarding table entry with a shorter prefix. For example, a benefit for aggregation might exist if two downstream networks A and B each had slash 16 address space allocated to them. But upstream, all the traffic always came through the same upstream network, C. If the rest of the internet only reached A and B via C, then the rest of the internet need only know about C's address space which might be 12/8. This might allow the upstream network to simply aggregate, or not announce these more specific prefixes, since they're already covered by the less specific upstream prefix.

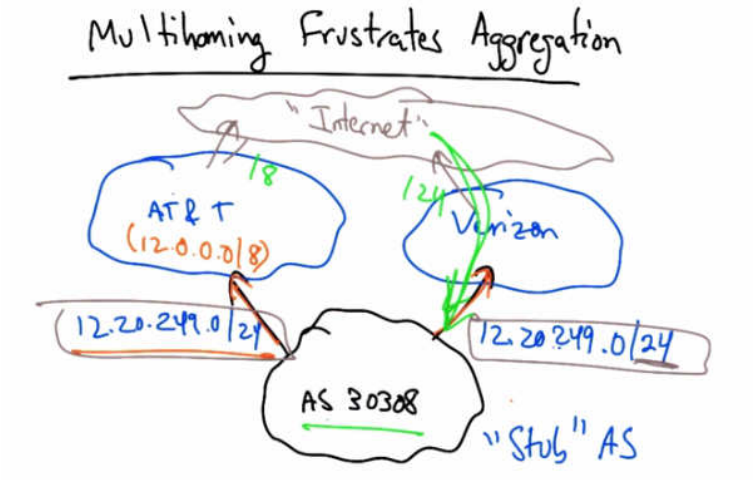


Now cider had a significant effect on slowing the growth of the internet routing tables from 1994 to 1998. So, from 1994 to 1998, we see roughly linear growth in the number of prefixes in the internet routing table. Around 2000, fast growth in routing tables resumed.

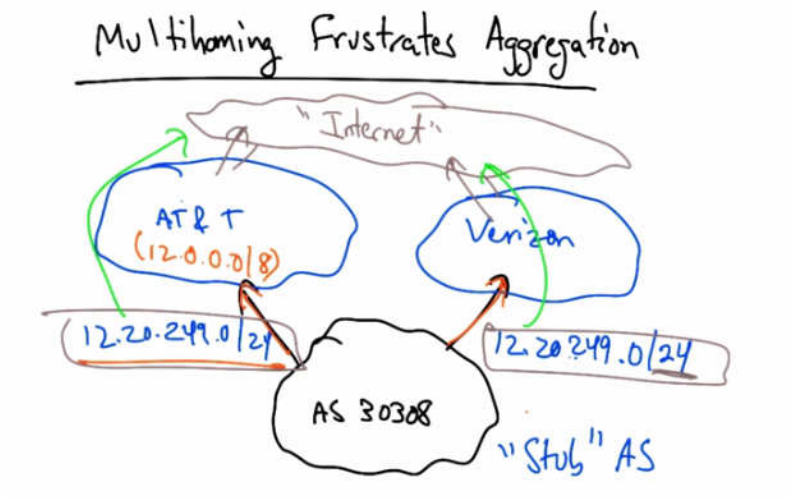


You can see that growth here once again started to pick up a significant contributor to this growth, was a practice called multi-homing. Multi-homing can actually make it difficult for upstream providers to aggregate IP prefixes together, often requiring an upstream provider to store multiple IP prefixes for a single autonomous system. Sometimes those IP prefixes are contiguous and sometimes they aren't. Let's take a quick look at how multi-homing can stymie aggregation.

Multihoming Frustrates Aggregation



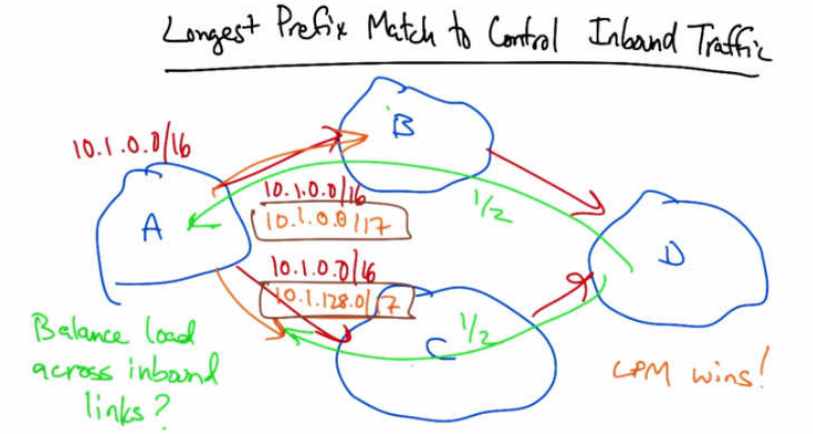
This example, a stub AS, in this case 30308, might receive IP address space, say, 12.20.249/24, from one of its providers, such as AT&T, which happens to own 12.0.0.0/8. Now in this case AS 30308 wants to be multihomed. In other words, it wants to be reachable via two upstream Internet service providers. In this diagram, the two Internet service providers are AT&T and Verizon. To be reachable by both of these ISPs, AS 30308 has to advertise its prefix, which it received from AT&T via both AT&T and Verizon. The problem occurs when AT&T and Verizon want to advertise that prefix to the rest of the internet. Well, unfortunately, although AT&T might like to aggregate this prefix as I previously described, it can't. If it did, Verizon would still be advertising the longer /24 via it's upstream link. And because of longest prefix match, all of the traffic would then arrive via the Verizon link regardless of what AS 30308 wanted to have happened to that incoming traffic.



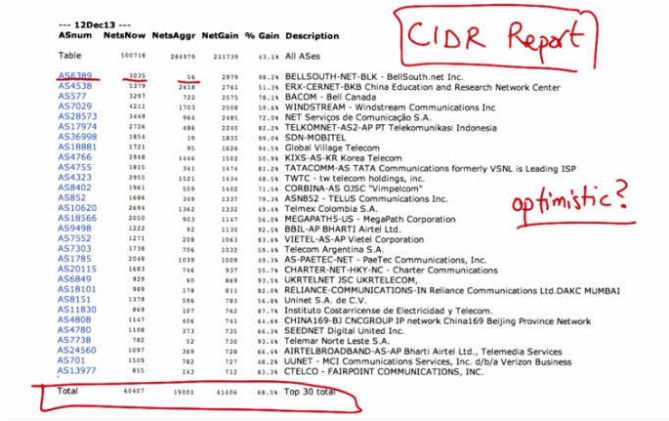
As a result, both AT&T and Verizon must advertise the same /24 to the rest of the internet. This results in an explosion of /24s in the global internet routing table. You can imagine, that if a lot

of stub AS's wanted to be multihomed, then suddenly, we've got a lot more /24s in the global routing table than might otherwise exist without multihoming.

Longest Prefix Match to Control Inbound Traffic

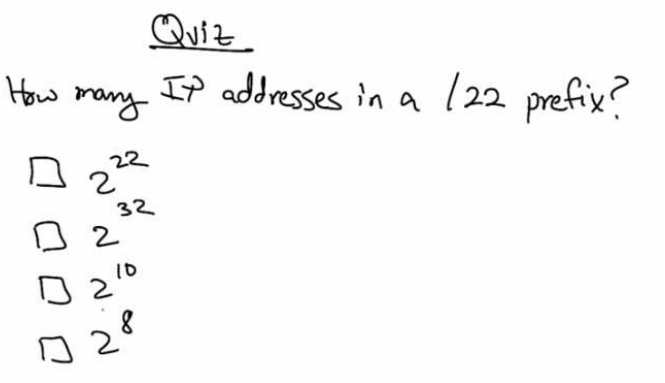


Now in a previous lesson, we looked at how AS path prepending, can be used to control inbound traffic. As it turns out, longest prefix match can also be used to control inbound traffic. Suppose that AS A owns 10.1.0.0/16, and it might advertise that prefix out both of its upstream links and that route might similarly be advertised further upstream. Now of course as we know from a previous lesson, given the advertisement of one prefix upstream, AS D is going to pick one best BGP route along which to send traffic back to A. But let's suppose that AS A wanted to balance that traffic across its incoming links. Well in that case, ASA could actually advertise routes for 2 more specific prefixes, effectively splitting the slash 16 in half, so in addition to advertising 10.1/16, across both links, AS A might advertise 10.1/17 on the top link and 10.1.128.0/17, the other half of the /16 on the bottom link. Now, if either link fails, the covering /16 will ensure that the prefix remains reachable by one of the two upstream links. But because longest prefix match wins, the traffic for 10.1.128 would now traverse the bottom link, and the traffic for 10.1/17 would now traverse the top link, effectively sending traffic for half of the prefixes along the top path and traffic for the other half of the prefixes along the bottom path. Although we just explored a perfectly good reason to deaggregate a contiguous prefix, it turns out that sometimes autonomous systems may deaggregate larger prefixes unnecessarily.



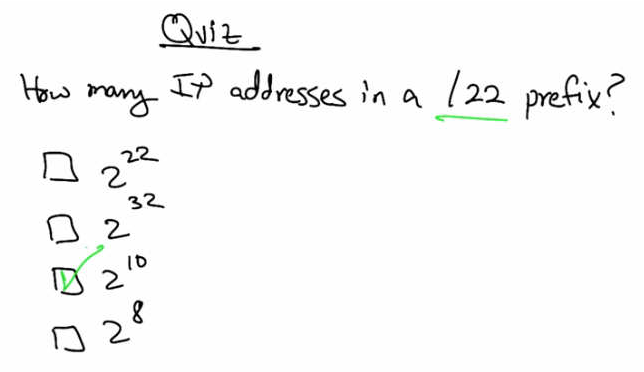
A report called the CIDR Report, which is released weekly, shows autonomous systems who are advertising IP prefixes that, at least according to observation, are continuous and could be aggregated. For example, the top offender for the week of December 12th, 2013, was AS6389. This single autonomous system is actually advertising more than 3,000 unique IP prefixes. The CIDR report analysis suggests that with appropriate aggregation, this autonomous system could instead advertise only 56 unique IP prefixes. Now this might be overly optimistic. As we just explored, there are perfectly good reasons to deaggregate a contiguous IP prefix into multiple smaller contiguous IP prefixes. But nonetheless, the report shows that there are probably a lot more IP prefixes in the Global Internet Routing table than there could be if AS's took full advantage of aggregation.

CIDR Quiz



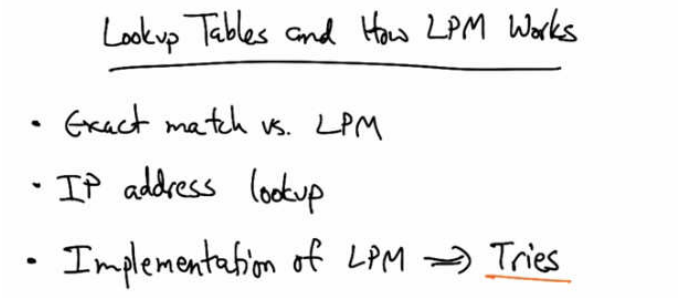
Let's have a quick quiz about cider. So, how many IP addresses does a /22 prefix represent? Two to the 22, two to the 32, two to the tenth, or two to the eighth?

CIDR Solution



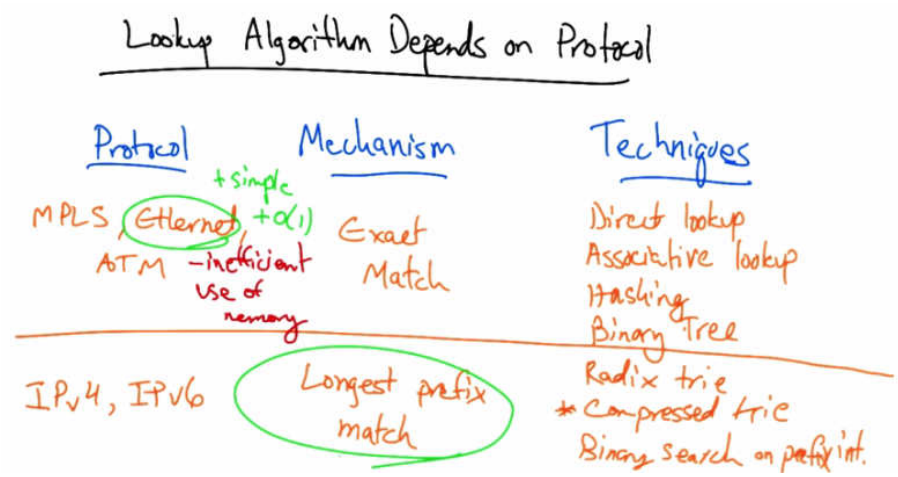
The /22 represents the length of the network ID, and the remaining 10 bits are for hosts in that /22 prefix. So those 10 bits reserved for the host for that /22 mean that this /22 prefix represents 2 to the tenth IP addresses.

Lookup Tables and How LPM Works (put with other slide)



Okay, in this lesson, we will explore how lookup tables in routers are designed and how longest prefix match works; we'll explore exact match versus longest prefix match and when each is used; we'll explore IP address lookup in more depth; and finally, we'll explore how longest prefix match is implemented in the form of tries.

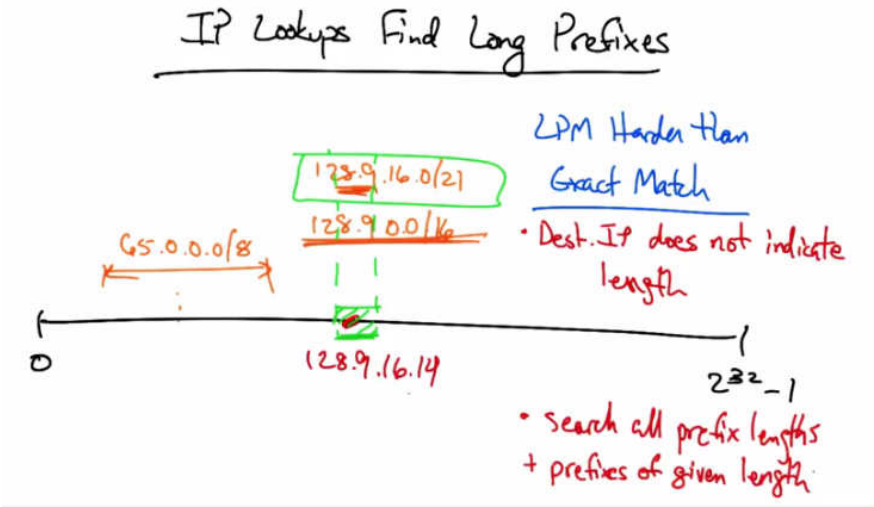
Lookup Algorithm Depends on Protocol



So, the look up algorithm that a router uses depends on the protocol that it's using to forward packets, and the look up mechanism might be implemented with a variety of different algorithms or techniques. For example, MPLS, Ethernet, and ATM use an exact match look up. Exact matches can be implemented as a direct look up, an associative look up, hashing, or via a binary tree. IPv4 and IPv6 on the other hand are implemented with what's called longest prefix match. We've already looked at longest prefix match a little bit in some lessons and, in this lesson we'll look at it in a bit more detail as well as how it's implemented. It might be implemented as a radix trie, a compressed trie, which is something that we will look at in this lesson, and it can also be implemented as a binary search on the prefix intervals. Ethernet based forwarding is based on exact match of a layer two address which is usually 48 bits long. It's address is global, not just local to the link. And the range or size of the address is not negotiable. Now 2 to the 48th is far bigger than 2 to the 12th, therefore, it's not possible to hold all the addresses in the table and use direct look up. The advantages of exact matches and Ethernet switches is that exact match is simple and the expected lookup time is small, or O of 1. But the disadvantages include inefficient

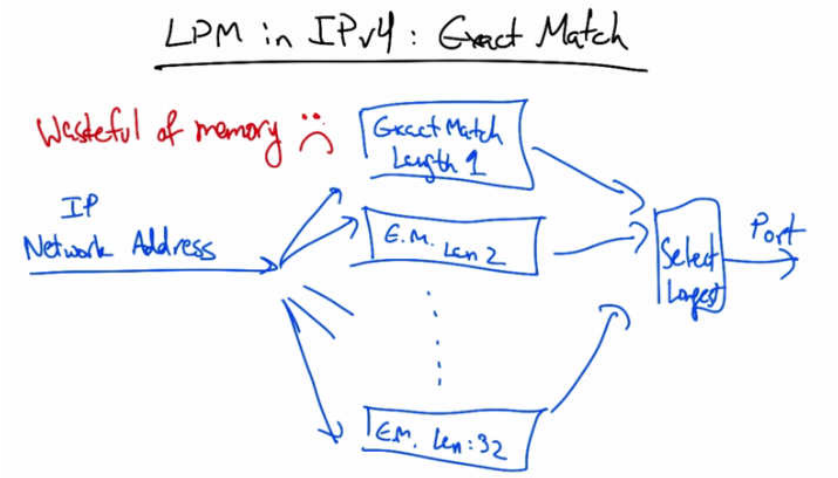
use of memory. This potentially results in nondeterministic lookup time if the lookup might require multiple memory accesses. Lets now take a closer look at longest prefix match.

IP Lookups Find Long Prefixes



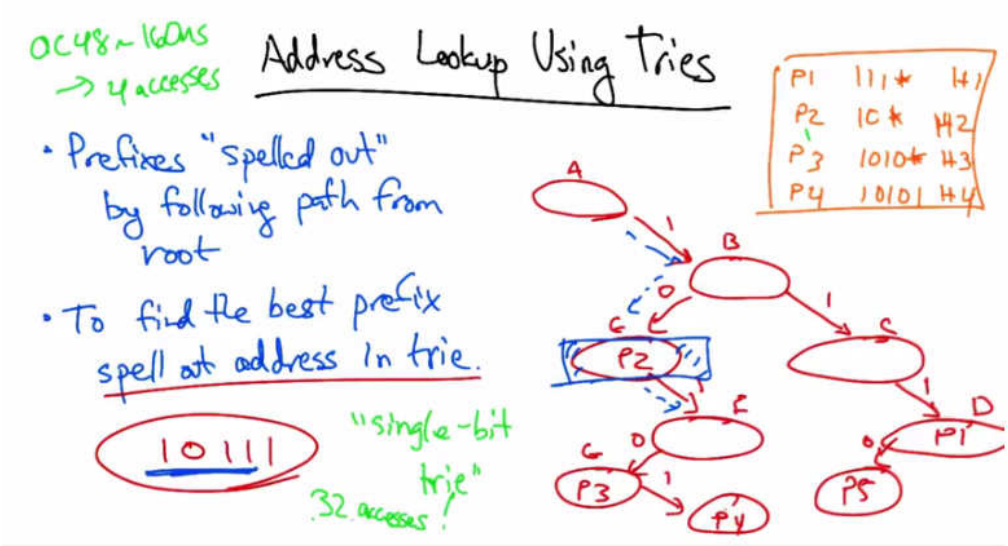
IP lookups find longest prefixes. Let's suppose that we want to represent a particular IP address as one point in the space from zero to 2 to the 32 minus 1, or the range of all 32 bit IP addresses. Each prefix represents a smaller range inside this larger range of 32-bit numbers. Obviously, this is not to scale. Now these ranges, of course, might be overlapping, as is shown here, and the idea is that longest prefix match will match the smallest prefix for which the IP address range overlaps that of the specified IP address. So longest prefix match is harder to perform than exact match. For one, the destination IP address does not indicate the length of the longest matching prefix, so some algorithm needs to determine the length of the longest matching prefix, which in this case is 21. So we somehow need a way to search the space of all prefix lengths, as well as prefixes of a given length.

LPM in IPv4 Exact Match



Suppose, for example, that we wanted to implement longest prefix match for IPv4 using exact match. Now in this case we might take our network or our IP address, and send it to a bunch of different exact match tables. And then among the tables that had a match, we would select the longest, and then forward the packet out the appropriate output port. Of course, this is horribly inefficient, because we'd have to have tables for each of these links, and every time a packet arrived, we'd have to send it to each one of these 32 tables. This is extremely wasteful of memory.

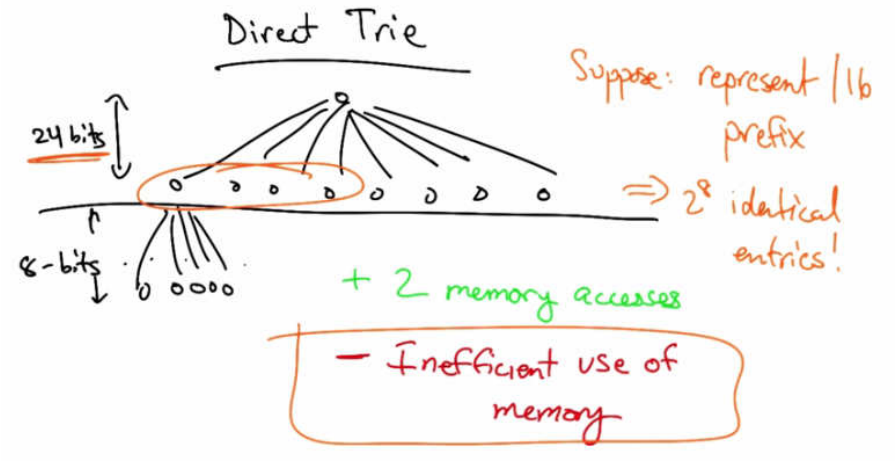
Address Lookup Using Tries



An alternative is to perform address lookups using a data structure called a trie. In a trie, prefixes are spelled out by following a path from the root. And to find the best prefix, we simply spell out the address in the trie. For example, let's suppose we had the following table. Such a lookup table has entries of varying lengths. Let's see how this might be encoded in a trie. In a trie, spelling out the bit one always takes us to the right, and spelling out the bit zero always takes us to the left. So to insert one one one star, we'd basically start here. One. One. One. And then we insert P1, and then we repeat this process. One zero star results in P2. One zero one zero results in P3. And one zero one zero one results in P4. If we want to insert one one one zero, insertion is easy. We can simply insert P5 as such. Look ups are easy, so for example let's suppose we want to look up 10111. Well all we have to do, is spell this out in the trie. So we can follow 1-0-1 and now, we see that there's no entry for 1011. So, we use the entry of the last node in the tree that we traverse that has an entry, in this case P2. Now this structure here is what's called a single bit try. Single bit tries are very efficient. Note that every node in this try exists due to one of the five folding table entries that we've inserted in the try. So, a single bit trie is a very efficient use of memory. Updates are also very simple. We saw how easy it was, to insert the entry for P5. Unfortunately, the main problem is the number of memory accesses that are required to perform a lookup. For 32 bit address, we can see, that looking up the address in a single bit trie, might require 32 look ups, in the worst case, one for each bit. So it's each bit in the address requires, one traversal in the trie, or one memory look up. So this could be very bad. At worst, 32 accesses in the worst

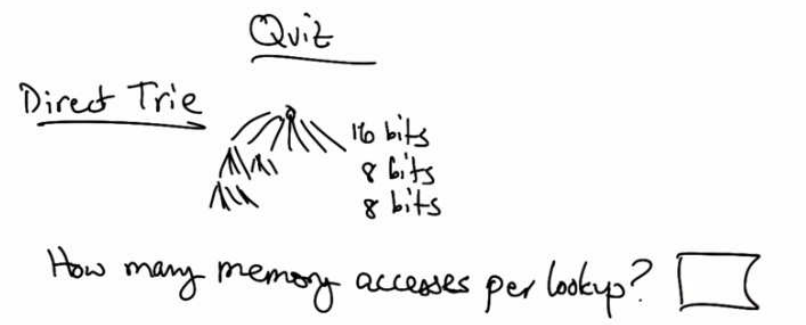
case. To put this in perspective, an OC48 requires a 160 nanosecond lookup, or simply 4 memory accesses. So 32 accesses, is far too many, especially for high speed links.

Direct Trie



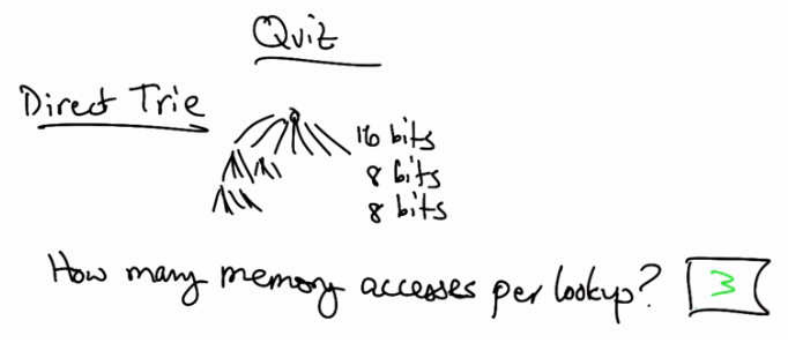
The other extreme, of course, is to use a direct trie where instead of 1 bit per look up we might have 1 memory access responsible for looking up a much larger number of bits. So, for example, we might have a two level try where the first memory access is dictated by the first 24 bits of the address, and the second memory access is dictated by last 8 bits of the address. Now here we can look up an entry in the forwarding table with just two memory accesses. The problem is that this structure results in a very inefficient use of memory, unlike the single bit trie. To see why, suppose that we want to represent a /16 prefix. Well unfortunately we have no way of encoding a lookup that's just 16 bits. We have to rather encode 2 to the 8th identical entries, corresponding to the 2 to the 8th /24 prefixes that are contained in that /16, so this is extremely inefficient use of memory.

Direct Trie Quiz



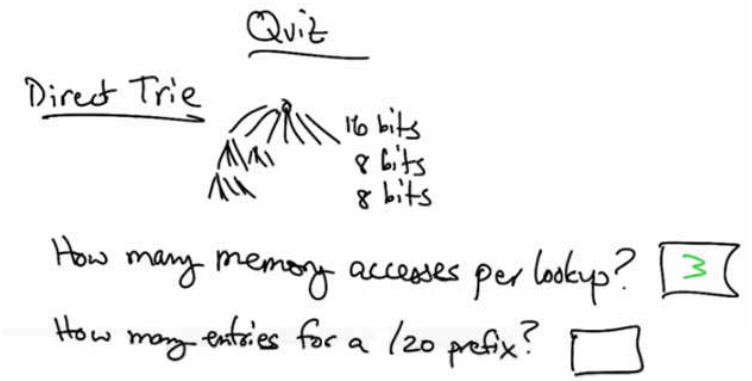
As a quick quiz, suppose we have a direct trie, and the first level is 16 bits, the next level is eight bits, and the third level is the final eight bits. In the worst case, how many accesses would be required per lookup?

Direct Trie Solution



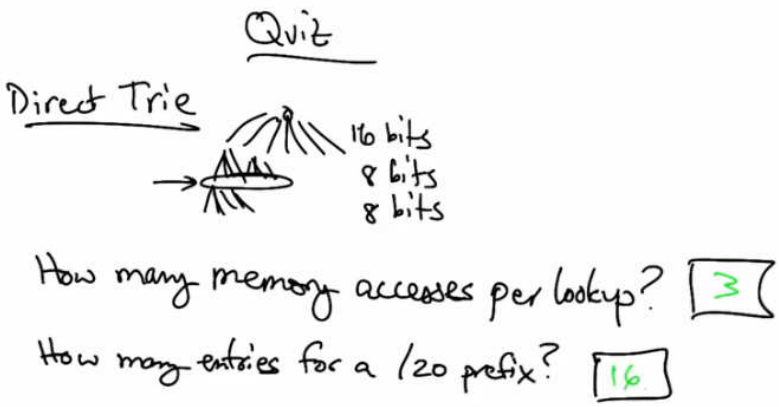
Because the Trie has a depth of three, in the worst case, a look up might require three memory accesses.

Direct Trie Quiz 2



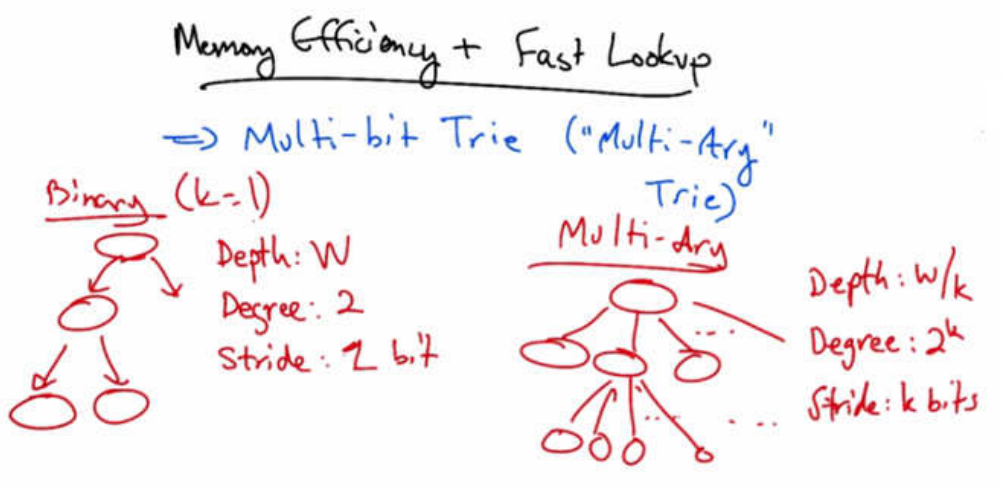
How many entries, would I need, to represent a /20 prefix?

Direct Trie Solution 2



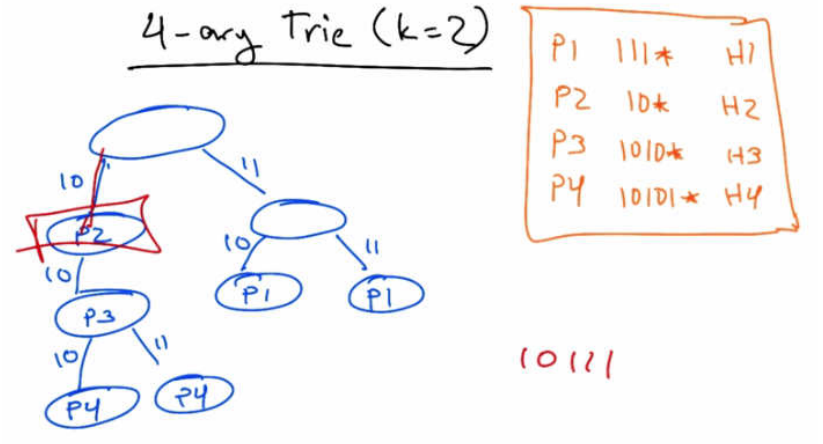
A /20 prefix is 2 to the 4th, or 16, /24's. And I need to basically represent 16 entries, at the 24 bit level of the trie, or the second level, and therefore, I'd need 16 entries to represent, a /20 prefix.

Memory Efficiency and Fast Lookup



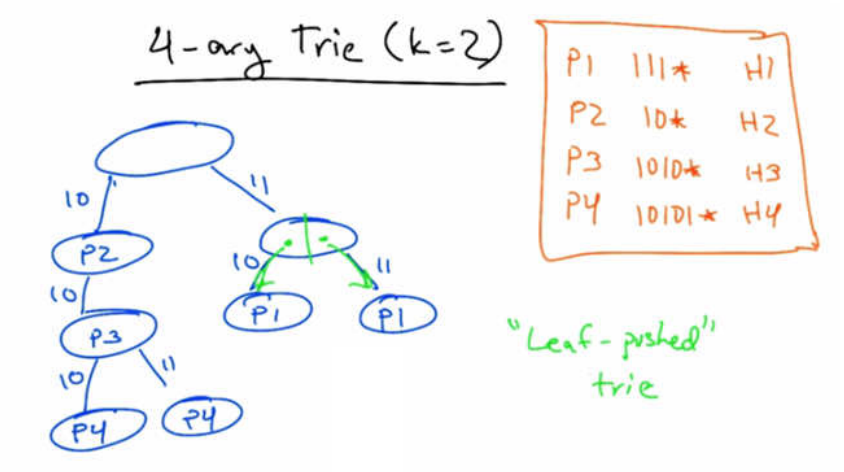
To achieve the memory efficiency of a single bit trie with the fast lookup properties of a direct trie, a compromise is to use what's called a multi-bit trie, or a multi-ary trie. Let's start with a binary trie, where one bit is resolved at each node. Here, the depth is big W, the degree of each node is two, and the stride for each lookup is one bit. Now we can generalize this to a multi-ary trie, where the depth is now W over K if the degree is 2 to the K, and each level resolves K bits. The binary trie is a simple case of the multi-ary trie, where K equals 1.

4 ary Trie

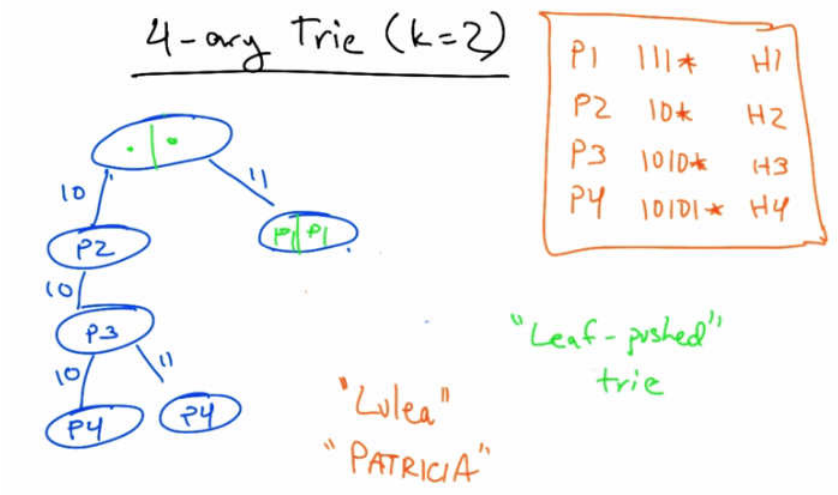


Le'ts take a look at the 4-ary trie where k equals 2. Suppose we have the same forwarding table as before. But now, each node in the trie is responsible for resolving two bits. So if we take one one, and now we take one star, that's one zero and one one. And now we basically have to put p1 in two places in the tree. One zero star results in just one entry. 1010 star results in two

traversals, and 10101 star again represents two entries, for 101010 and 101011. Now suppose we want to look up 10111. Again, we can spell this out, 101, and we can see that we get no further than P2 and again, we match at P2.

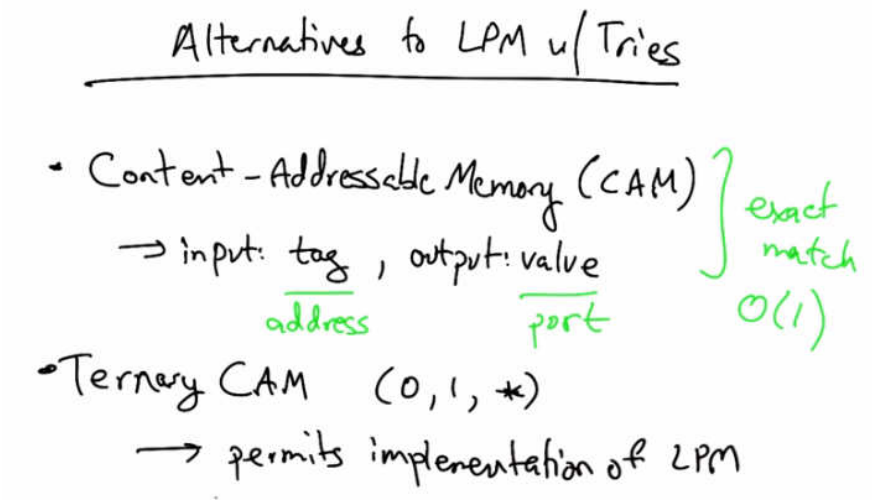


One thing we can do to save space further is create what's called a leaf-pushed trie. In such a setting, we can save our self some space. Instead of having these pointers, we can push these entries into the left and right side of this node, respectively.



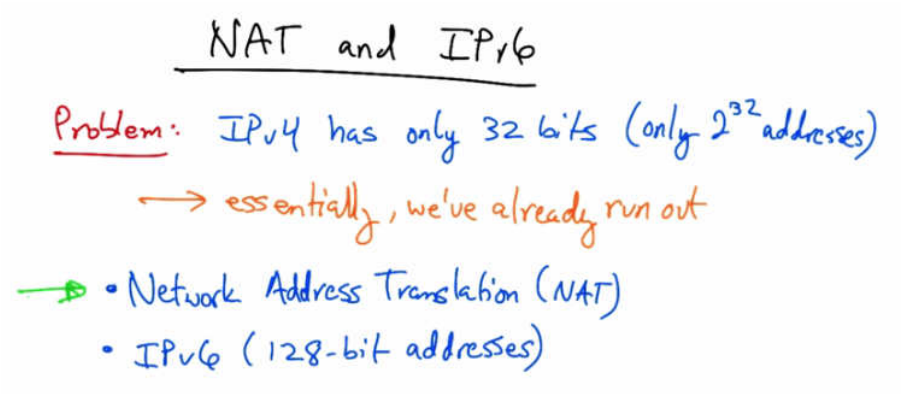
So 10 becomes P1 on the left side and 11 becomes P1 on the right side. There are variety of other optimization algorithms, including one called Lulea and another called Patricia. Each of them use the same basic idea that we have explored here, except some of them like Lulea are a three level trie, and often they use a bitmap to compress out repeated entry such as those that exist here.

Alternatives to LPM with Tries



Now there are alternatives to implementing longest prefixes match with a trie. One could start with a content addressable memory or a CAM. Now a CAM is a hardware base route look up where the input is a tag and the output is a value. So, for example, the tag might be an address and the value might be the output port. Now the CAM really only supports exact match but it is an O of 1 lookup. There is something called a ternery CAM, where instead of exact matching In the tag, you can have 0, 1, or don't care, or a star. The ternary CAM and in particular its support for a wild card permits an implementation of longest prefix match. One can thus have multiple matching entries, but prioritize the match according to the longest prefix in the ternary CAM.

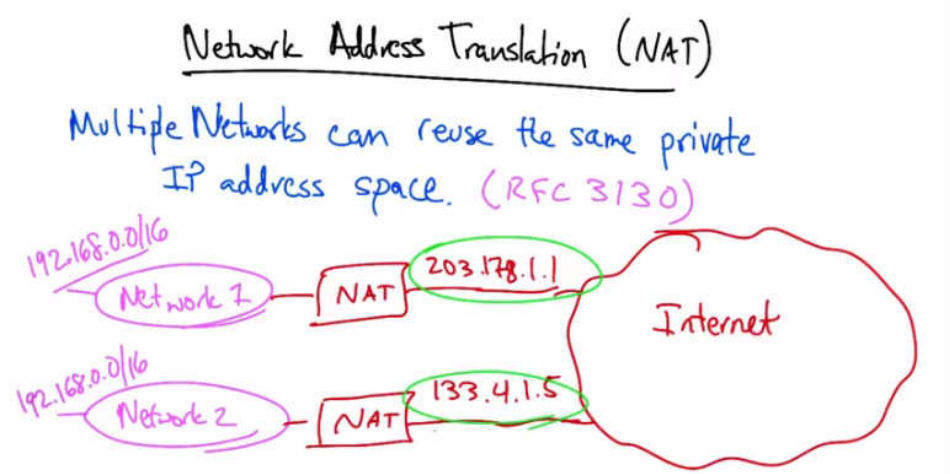
NAT and IPv6



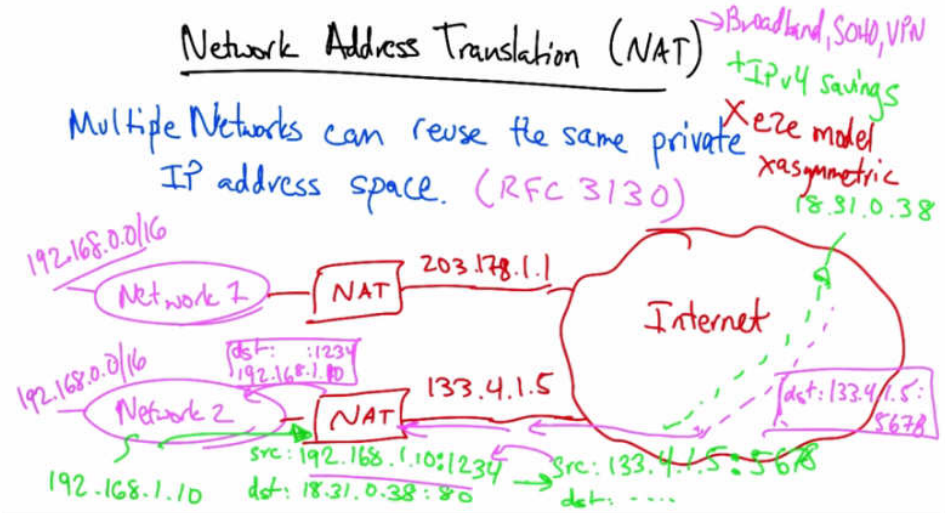
Let's now talk about various problems that resulted from IPv4 and the growth of the internet routing tables, and two different solutions to internet routing table growth: network address translation, or NAT, and IPv6. So the main problem that we are seeing is that IPv4 addresses have only 32 bits, which means that there can only be a total of 2 to the 32 unique IP addresses. Not only that, as we've seen, IP addresses are allocated in blocks, and fragmentation of this space can mean that IPv4 addresses can be quickly exhausted. In fact, we've already seen the last slash

eight from IPv4 address space allocated to the registries. So we're well on our way towards running out of IPv4 addresses. In some sense, you can say that we've essentially already run out. In this lesson, we're going to look at two solutions: network address translation, or NAT, and IPv6, whose main feature is 128 bit addresses. Let's first take a look at NAT.

Network Address Translation

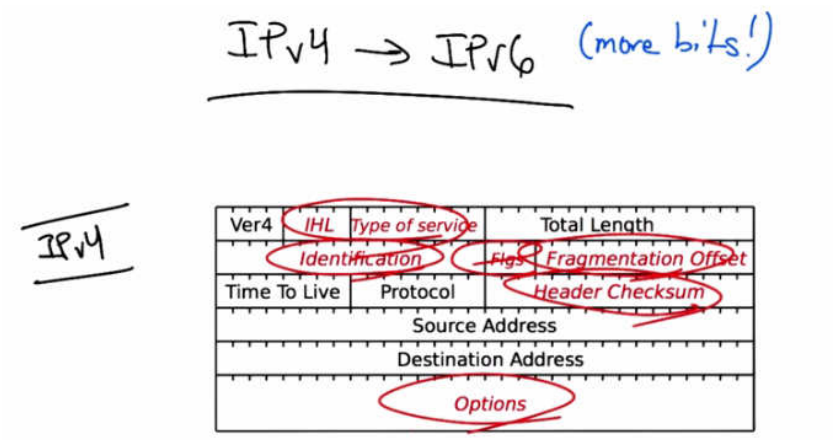


NAT allows multiple networks to reuse the same private IP address space. Let's suppose that we have two networks. These networks might be, for example, homes or they might be larger networks in regions of the Internet, where IPv4 address space is scarce, for example, in developing regions. What NAT allows these networks to do is reuse the same portion of internet address space. For example, a particular, special, private IP address space, is 192.168/16. Other private IP address space is specified in RFC 3130. Now, obviously these two networks couldn't coexist on the public Internet, because routers wouldn't know if they got a packet destined for an IP address in this space, which network the packet should be sent to. What a NAT, or a Network Address Translator does, is take the private IP addresses that are behind the NAT and translate those IP addresses to a single, globally visible IP address.

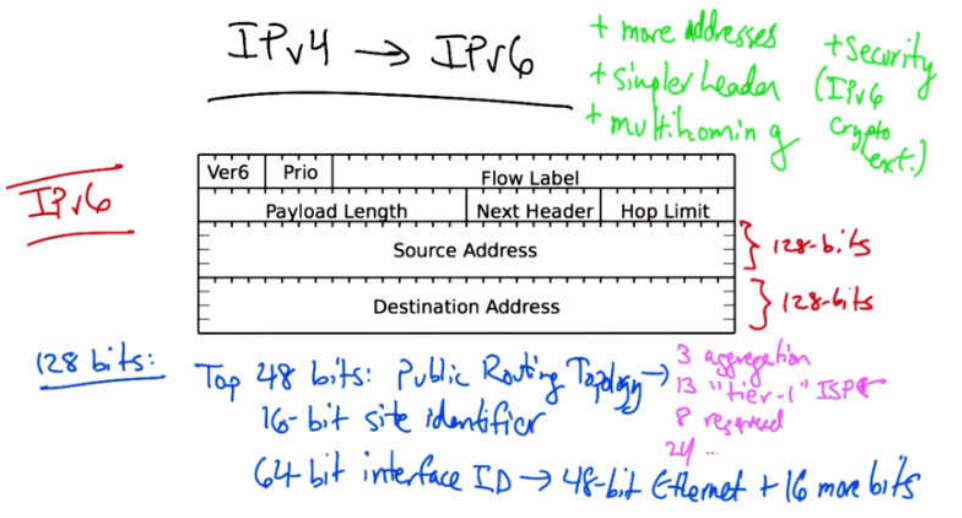


Now, to the rest of the Internet, network one appears to be reachable by a single IP address, 203.178.1.1, and network two is reachable via a single distinct global IP address 133.4.1.5. Now, a host back here, say 192.168.1.10 might send a packet towards a global internet destination. Now, the key behind NAT is that this packet has a source port and the NAT is basically going to take that source IP address and port and it's going to translate it into a publicly reachable source IP address and port, and the destination will remain the same. That packet will make its way to a global destination and the reply will make its way to the globally reachable IP address on the corresponding port. Now, when that packet with that particular destination IP address and port reaches the NAT, the NAT has a table that knows the mapping between that public IP address and port and the private one that it rewrote to generate the corresponding public IP address and port. So we can simply now rewrite the destination IP address of this packet to the corresponding private address and port. NATs are popular on broadband access networks, small or home offices and VPNs. There's a clear savings in IPv4 address space, since there can be many devices in one of these private networks and yet all of the devices that are behind the NAT only use up one public IP Address. The drawback, of course, is that the end-to-end model is broken. And we talked about the end-to-end model in a previous lesson and let me just remind you how NAT breaks it. If the NAT device failed in this instance, for example, the mapping between the private source IP address and port and the public source IP address and port would be lost, thereby breaking all active connections for which the NAT is on the path. It's also asymmetric. Under ordinary circumstances it's rather difficult for a host on the global Internet to reach a device in a private address space in network one or network two, because by default those devices in these private networks do not have public globally reachable IP addresses. So, NAT both breaks end-to-end communication, and it also breaks by directional communication.

IPv4 to IPv6

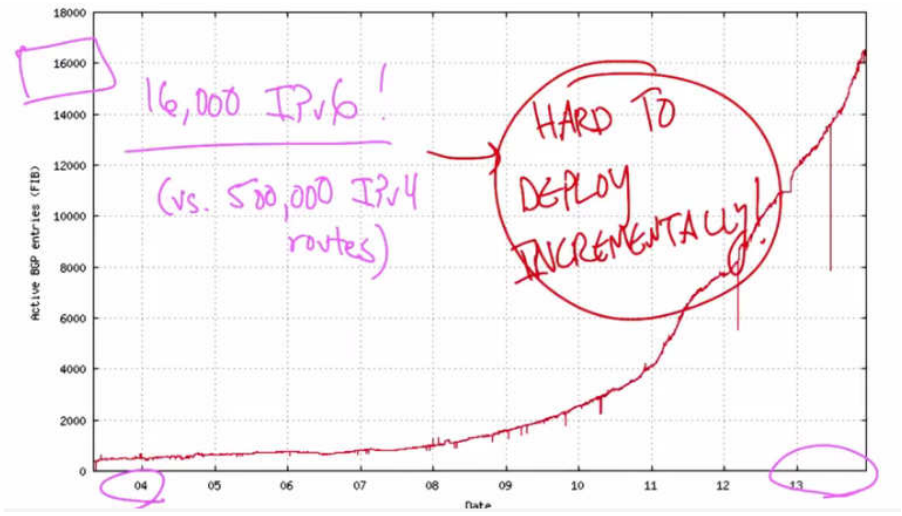


Another possible solution to the IP address space exhaustion problem is to simply add more bits. This is the gist of the contribution of the IPv6 protocol. Here's a picture of the IPv4 protocol header, and all of the fields shown in red have basically been removed in IPv6, resulting in both a much simpler header and addresses that are much larger.



By contrast, here's the IPv6 header. The IPv6 header provides 128 bits for both the source and destination IP addresses. Now the format of these addresses are as follows. Of the 128 bits, the top 48 bits are for the public routing topology, and we have a 16-bit site identifier. And finally, a 64-bit interface ID, which effectively has the 48-bit Ethernet address of the interface plus 16 more bits. Now, the top 48 bits can be broken down further. They include top level provider, something like a tier one ISP, 8 reserve bits, and 24 additional bits. Now, note that there are 13 bits in the top 48 that directly map to the tier one ISP, meaning that addresses are purely provider-based, thus changing ISPs would require renumbering. IPv6 has many claimed benefits. There are more addresses, the header is simpler, multihoming is supposedly easier, various aspects of security are built in, such as the IPv6 crypto extensions. Now despite all of these benefits, we have yet to see a huge deployment of IPv6 yet.

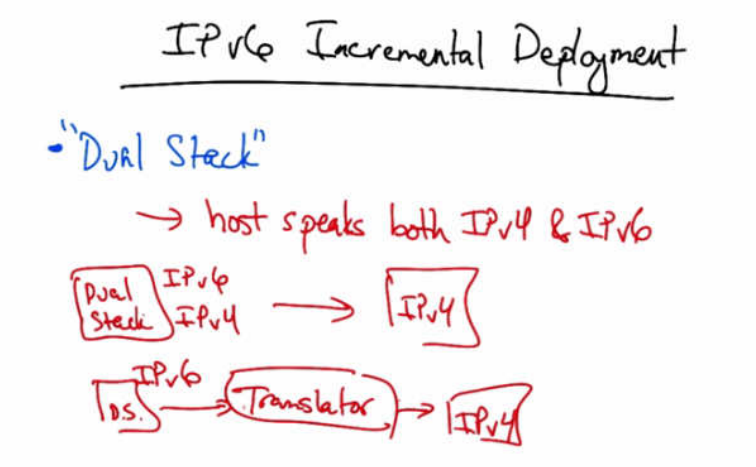
IPv6 Routing Table Entries



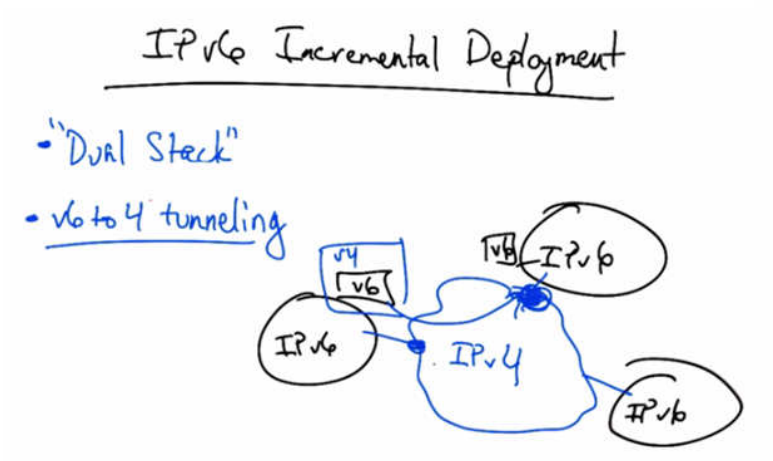
Now despite all of these benefits, we've yet to see a significant deployment of IPv6. Here you can see the number of routing table entries for IPv6 routes, as well as the growth over time from

2004. To the end of 2013. What's remarkable is that we only see 16,000 IPv6 routes in the global routing table. This is not that many considering that there are about 500,000 IPv4 routes in the global routing table. The problem is that IPv6 is very hard to deploy incrementally. Remember our discussion of the narrow waist. Everything runs over IPv4 and IPv4 was designed to run over a variety of physical layers. This common protocol has allowed tremendous growth, but because everything depends on the narrow waist of IPv4 and because IPv4 is built on top of so many other types of infrastructure, changing it becomes extremely tricky. Incremental deployment where part of the internet is running IPv4 and other parts have been upgraded to IPv6 results in significant incompatibility. There are various incremental deployment options for IPv6.

IPv6 Incremental Deployment



One is what's called a dual stack deployment. In a dual stack deployment a host can speak both IPv4 and IPv6. It communicates with an IPv4 host using IPv4 and communicates with an IPv6 host using IPv6. What this means is that the dual stack host has to have an IPv4 compatible address. Either the host has both an IPv4 and an IPv6 address, thus allowing it to speak to an IPv4 host, or it must rely on a translator which knows how to take a v4 compatible IPv6 address, and translate it to the v4 address. One possible way of ensuring compatibility of a v6 address with IPv4, is simply to embed the IPv4 address in 32 bits of the 128 that are allocated for the IPv6 address.

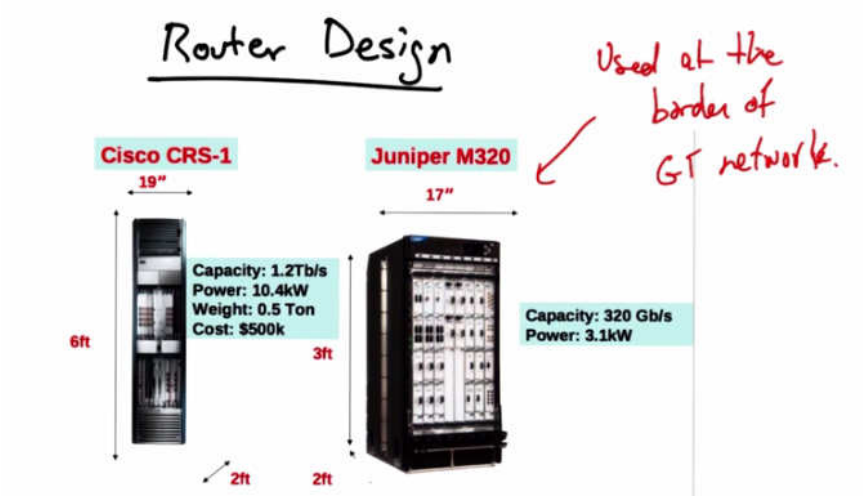


Now, a dual stack host configuration or a v4 compatible IPv6 address solves the problem of host IP address assignment, but it doesn't solve the problem that IPv6 deployments might exist as islands. For example, multiple independent portions of the Internet might deploy IPv6, but what if the middle of the network only speaks in routes IPv4? The solution here is to use what's called 6 to 4 tunneling. In 6 to 4 tunneling, a v6 packet is encapsulated in a v4 packet. Now, that v4 packet is routed to a particular v4 to v6 gateway corresponding to the v6 address that lies behind that gateway. And at this point the outer layer of encapsulation can be stripped, and the v6 packet can be sent to its destination. This of course, requires the gateways at the boundaries between the v4 and v6 networks to perform encapsulation of the packet as it enters the v4 only part of the network, and de-capsulation as the packet enters the v6 island, where the destination host resides.

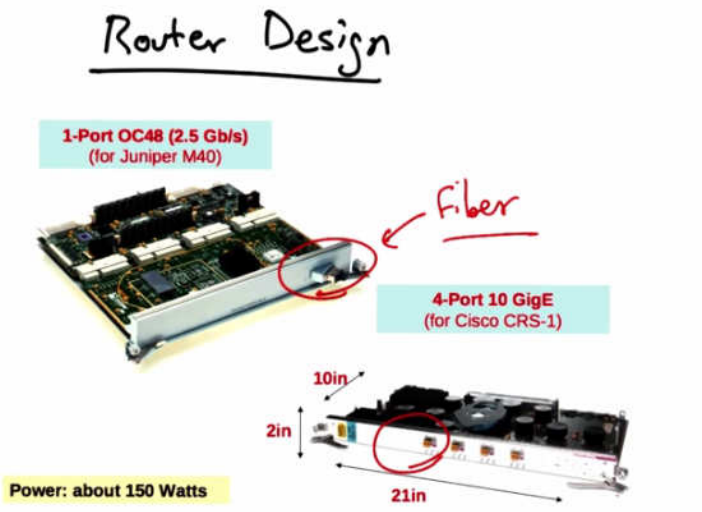
CS 6250: Computer Networking

Lecture 5.1: Router Design Basics

Router Design

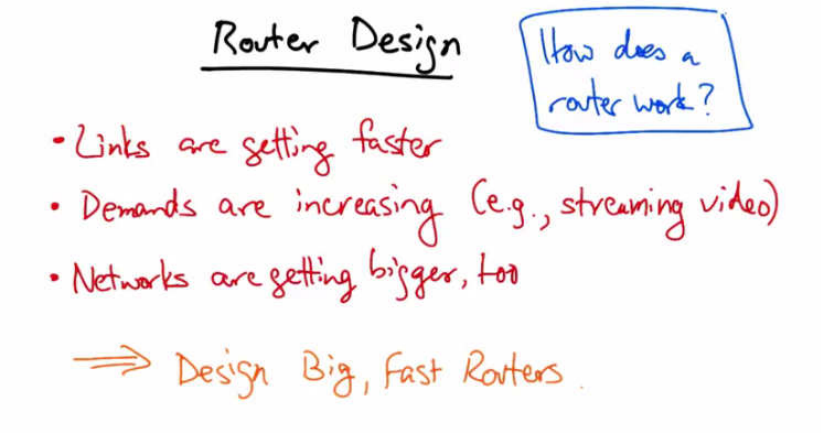


In this lesson, we will cover the design of big, fast, modern routers. Here's a picture of two modern router chassis. On the left, we have a picture of the Cisco CRS-1, and on the right, we have a picture of the Juniper M320. The M320 has for some time been used at the border of the Georgia Tech network between Georgia Tech and the rest of the Internet.



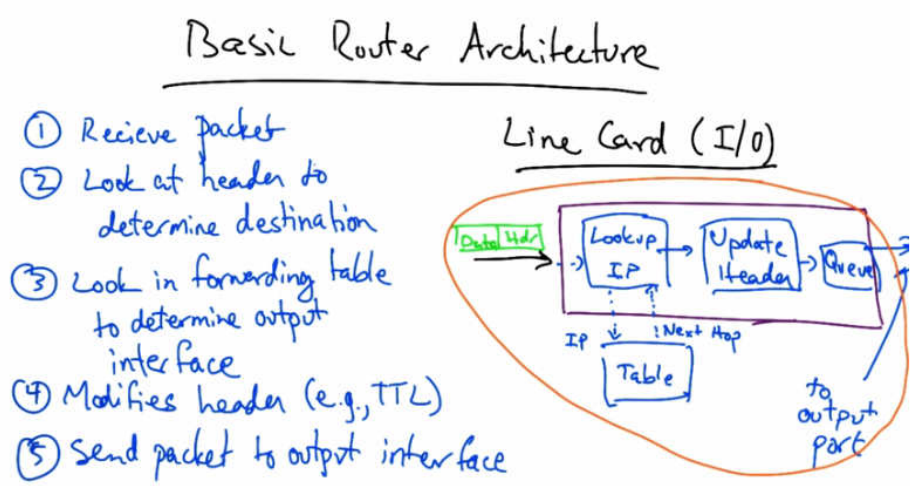
Here's a picture of a couple of line cards that go into these chassis. These kind of look like network interface cards, except the ports are special. Instead of terminating Ethernet, these ports

terminate high capacity fiber links. As you can see, these cards are actually a whole lot bigger than your typical network interface card as well. And, as a result, these chassis are often anywhere from three to six feet tall, and can fill up an entire rack.



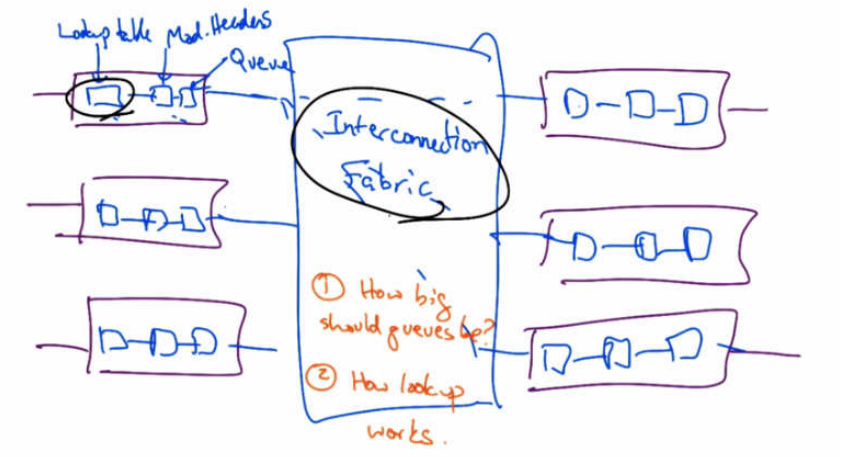
There's a significant need for big, fast routers. Links are getting faster. Traffic demands are also increasing, particularly with the rise of demanding applications, such as streaming video. Networks are getting bigger too, in terms of the number of hosts, routers, and users. So there's a perennial need to design big, fast routers, particularly in Internet backbone networks. The rest of this lesson will focus on how a router works, in particular, how it goes from the process of taking a packet as input and sending it on to where it needs to go. The Internet's routing protocols, of course, are responsible for populating the forwarding tables on a router. But once those tables are populated, the router still has the hard job of taking a packet as input and ultimately getting it to the appropriate output port, so that the traffic can proceed en route to the destination.

Basic Router Architecture



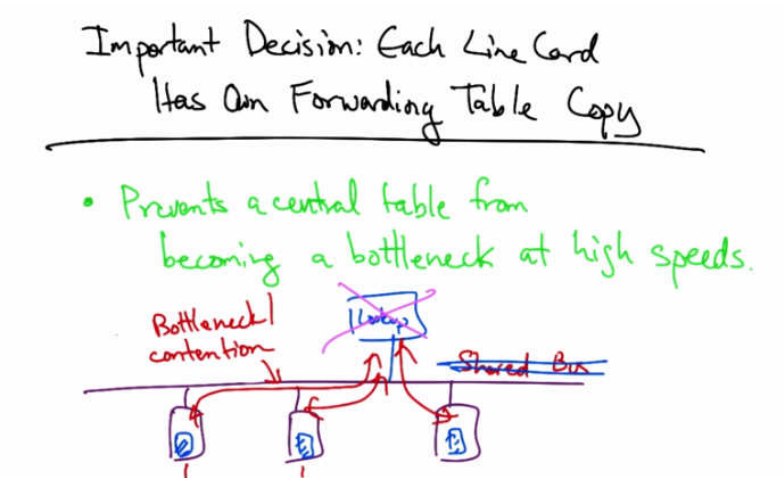
Let's take a look at a generic router architecture. As a summary of basic router function, a router receives a packet. It then looks at the packet header, to determine the packet's destination. It looks in the forwarding table to determine the appropriate output interface for the packet. It

modifies the packet header, such as decrementing the time to live field and updating the IP header check sum appropriately. And finally, it sends the packet to the appropriate output interface. The basic I/O component of a router architecture is the line card, which is the interface by which a router sends and receives data. When a packet arrives, the line card looks at the header to determine the destination, and then it looks in the forwarding table to determine the output interface. It then updates the packet header and finally sends the packet to the appropriate output interface. Now this drawing shows just a single line card. But in fact, when the packet is sent to the output interface, it must traverse the router's interconnection fabric, to be sent to the appropriate output port.



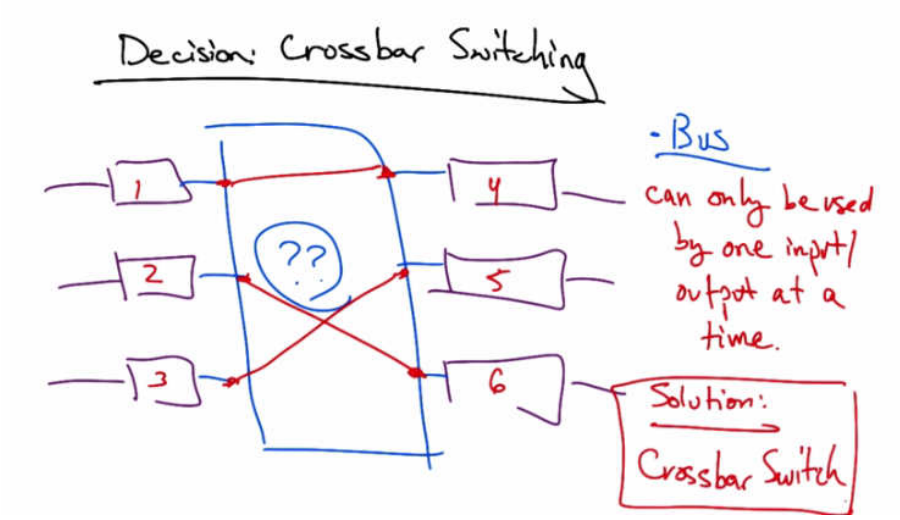
So in fact, we can zoom out from that depiction of a single line card, and what we have is a bunch of line cards that are all connected via an interconnection fabric. Each of these line cards has a lookup table, the capability to modify headers, and a queue, or buffer, for packets as they enter and leave the line card. In other lessons we talk about several important questions such as how big queues should be and how lookup works. In the rest of this lesson, I'll discuss important decisions in router design such as the placement of lookup tables on each line card and the design of the interconnection fabric.

Each Line Card Has Own Forwarding Table Copy



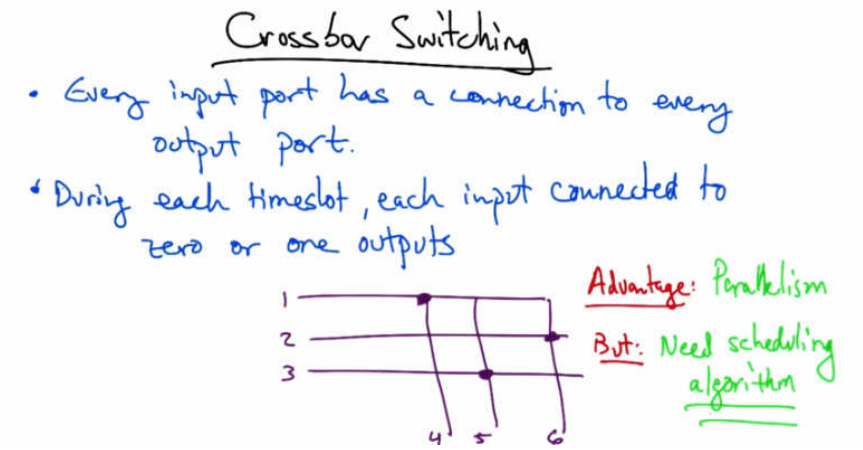
One important decision in the design of the modern routers was to place a copy of the forwarding table on each line card in the router. Well, this introduces some complications in making copies of the forwarding table. Doing so prevents a central table on the router from becoming a bottleneck at high speeds. Consider an alternative where the router has only one copy of the forwarding table. In that case all of the line cards would need to be performing look ups on a central table which involves communication across the back plane as well as many more look ups against a central table. So while distributing the forwarding table across line cards prevents a central table from becoming a bottleneck, early router architectures did not place the look up table on each line card. And as a result, when packets arrived at an individual line card, they would induce a look up in a shared buffer memory which could be accessed over a shared bus. But this shared bus, of course, introduces a bottleneck, as well as contention between the different line cards that may be all performing lookups to the same shared memory. The solution, of course, was thus to remove the shared memory and instead place copies of the forwarding table on each line card. In summary, an important innovation in the design of these router was to eliminate the shared bus and place the look up table on individual line bus.

Decision Crossbar Switching



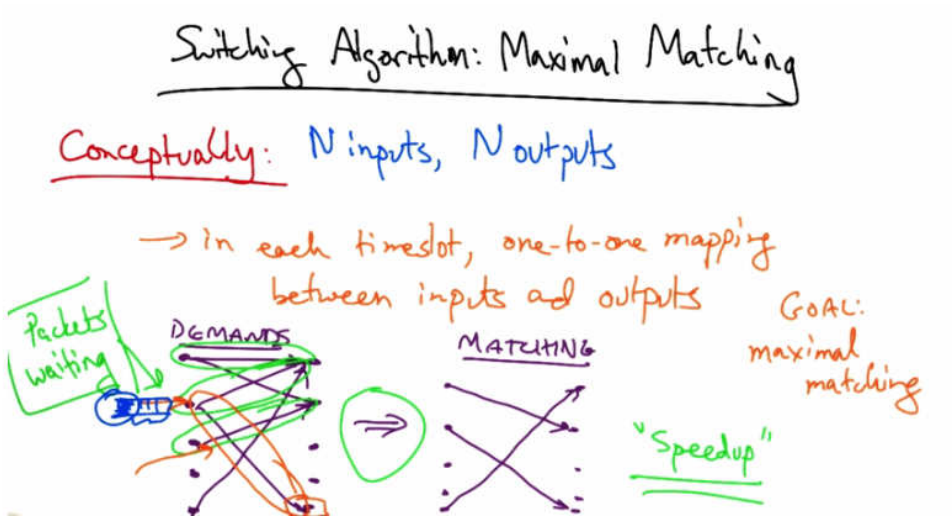
The second important decision is the design of the interconnect, or how the line cards should be connected to one another. Now one possibility is to use a shared bus. But the disadvantage of a bus for the interconnect is that it can only be used by one input-output combination in any single time slot. What we'd like to do is enable input output pairs that don't compete to send traffic from input to output during the same time slot. For example, one should be able to send one to four, two to six and three to five, all in the same time slot. The solution to this problem is to create what's called a crossbar switch, or sometimes is also called a switched backplane.

Crossbar Switching



In crossbar switching every input port has a connection to every output port, and during each time slot, each input is connected to zero or one outputs. The crossbar is often depicted as follows. So if one wants to send to four, we could connect the input to the output in that time slot, and now this row and this column is occupied. But we could connect two to six and three to five in the same time slot without introducing contention. So the advantage of this design is that it can exploit parallelism by allowing multiple packets to be forwarded across the interconnect in parallel. But of course we also need proper scheduling algorithms to ensure fair use of the crossbar switch. Let's take a quick look at what this algorithm needs to achieve.

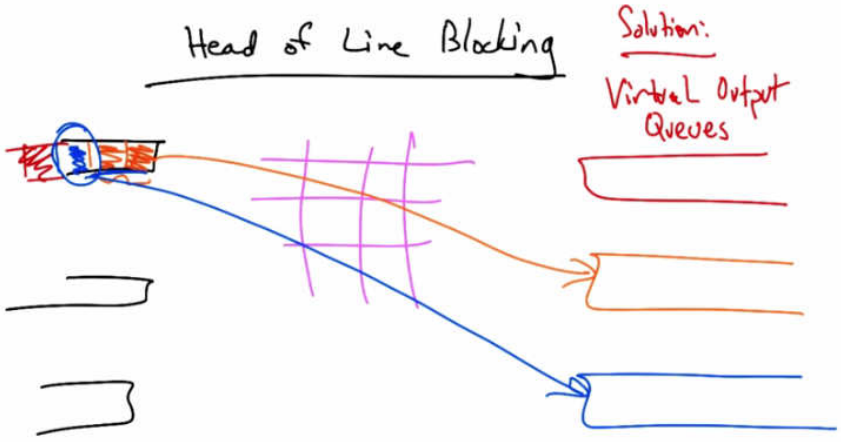
Switching Algorithm Maximal Matching



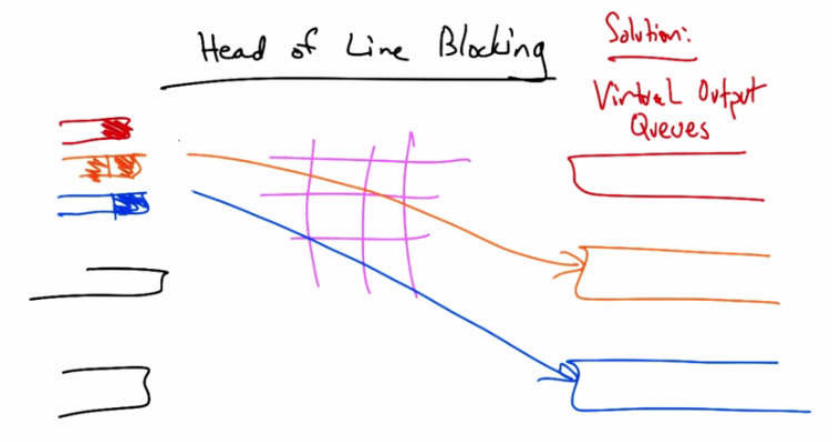
We'd like the cross bar switching algorithm to achieve what's called a maximal matching. Conceptually we have a router with n inputs and n outputs, but of course the inputs are also outputs. It's just easier to think about the inputs and the outputs being separate when we talk about the switching problem. Now in each time slot we would like to achieve a one-to-one

mapping between inputs and outputs, which is a matching. And our goal is that the matching should be maximal. So in a particular time slot, we might have a certain set of traffic demands, or traffic at certain input ports, that is destined for certain output ports. And our goal is, given these demands to produce a matching that is maximal and fair. Now, given demands for a particular time slot and the resulting matching, notice that certain demands were not satisfied. These packets that arrived at inputs must wait until the next time slot to be forwarded to the appropriate output port, because they couldn't be matched in the same time slot as those shown here. Remember that there must be exactly a one-to-one matching between any inputs and outputs in a particular time slot. Most router crossbars have a notion of speedup whereby multiple matchings can be performed in the same time slot. So, for example, if the line cards are running at, say, ten gigabits per second, then running the interconnect twice as fast would allow matchings to occur twice as fast, as packets would arrive on the inputs or be forwarded from the outputs. It is thus common practice to run the interconnect at a higher speed than the input and output ports. Just speeding up the interconnect does not solve all problems. Note, for example, that in this set of demands we have packets arriving at this input port destined for this output port, but if there's only a single queue at this input, the packets that are destined for the output port circled in orange, might actually be blocked behind a set of packets that are destined for other output ports. So even if we could induce a speed up at the interconnect, certain packets may be blocked in the queue by packets ahead of them destined for other output ports.

Head of Line Blocking

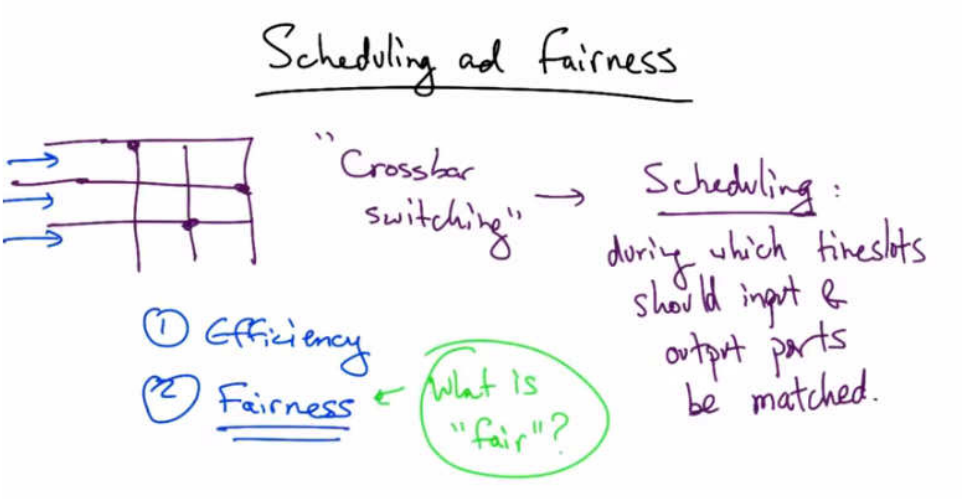


For example, if we have packets arriving in this queue destined for the orange queue, at the front of the queue, then even with the speed up, there may be packets that are sufficiently far behind in the queue that they're waiting behind the orange packets. What we'd like to be able to do is perform matchings to allow these packets to be sent to the output ports, and not have to wait for the entire queue to be drained of packets destined for the orange output port.



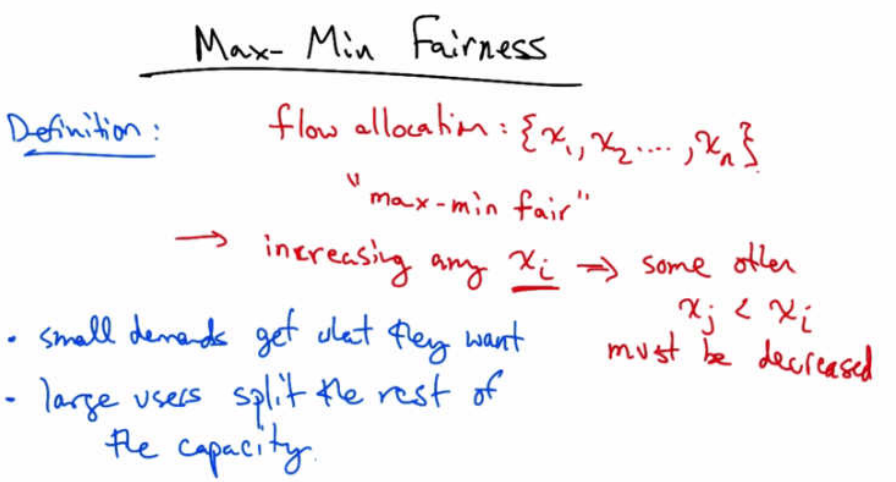
A solution is to create virtual output queues, where instead of having a single queue at the input, we have one queue per output port. This prevents packets at the front of the queue that are destined for a particular output port from blocking packets that could otherwise be matched to other output queues in earlier timeslots.

Scheduling and Fairness



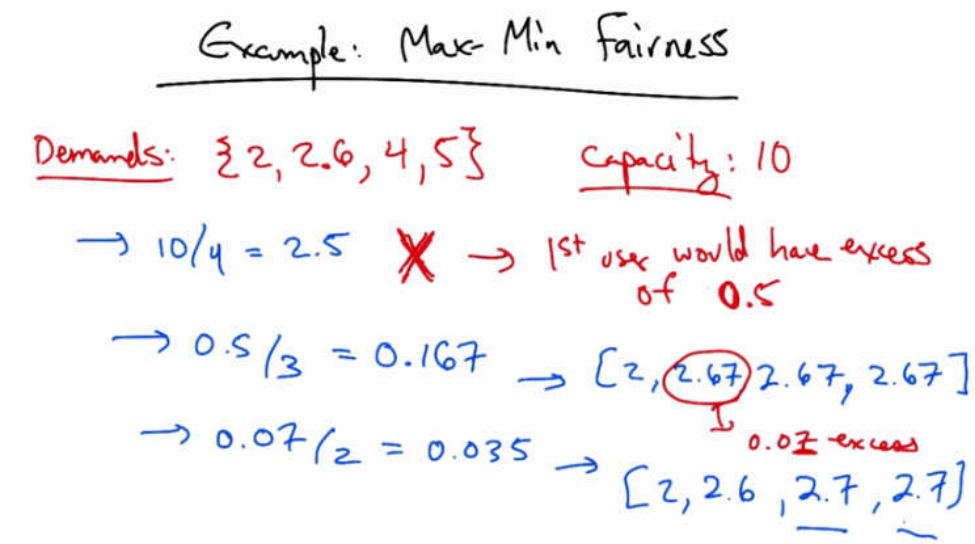
Let's now talk about scheduling and fairness. And when we talk about, in a crossbar switch, the process of matching input ports to output ports, the decision about which ports should be matched in any particular time slot is a process called scheduling. There are two important goals in scheduling. One is efficiency, which is to say that if there is traffic at inputs destined for output ports, the crossbar switch should schedule inputs and outputs so that traffic isn't sitting idle at the input ports if some traffic could be sent to the available output ports. Another consideration in scheduling is fairness, which is to say that given demands at the inputs, we want to make sure that each queue at the input is scheduled fairly for some definition of fairness. Now, defining fairness is tricky. And there are multiple possible definitions of fairness. Here, we'll look at an important fairness definition called max min fairness.

Max Min Fairness



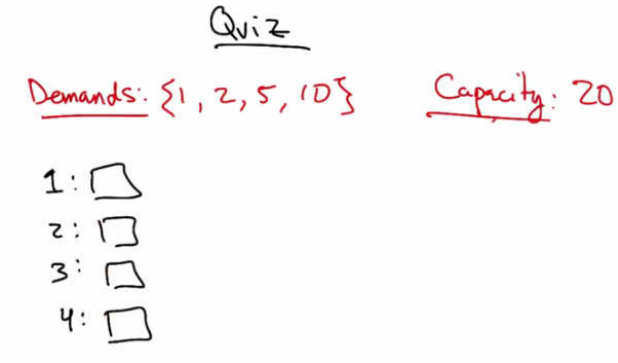
Now, to define max-min fairness, let's first assume that we have some allocation of rates across flows x_i. Now, we say that this allocation is max-min fair if increasing any rate x_i implies that some other x_j that is smaller than x_i must be decreased to accommodate for the increase in x_i. So in other words, the allocation is max-min fair if we can't make any one of these flow rates better off without making some flow rate worse off, that's already worse than the flow rate x_i. So the upshot results in small demands getting exactly what they asked for, and the larger demands splitting the remaining capacity among themselves equally. More formally, we perform this procedure as follows. We allocate resources to users in order of increasing demand. No user receives more than what they requested. And users that still have unsatisfied demands, split the remaining resources.

Max Min Fairness Example



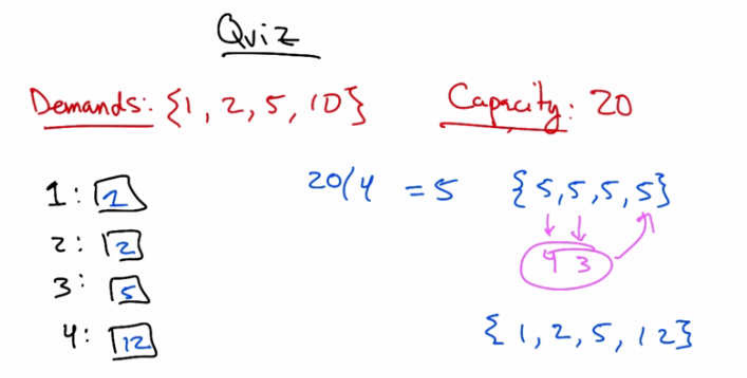
Let's consider an example for max-min fair allocation. Let's suppose that we have a link with capacity ten and four demands: 2, 2.6, 4, and 5. Now, obviously the demands exceed capacity. So we need to figure out a way of allocating rates to each of these demands that is max-min fair. First, note that 10 divided by 4 is 2.5. But this is not a good solution, because the first user only needs 2. So the first user would have an excess of .5 under this allocation. So what we want to do is take this excess of .5 and divide it among the remaining 3 users, whose demands have not yet been fulfilled. This would yield an allocation of 2, 2.67, 2.67, and 2.67. But now user two has an excess of 0.07, so we take that excess, divide it among the remaining 2, and that gives us our final max-min fair allocation. Note that this is called max-min fairness, because it maximizes the minimum share to each user whose demand is not fully serviced. In this case, the 3rd and 4th users.

Max Min Fairness Quiz



As a quick quiz, let's try doing a max min fair allocation. Suppose that we have demands of one, two, five, and ten, and link, whose rate is 20. Please give the max-min fair allocation across these four users.

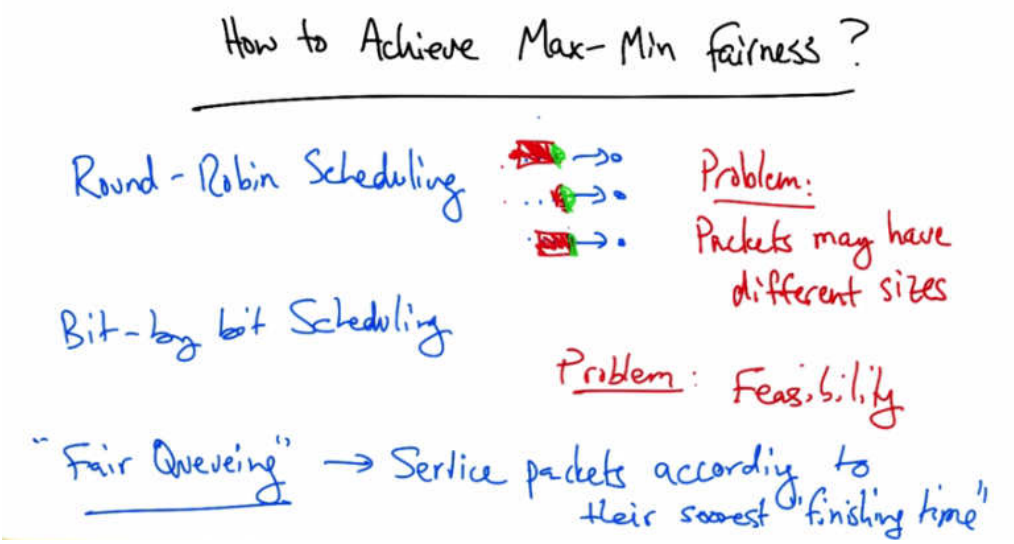
Max Min Fairness Solution



To compute the max min fair allocation, we take 20 and we divide it by 4, which yields 5. But the first user only needs one, which yields an excess of four. The second user only needs two, which yields an excess of three. So in this case the max min fair allocation is easy. We simply

take this excess of seven and give it to the only user whose demand is not yet satisfied, resulting in the max min fair allocation of one, two, five, and twelve.

How to Achieve Max Min Fairness

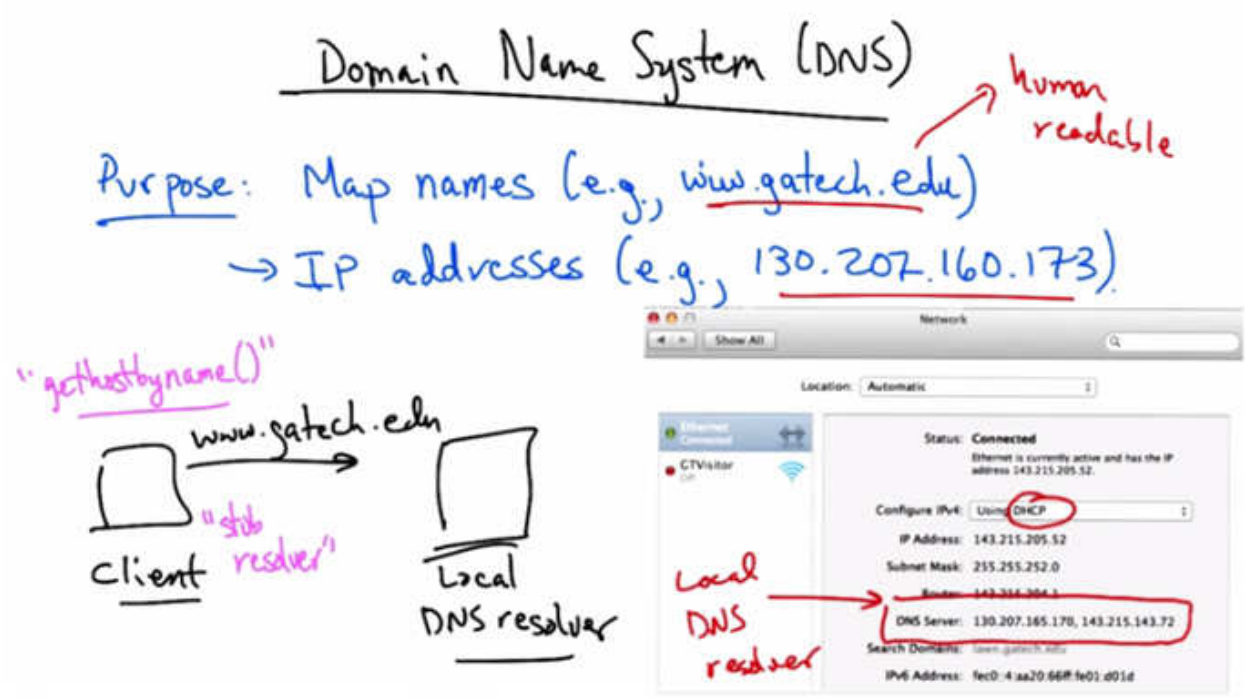


Now, how do we achieve max-min fairness? One approach is via round robin scheduling, where given a set of cues, the router simply services them in order. The problem here is that packets may have different sizes. So if the first queue had a huge packet, and the second queue had a little packet, and the third queue had a medium sized packet, then servicing these queues in order obviously isn't fair. Because the first queue would effectively get more of its fair share, because its packet just happened to be bigger. An alternative is to use bit by bit scheduling, where during each time slot, each queue only has one bit serviced. This, of course, is perfectly fair, but the problem is feasibility. How do we service one bit from a queue? A third alternative is called Fair Queuing, which achieves max-min fairness by servicing packets according to the soonest finishing time. A Fair Queuing algorithm computes the virtual finishing time of all candidate packets, which are the packets at the head of all non-empty flow queues. Based on these virtual finishing times, Fair Queuing compares the finishing times of each queue and services the queue with the minimum finishing time. So the queue whose packet has the minimum virtual finishing time is serviced.

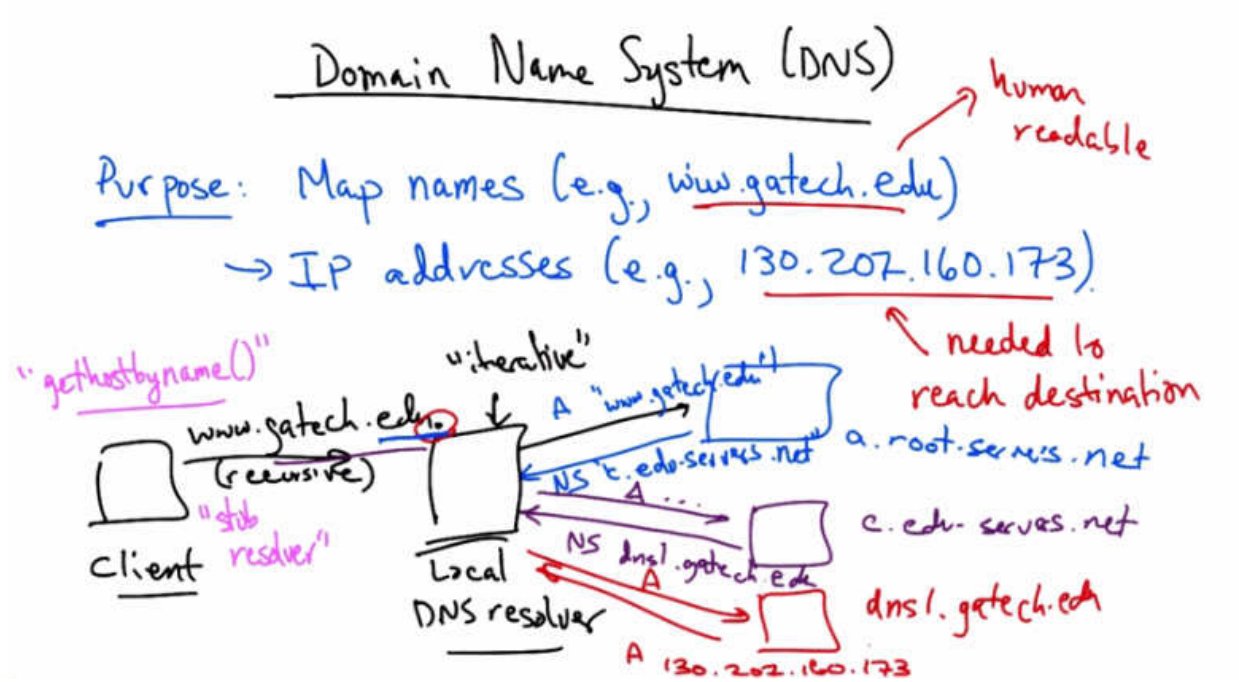
CS 6250: Computer Networking

Lecture 5.2: DNS

Domain Name System Part 1

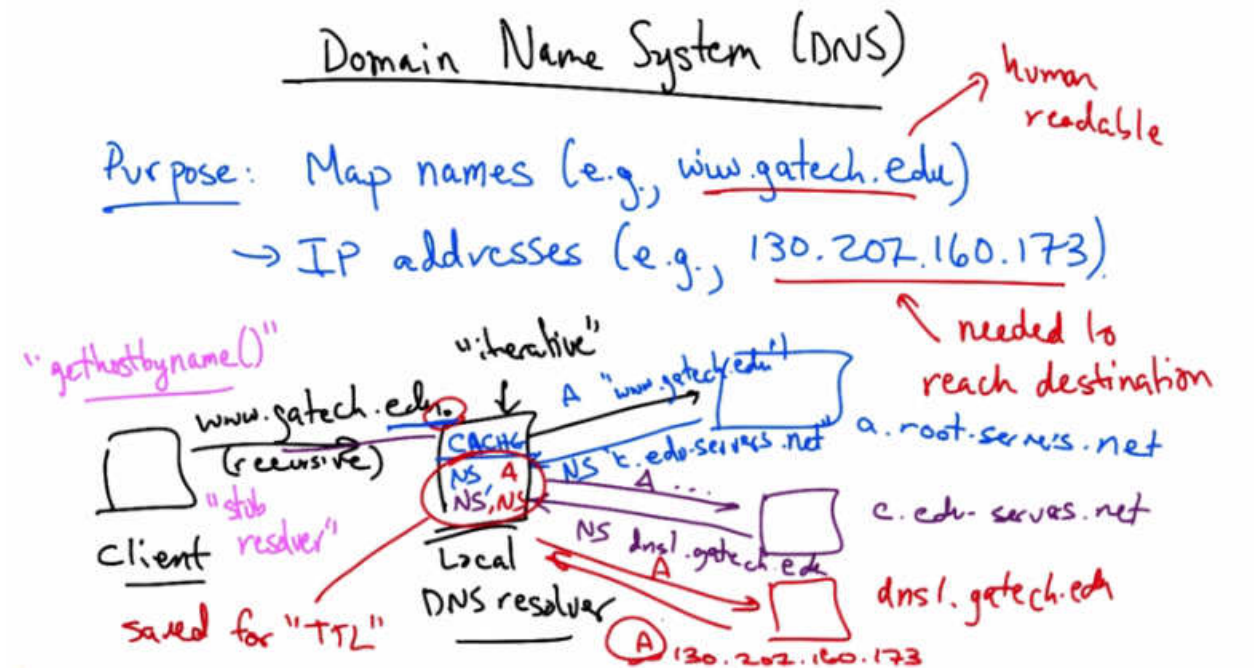


We'll now have a look at the domain name system or DNS. The purpose of the domain name system is to map human readable names such www.gatech.edu to IP addresses such as 130.207.160.173. A name such as this is human readable and much easier to remember and type than an IP address. But in fact, the IP address is what's needed to send traffic to the intended destination. So, we need a lookup mechanism that takes a human readable name and maps it to an IP address. The system that does this is a Domain Name System, or the DNS. The system roughly works as follows. A client may want to look up a domain name such as www.gatech.edu. A networked application source code might do so by invoking a function such as get_host_by_name, which takes as an argument a domain name and returns an IP address. The client typically has what's called a stub resolver, and that stub resolver takes that name and issues a query. The stub resolver might have cached the answer or the IP address corresponding to this name, but if not, the query is sent to what's called a local DNS resolver.



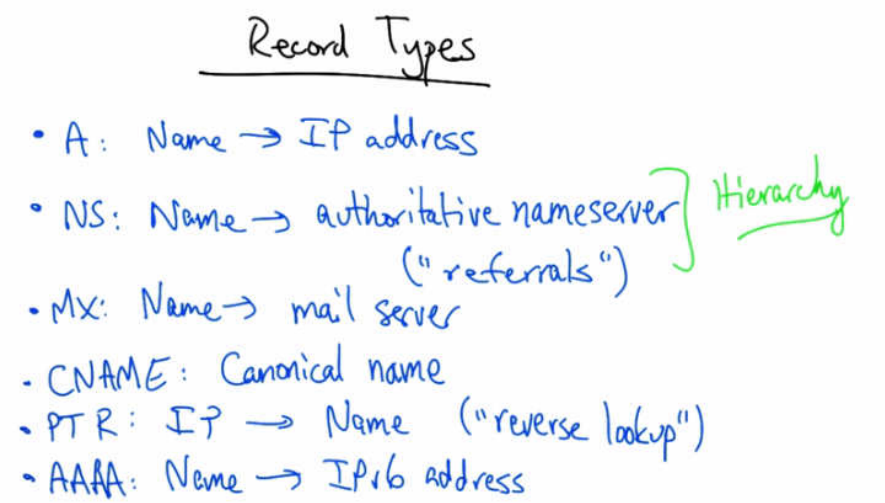
Your local DNS resolver, is typically configured automatically when your host is assigned an IP address using a protocol called the domain host control protocol or DHCP. In your host configuration such as this one, you can see that this local host has two local DNS resolvers. Typically, a client will try the first DNS resolver and if it doesn't receive a response within a preconfigured timeout, it will try sending the same query to the second local DNS resolver as a backup. This query is typically issued recursively, meaning the client does not want intermediate referrals sent back to it. It only wants to hear when it's received the final answer. The local resolver on the other hand will perform iterative queries. It might have the answer to this particular query in the cache, in which case it would simply reply with the answer. But let's suppose for the moment, that nothing is cached. Each fully qualified domain name is presumed to end with a dot, indicating the root of the DNS hierarchy. Now the IP addresses for the root servers, or those that are authoritative for the root, may already be configured in the local DNS resolver. In this case, the resolver may be able to query for the authoritative server for .edu, say a.rootservers. net. This would be an A record query. The answer might return with what's called an NS record, which is a referral. In this case the answer might be a referral to the edu servers. Now the local resolver issues the same query to the edu servers and receives a referral to the authoritative servers for gatech.edu. Finally the local resolver might query the authoritative name server for gatech.edu and actually receive an A record indicating the actual IP address that corresponds to that name.

Domain Name System Part 2



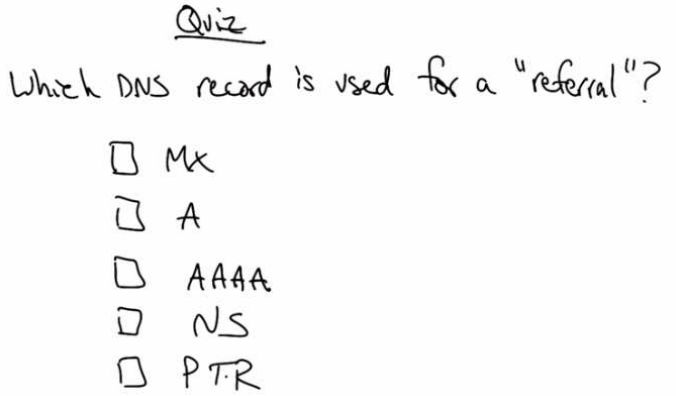
Now this process of referrals, as you can see, can be rather slow. A particular DNS query might thus require round trips to multiple servers that are authoritative for different parts of the hierarchy. The blue server is authoritative for the root. The purple server is authoritative for .edu and the red server is authoritative for gatech.edu. Now supposing we wanted to save the extra time in trouble of these round trip times. This local resolver would typically have a cache that stores the NS records for each level of the hierarchy as well as the A records. And each of these answers would be stored or cached for a particular amount a time. Each one these replies has what's called a time to live or a TTL that indicates how long each of these answers can be saved before they need to be looked up again. Caching allows for quick responses from the local DNS resolver, especially for repeated mappings. For example, since everyone is probably looking up domain names such as google.com it's much faster to keep the answer in cache. So, given multiple clients trying to resolve the same domain name, the answers can all be resolved in a local cache. Some queries can reuse parts of this look up. For example, it's unlikely that the authoritative name server for the root is going to change very often. So that answer might be kept, or cached, for a much longer period of time. A typical time might be hours or days, or even weeks. The mapping for a local name, such as www.gatech.edu, on the other hand, might change more frequently and thus these local TTL's might need to be smaller. Now the most common type of DNS record is what's called an A record, which maps an IP address to a domain name. But there are other important record types as well.

Record Types



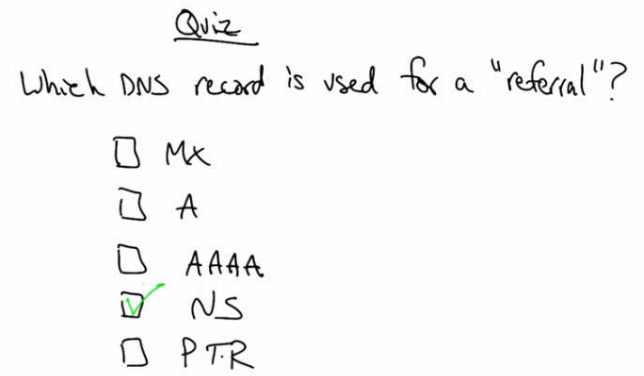
A records map names to IP addresses as we have seen. We have also seen what's called an NS or a Nameserver record which maps a domain name to the authoritative nameserver for that domain. So we saw a bunch of NS records in the form of referrals, whereby, if we ask the route for a mapping of gatech.edu to an IP address, it doesn't specifically know the answer, but it can issue a nameserver reply or an NS record referring the resolver to a different nameserver that could be responsible for that part of the domain name space. This allows the domain name system to be implemented as a hierarchy. Another important DNS record type is an MX record, which shows the mail server for a particular domain. Occasionally, one name actually is just an alias for another name. For example, www.gatech.edu actually has a slightly different real name. The CNAME is basically a pointer from an alias to another domain name that needs to be looked up. The PTR is another record that we'll look at, and this maps IP addresses to domain names. For example if you wanted to know the name for a particular IP address, you need to issue a PTR query. This is sometimes called a reverse lookup. Finally, a AAAA record maps a domain name to an IPV6 address. Let's take a look at a couple of different examples of domain name lookups using a command line utility called dig.

DNS Quiz



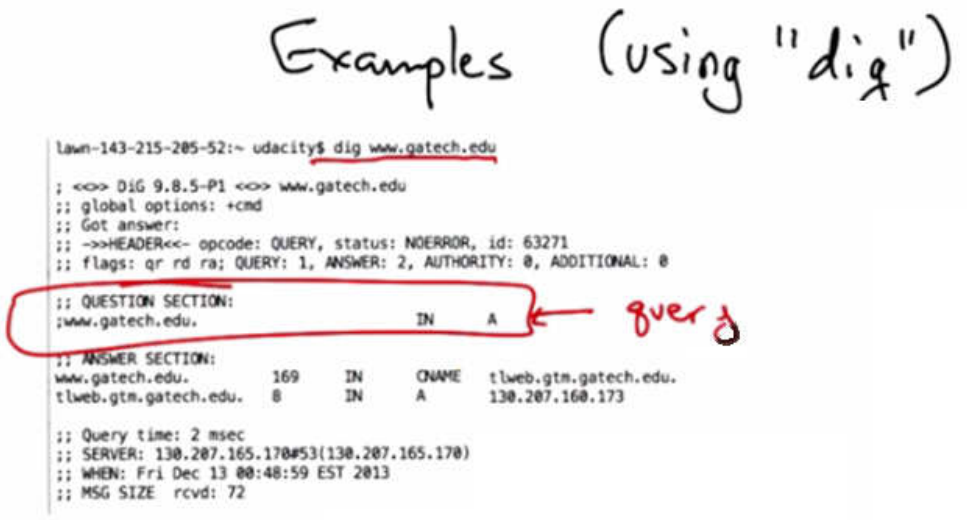
As a quick quiz, which DNS record is used for referral? Is it the MX record? A record? The Quad A record? The NS record? or the PTR?

DNS Solution

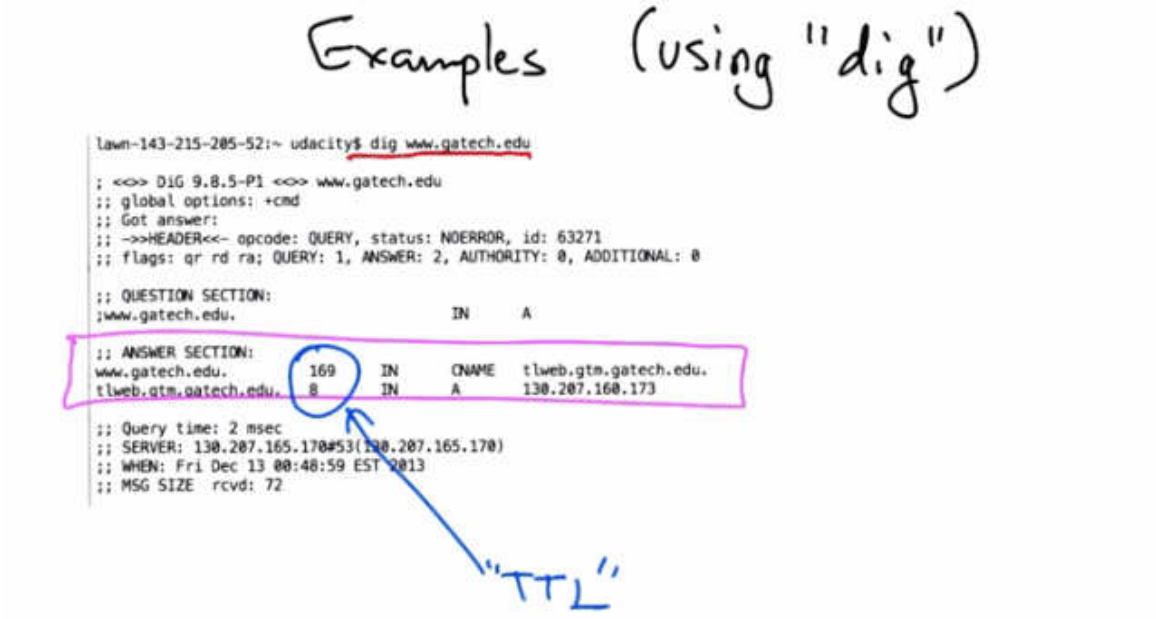


The NS record indicates the authoritative name server for a particular portion of the domain name space, and an NS record reply is often referred to as a referral. MX records indicate mail servers. A records are IP addresses for the domain name. AAAA are for IPv6 addresses, and a PTR is a name corresponding to an IP address being queried.

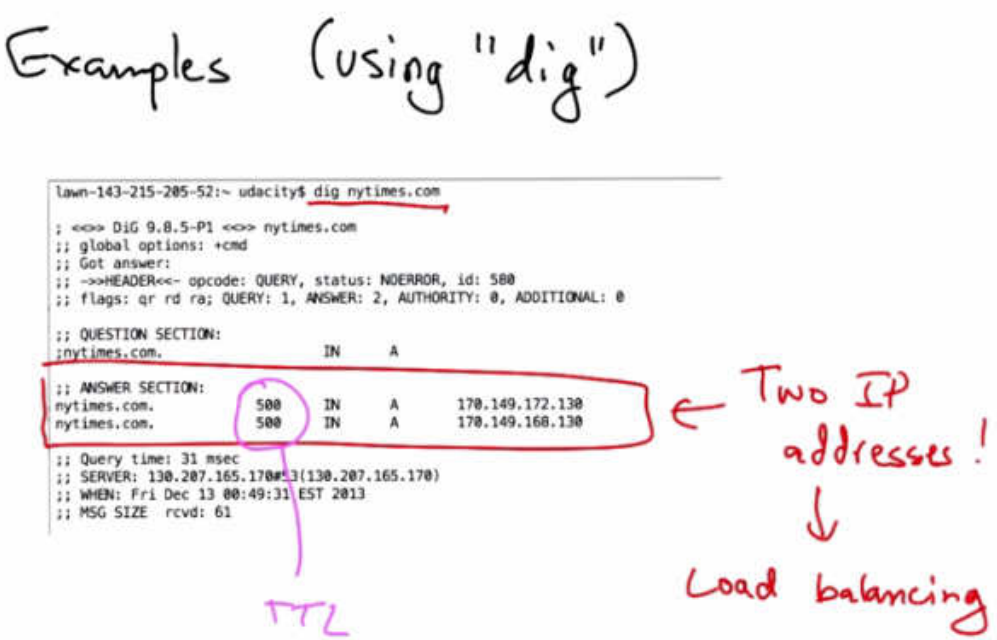
Examples (using dig) Part 1



Here's an example of a lookup for an A record for gatech.edu. You can try this at your own command line by typing, for example, dig www.gatech.edu. Now there are some interesting things to note in this trace. Here is our query and you can see that this is an A record query.

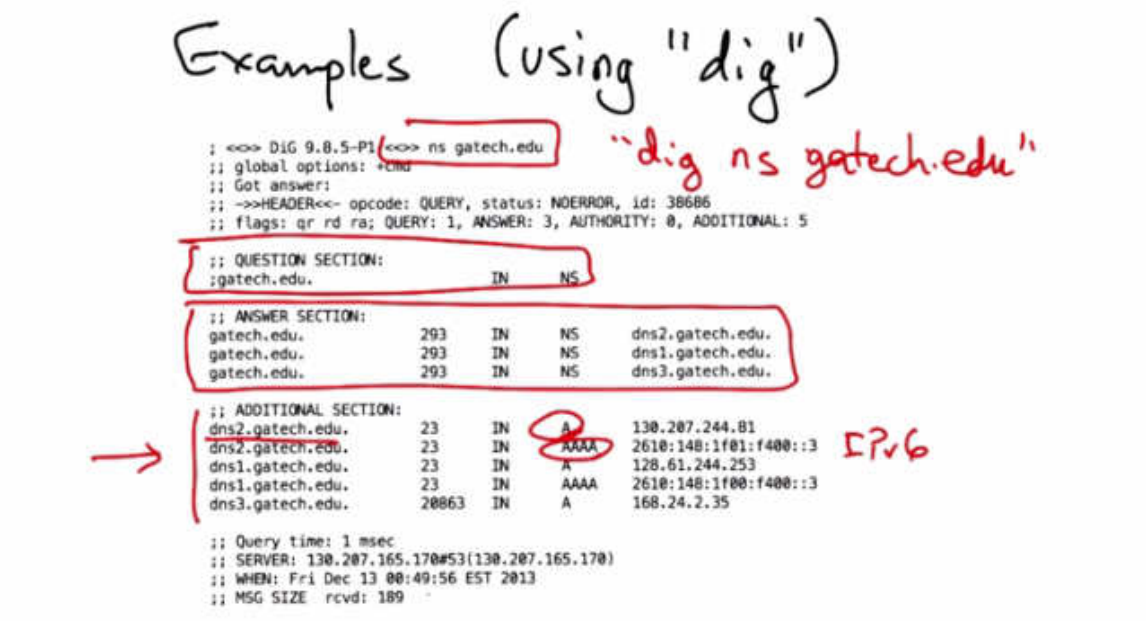


Here's our answer. You can see that the answer actually has a CNAME in it, which basically says, well you asked for gatech.edu but in fact what you really want to ask for is tlweb.gtm.gatech.edu. So then we issue an A record query for that name, and we ultimately get the IP address. These numbers here indicate the time to live or the amount of time in seconds that the entry can be stored in the cache.

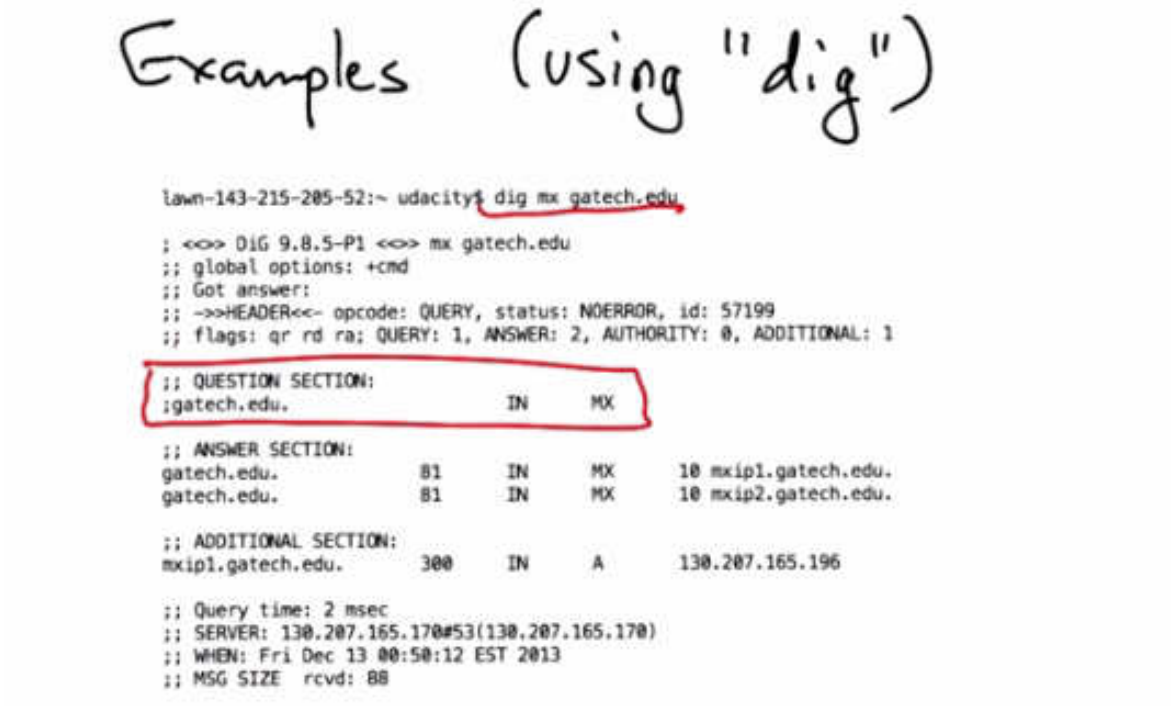


Here's another example of a DNS lookup from nytimes.com. The interesting thing to note here is that in response to the A record query, we see two IP addresses. This is typically performed when a service wants to perform load balancing So, the client could use either one of these. It might prefer the first one, but if we issued the same query again, we might actually get these IP

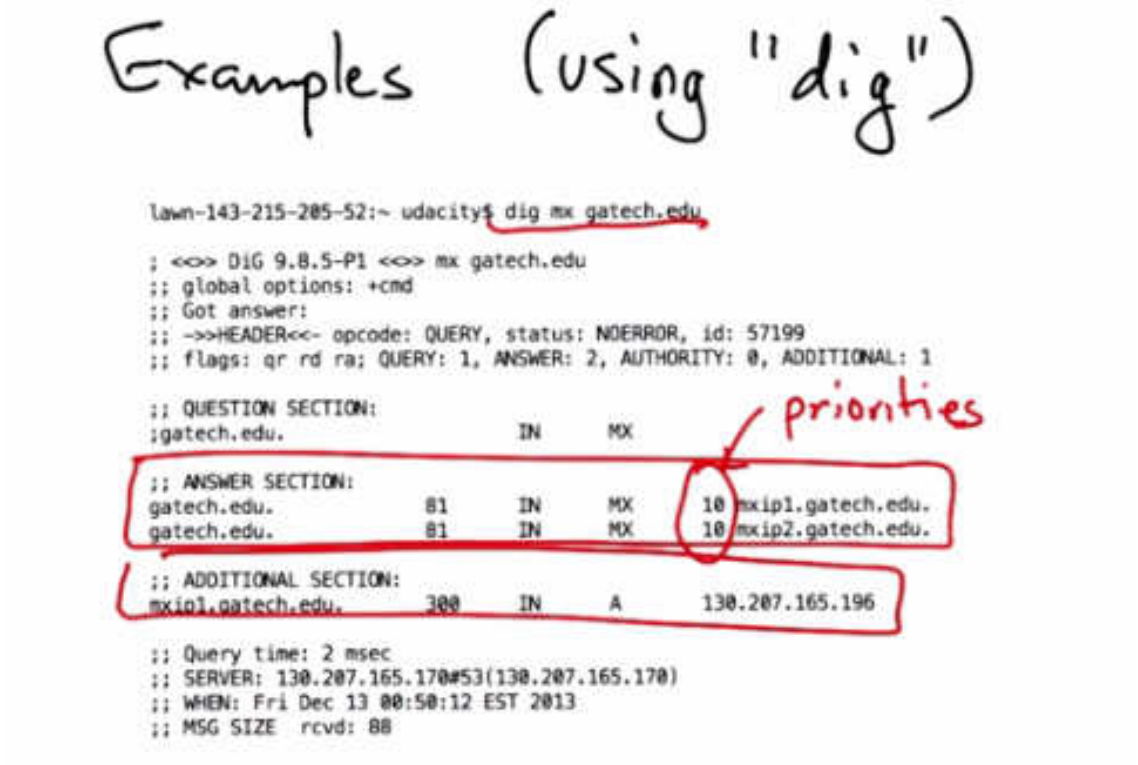
addresses in a different order. Now, again, here you can see the TTL value which indicates how long these A records can be stored in cache. In a subsequent example, we'll look at other query types that have much longer TTL values.



Here's an example of a query for the NS record for gatech.edu. You can see this output by typing dig ns gatech.edu. You can see here in the question section, now instead of an A record query we have an NS record query. And our answer is a bunch of NS records that are dns1, 2, and 3.gatech.edu, any of which could answer authoritatively for sub-domains of gatech.edu. You can see that in addition to the answer, which return the name servers, we also need the IP addresses of those name servers, which is returned in the additional section of the answer. You can see here that we received not only A records for each domain name but also quad A or IPv6 addresses corresponding to each authoritative name server.



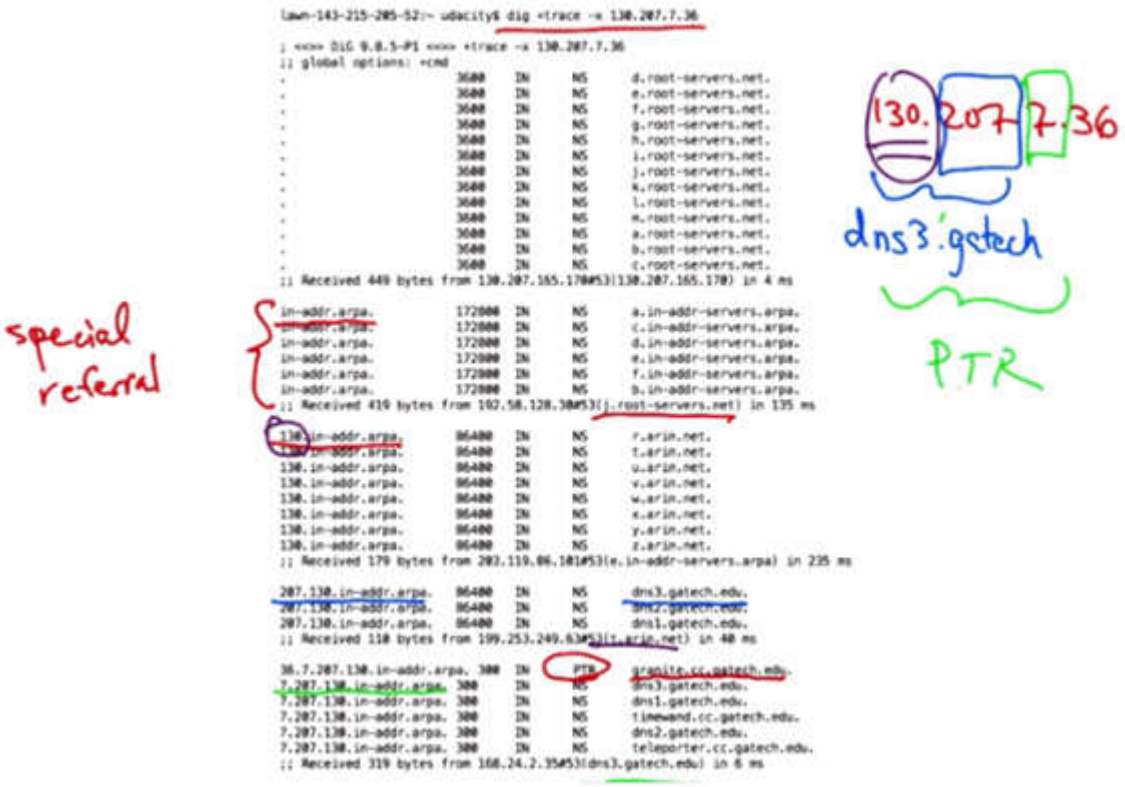
Here's an example of a query for an MX record or the mail server corresponding to gatech.edu. Now, here again you can see the question is the MX record and you can see the answer which returns two mail servers as well as the additional section, which returns an A record indicating the IP address corresponding to the mail server that was returned in the MX record. In addition to the TTL, we also have some metrics that indicate priorities that would allow a system administrator to configure a primary and a backup mail server.



In this case, the mail servers, just happen to have the same priority level.

Examples (using dig) Part 2

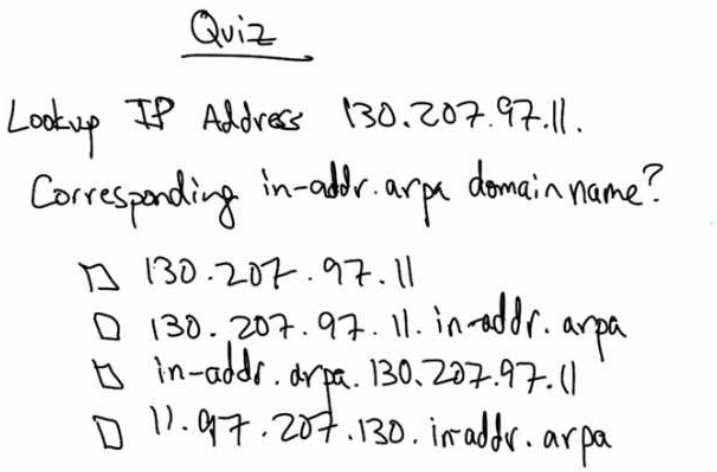
Let's put everything together now by looking at a trace of an entire lookup. Now in the examples before, we didn't get to see the full lookup hierarchy because we issued a recursive query. But let's suppose we wanted to see every step of the DNS lookup process. You can do this by using the trace option in dig. Here you can see exactly what we saw before, which is the local resolver. In this case, issuing a query to a local resolver and receiving a referral to an authoritative server for dot which could be any of the following. That query, elicits an answer for the .edu servers which subsequently issues a referral to the servers that are authoritative for gatech, which ultimately reply with the appropriate a records as well as the authoritative nameservers for gatech.edu.



A final interesting example explores how to map an IP address. back to a name. In this case, we're ultimately looking for PTR record, which is the name corresponding to this IP address. But first, what happens is we receive a special referral. When we ask the root servers about this particular IP address, instead of being referred to a particular .com or .edu domain, we're referred to a special top level domain called inaddr.arpa, which maintains referrals to authoritative servers that are maintained by the respective internet routing registries, such as ARIN, RIPE, APNIC and so forth. So here we see a referral to inaddr.arpa. Subsequently, we see a referral to 130.in-addr.arpa corresponding to the first octet of the IP address. Next when we ask ARIN about 130.in-addr.arpa we receive another referral, which is to is to 207.130.in-addr.arpa. And because 130.207 is allocated to gatech.edu, ARIN knows that the appropriate referral for this part of the address space is to DNS 1, 2, or 3.gatech.edu. Next we issue a query for the next part of the octet. 7.207.130.in-addr.arpa corresponding to the first 3 octets. And now we actually get the PTR because DNS3.gatech.edu knows the reverse mapping between 130.207.7.36 and the name for that IP address. So you can see that the PTR records, or those that map IP addresses to names,

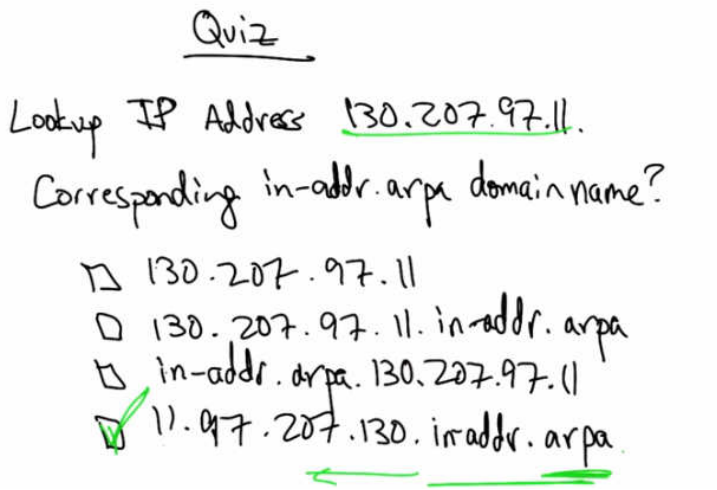
are resolved through a special hierarchy through in-addr.arpa at the root followed by a walk through the regional registries and ultimately, to the domains, such as gatech, that are responsible for particular regions of the IP address space.

Lookup IP Address Quiz



As a quick quiz, suppose we wanted to look up the IP address 130.207.97.11. What is the corresponding in-addr.arpa domain name? Is it 130.207.97.11? Is it 130.207.97.11.in-addr.arpa? Is it in-addr.arpa.130.207.97.11? Or is it 11.97.207.130.in-addr.arpa?

Lookup IP Address Solution



The corresponding domain name for the PTR lookup for this IP address is the record corresponding to 11.97.207.130.in-addr.arpa. Notice that the reversal of the octets in this name corresponds to a strict traversal of the hierarchy from the highest levels of the hierarchy at inaddr.arpa to the lower levels, as the IP address moves from higher to lower parts of the hierarchy.

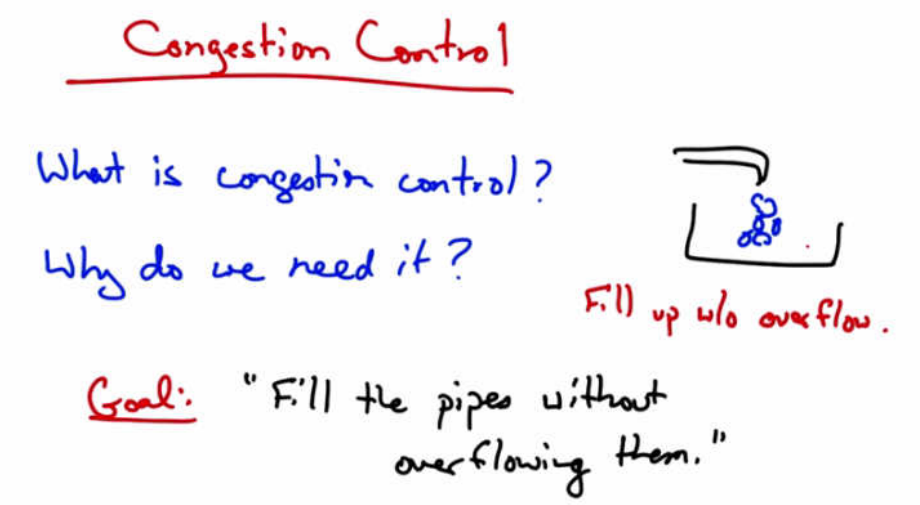
CS 6250: Computer Networking

Lecture 6: Congestion Control, & Streaming

Lesson 1 Intro

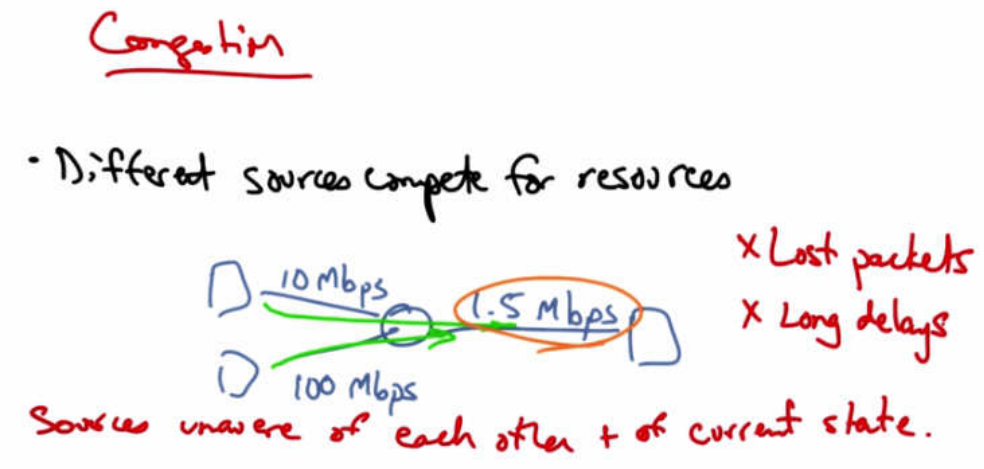
In this section of the course we'll learn about resource control and content distribution. Resource control deals with handling bandwidth constraints on links. And in this first section, we'll focus on controlling congestion along links. To learn more about TCPA and congestion control, your Mininet project will explore what happens when congestion control goes wrong.

Congestion Control



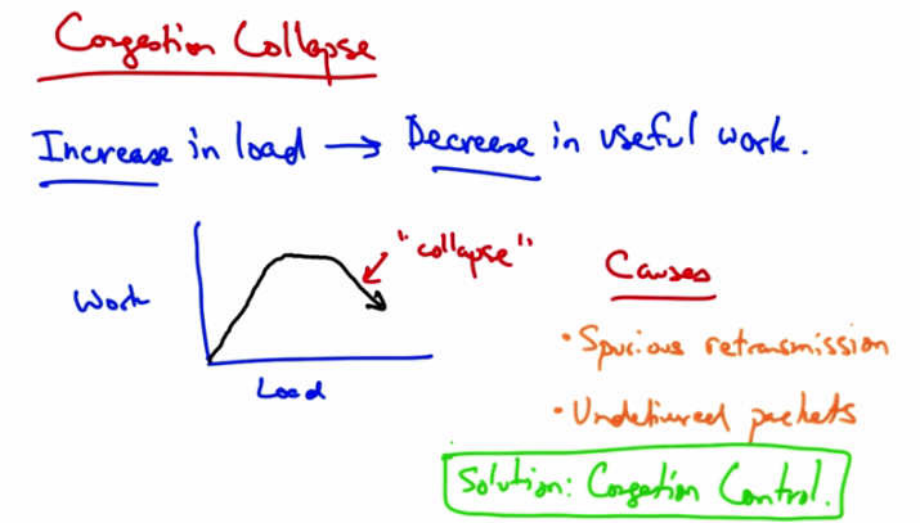
Okay, we're starting course two on congestion control and streaming. And we'll first talk about congestion control. In particular, what is congestion control and why do we need it? Simply put, the goal of congestion control is to fill the Internet's pipes without overflowing them. So to think about this in terms of an analogy, suppose you have a sink, and you're filling that sink with water. Well, how should you control the faucet? Too fast and the sink overflows. Too slow and you are not efficiently filling up your sink. So what you would like to do is to fill the bucket as quickly as possible without overflowing. The solution here is to watch the sink. And as the sink begins to overflow, we want to slow down how fast we're filling it. That's effectively how congest control works.

Congestion



Let's suppose that in a network we have three hosts shown as squares at the edge of the network that are connected by links with capacities as shown. The two senders on the left can send at rates of 10 megabits per second and 100 megabits per second, respectively. But the link to the host on the right is only 1.5 megabits per second. So these different hosts on the left are actually competing for the same resources inside the network. So the sources are unaware of each other and also of the current state of whatever resource they are trying to share, in this case, how much other traffic is on the network. This shows up as lost packets or long delays and can result in throughput that's less than the bottleneck link, something that's also known as congestion collapse.

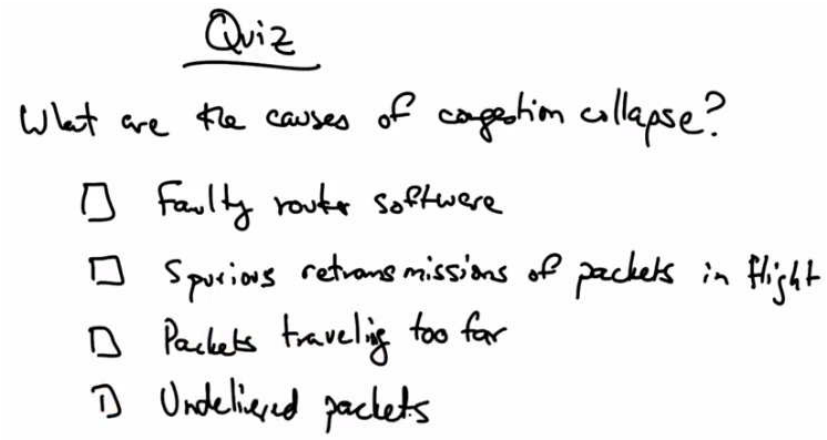
Congestion Collapse



In congestion collapse, an increase in traffic load suddenly results in a decrease of useful work done. As we can see here, up to a point, as we increase network load, there is an increase in

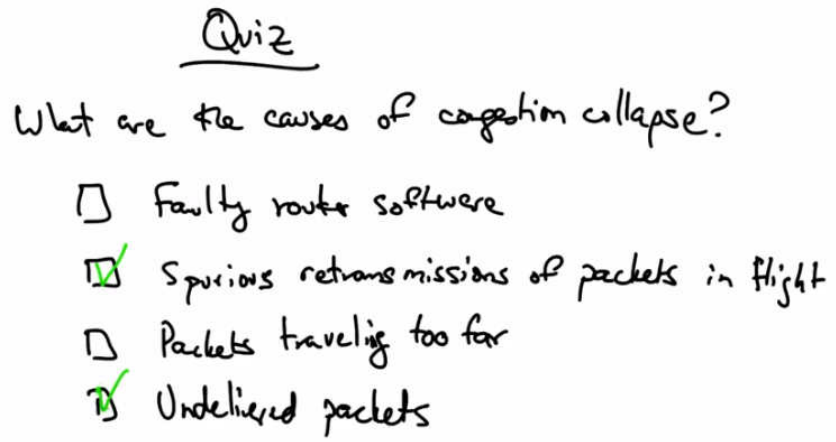
useful work done. At some point, the network reaches saturation, at which point increasing the load no longer results in useful work getting done. But at some point, actually increasing the traffic load can cause the amount of work done or the amount of traffic forwarded to actually decrease. There are many possible causes. One possible cause is the spurious re-transmissions of packets that are still in flight. So when senders don't receive acknowledgements for packets in a timely fashion, they can spuriously re-transmit, thus resulting in many copies of the same packets being outstanding in the network at any one time. Another cause of congestion collapse is simply undelivered packets, where packets consume resources and are dropped elsewhere in the network. The solution to spurious re-transmissions is to have better timers and to use TCP congestion control, which we'll talk about next. The solution to undelivered packets is to apply congestion control to all traffic. Congestion control is the topic of the rest of this lesson.

Congestion Collapse Quiz



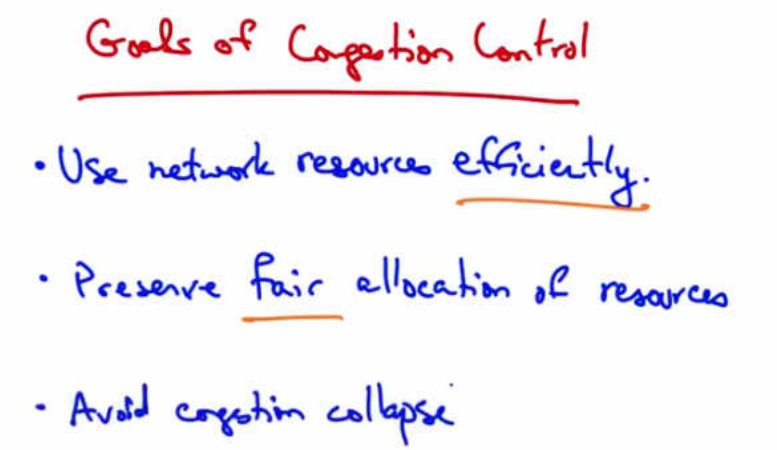
Let's take a quick quiz. What are some possible causes of congestion collapse? Faulty router software? Spurious retransmissions of packets in flight? Packets that are travelling distances that are too far between routers? Or, undelivered packets? Please check all options that apply.

Congestion Collapse Solution



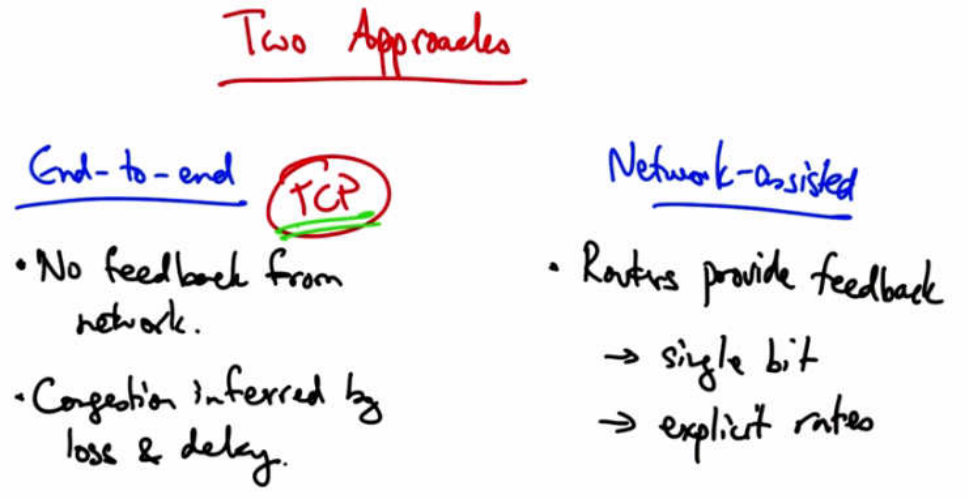
Congestion collapse is caused by spurious retransmissions of packets in flight and undelivered packets that consume resources in the network but achieve no useful work.

Goals of Congestion Control



Congestion control has two main goals. The first is to use network resources efficiently. Going back to our sink analogy, we'd like to fill the sink as quickly as possible. Fairness, on the other hand, ensures that all the senders essentially get their fair share of resources. A final goal, of course, is to avoid congestion collapse. Congestion collapse isn't just a theory, it's actually been frequently observed in many different networks.

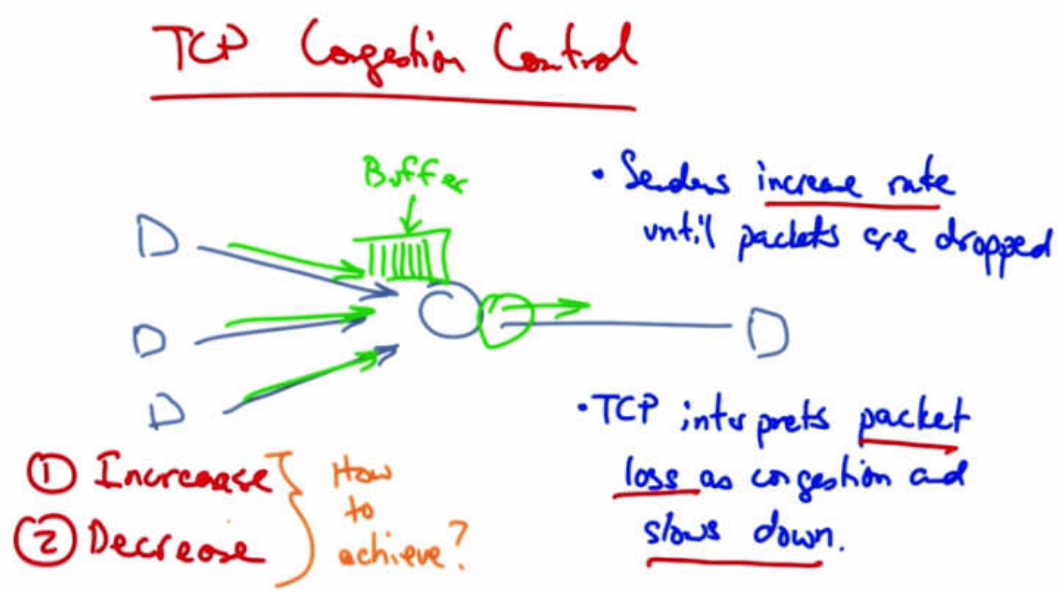
Two Approaches to Congestion Control



There are two basic approaches to congestion control: end-to-end congestion control and network assisted congested control. In end-to-end congestion control the network provides no explicit feedback to the senders about when they should slow down their rates. Instead, congestion is inferred typically by packet loss, but potentially, also by increased delay. This is

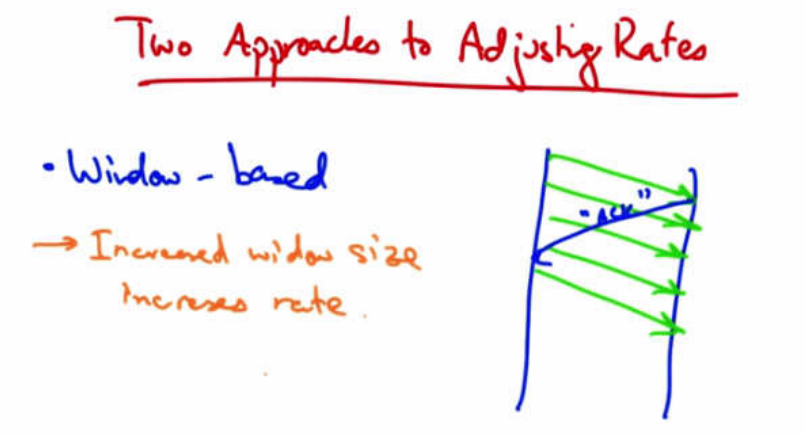
the approach taken by TCP congestion control. In network assisted congestion control, routers provide explicit feedback about the rates that end systems should be sending in. So they might set a single bit indicating congestion, as is the case in TCP's ECN, or explicit congestion notification extensions, or they might even tell the sender an explicit rate that they should be sending at. We're going to spend the rest of the lesson talking about TCP congestion control.

TCP Congestion Control

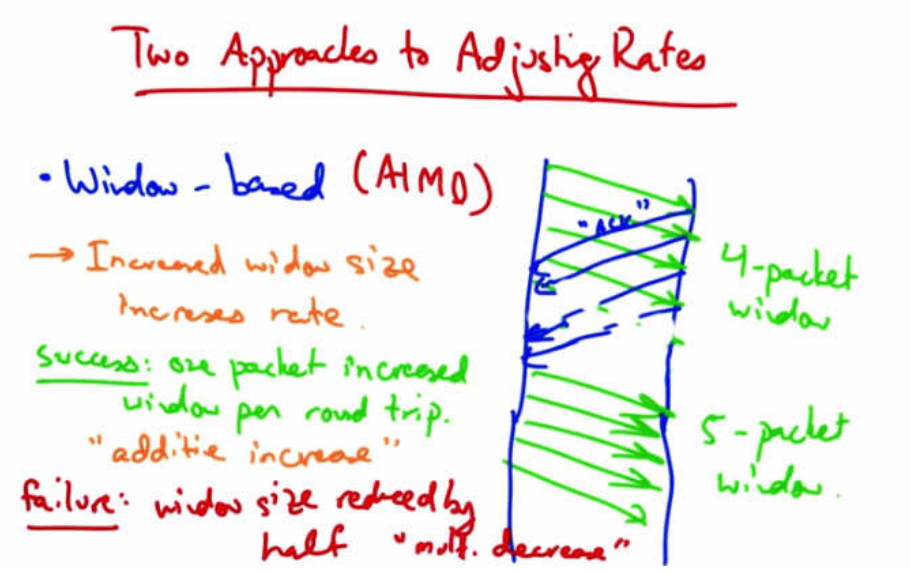


In TCP congestion control, the senders continue to increase their rate until they see packet drops in the network. Packet drops occur because the senders are sending at a rate that is faster than the rate at which a particular router in the network might be able to drain its buffer. So you might imagine, for example, that if all three of these senders are sending at a rate that is equal to the rate at which the router is able to send traffic downstream, then eventually this buffer will fill up. TCP interprets packet loss as congestion. And when senders see packet loss, they slow down as a result of seeing the packet loss. This is an assumption. Packet drops are not a sign of congestion in all networks. For example, in wireless networks, there may be packet loss due to corrupted packets as a result of interference. But in many cases, packet drops do result because some router in the network has a buffer that has filled up and can no longer hold anymore packets and hence it drops the packets as they arrive. So senders increase rates until packets are dropped, periodically probing the network to check whether more bandwidth has become available; then they see packet loss, interpret that as congestion, and slow down. So, congestion control has two parts. One is an increase algorithm, and the other is a decrease algorithm. In the increase algorithm, the sender must test the network to determine whether the network can sustain a higher sending rate. In the decrease algorithm, the senders react to congestion to achieve optimal loss rates, delays in sending rates. Let's now talk about how senders can achieve these increase and decrease algorithms.

Two Approaches to Adjusting Rate

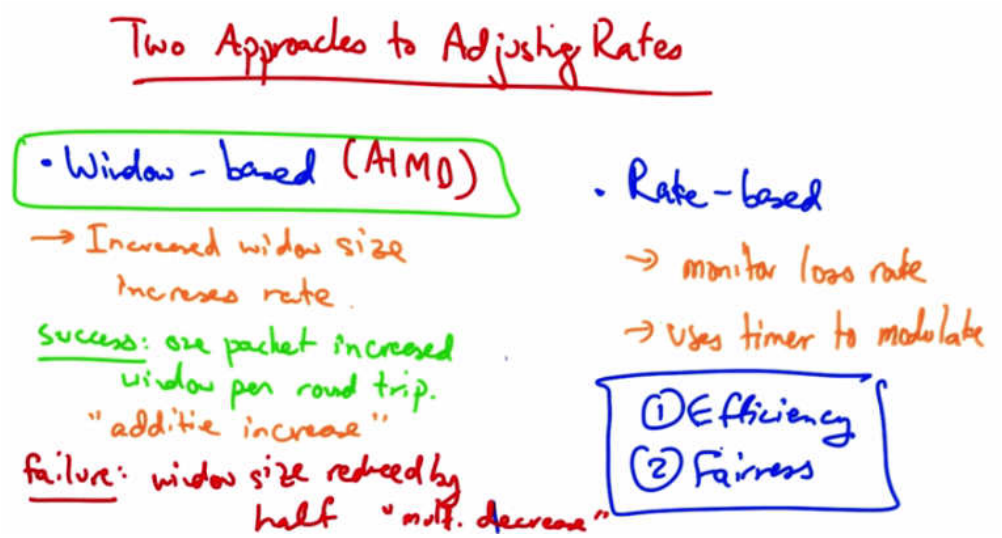


One approach is a window based algorithm. In this approach, a sender can only have a certain number of packets outstanding, or quote, in flight. And the sender uses acknowledgements from the receiver to clock the retransmission of new data. So let's suppose that the sender's window was four packets. At this point, there are four packets outstanding in the network. And the sender cannot send additional packets until it has received an acknowledgement from the receiver. When it receives an acknowledgment, or an ACK from the receiver, the sender can then send another packet. So at this point there are still four outstanding or four unacknowledged packets in flight. In this case if a sender wants to increase the rate at which it's sending, it simply needs to increase the window size. So, for example, if the sender wants to send at a faster rate, it can increase the window size from four, to five. A sender might increase its rate anytime it sees an acknowledgement from the receiver. In TCP, every time a sender receives an acknowledgement, it increases the window size.



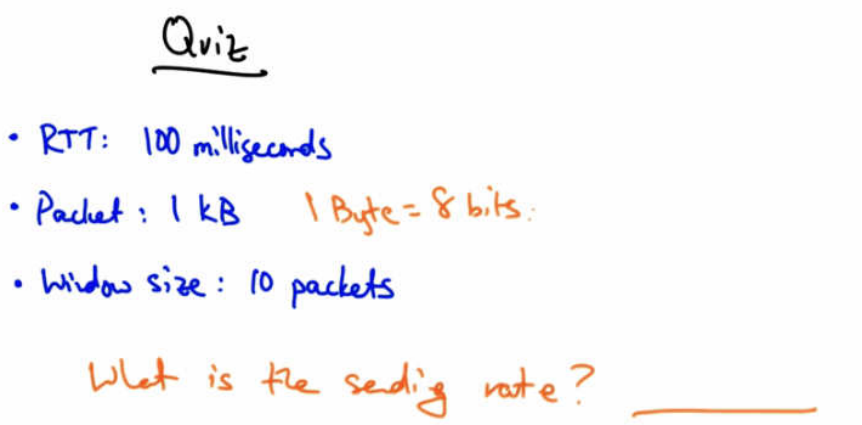
Upon a successful receipt, we want the sender to increase its window by one packet per round trip. So, for example, in this case if the sender's window was initially four packets, then at the

end of a single round trip's worth of sending, we want the next set of transmissions to allow five packets to be outstanding. This is called Additive Increase. If a packet is not acknowledged, the window size is reduced by half. This is called Multiplicative Decrease. So TCP's congestion control is called additive increase multiplicative decrease, or AIMD.



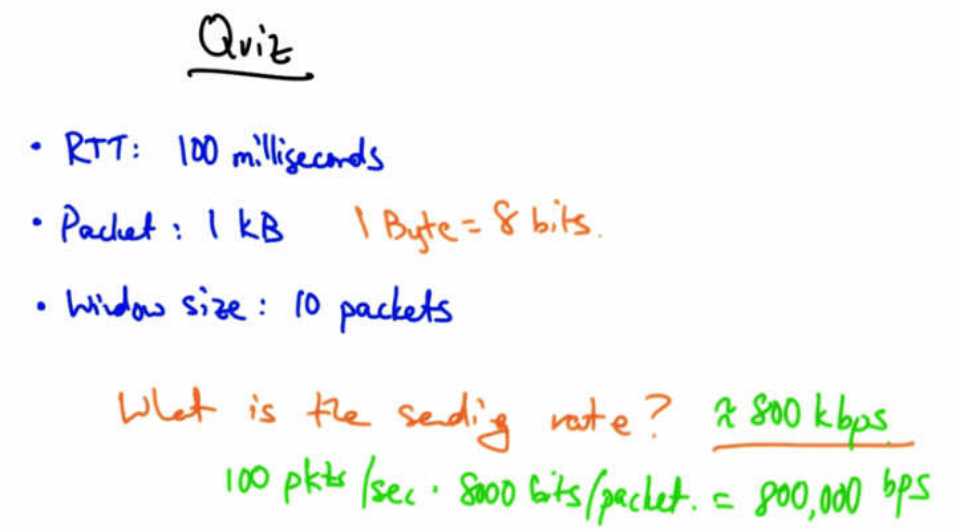
The other approach to adjusting rates is an explicit rate-based congest control algorithm. In this case the sender monitors the loss rate and uses a timer to modulate the transmission rate. Window based congestion control, or AIMD, is the common way of performing congestion control in today's computer networks. In the next lesson we will talk about the two goals of TCP congestion control further (efficiency and fairness) and explore how TCP achieves those goals.

Window Based Congestion Control Quiz



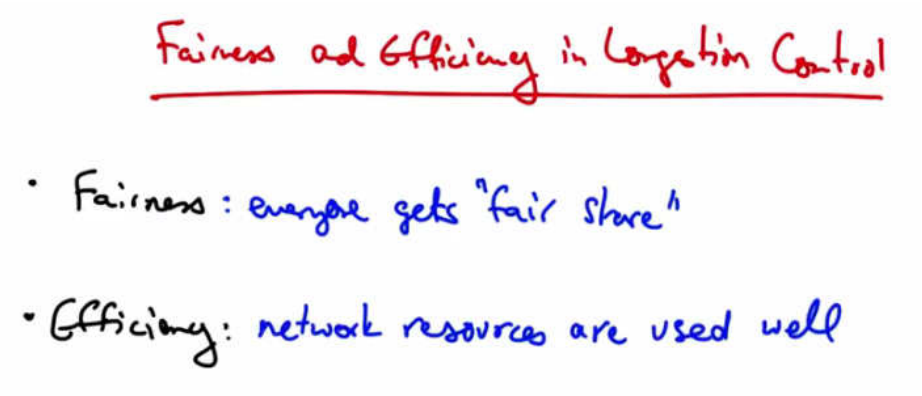
Let's have a quick quiz on window based congestion control. Suppose the round trip time between the sender and receiver is 100 ms, each packet is 1kb, and the window size is 10 packets. What is the rate at which the sender is sending? Please put your answer here in terms of kilobits per second, keeping in mind that 1 byte is 8 bits.

Window Based Congestion Control Solution

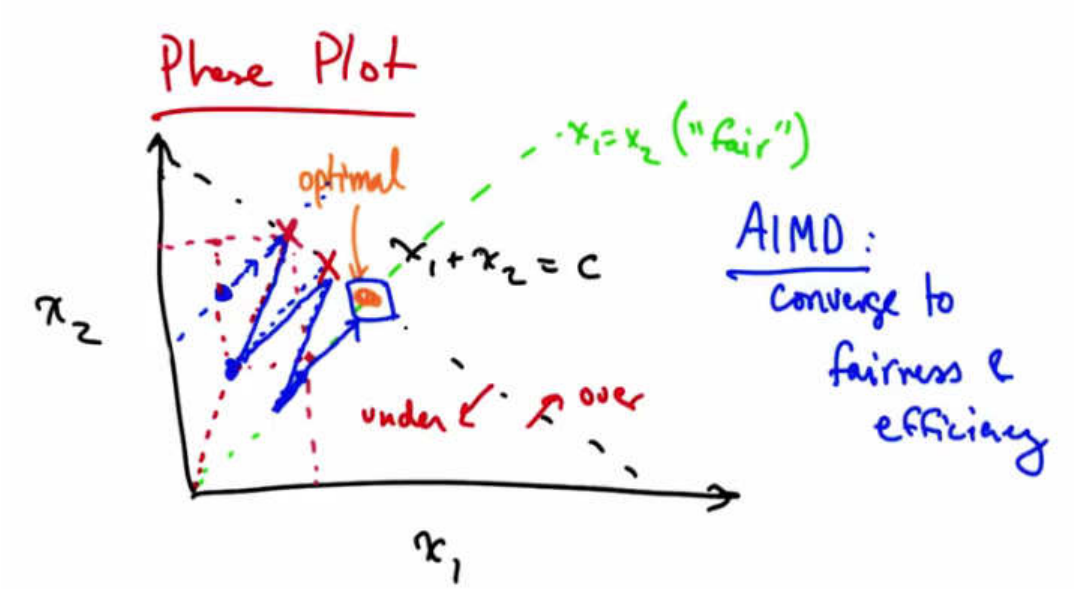


The sending rate of the sender is approximately 800 kbps. With a window size of ten packets and a round trip time of 100 milliseconds, the sender can send 100 packets per second. With each packet being one kilobyte, or 8000 bits, that gives us 800,000 bits per second or about 800 kilobits per second.

Fairness and Efficiency in Congestion Control

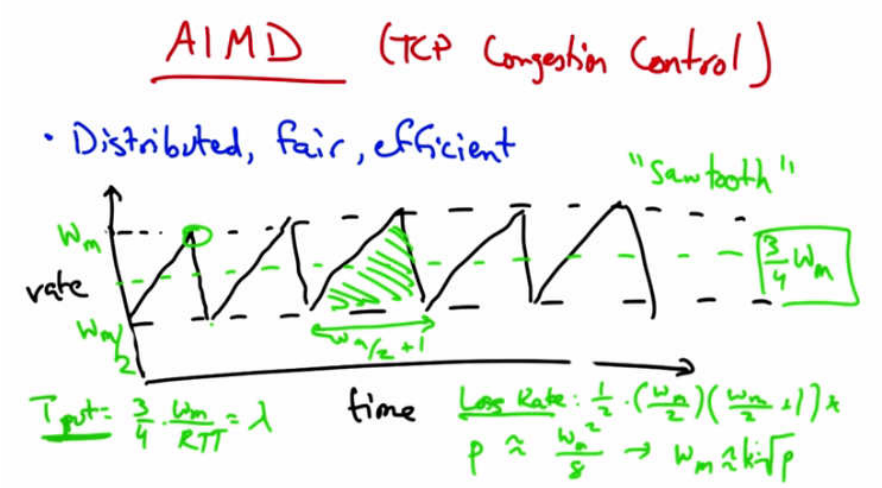


The two goals of congestion control are fairness (meaning every sender gets their fair share of the network resources) and efficiency (meaning that the network resources are used well). In other words we shouldn't have a case where there are spare capacity or resources in the network, and senders have data to send, but are not able to send it. So, we'd like the network to be used efficiently, but we'd also like it to be shared among the senders.

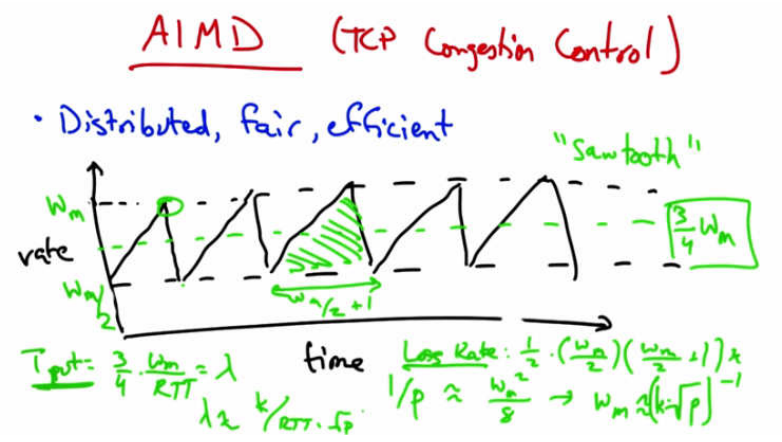


We can represent fairness and efficiency in terms of a phase plot, where each axis represents a particular user, or particular senders' allocation. In this case we just have two users, one and two, and we represent their allocations with X1 and X2. If the capacity of the network is C, then we can represent the optimal operating line as X1 + X2 being some constant, C. Anything to the left of this diagonal line represents under utilization of the network, and anything to the right of the line represents overload. We can also represent another line, X1 = X2 as some notion of fair allocation. So the optimal point is where the network is neither under or over utilized, and when the allocation is fair. So being on this diagonal line represents efficiency, and being on the green diagonal line represents fairness. We can use the phase plot to understand why senders who use additive increase multiplicative decrease, converge to fairness. The senders also converge to the efficient operating point. Let's suppose that we start at the operating point shown in blue. At this point both senders will additively increase their sending rates. Additive increase results in moving along a line that is parallel to x1 and x2, since both senders increase their rate by the same amount. Additive increase will continue until the network becomes overloaded. At this point the senders will see a loss and perform multiplicative decrease. In multiplicative decrease each sender decreases its rate by some constant factor of its current sending rate. For example, suppose each one of these senders decreases its sending rate by half. The resulting operating point is shown by this second blue dot. Note that that new operating point, as a result of multiplicative decrease, is on a line between the point on the efficiency line that the centers hit, and the origin. At this point the sender's will again increase their sending rate along a line that's parallel to X1 equals X2 until they hit over load again, at which point they will again retreat towards the origin. You can see that eventually the senders will reach this optimal operating point through the path that's delineated by the blue line. To think about this a bit more you can see that every time additive increase is applied, that increases efficiency. Every time multiplicative decrease is applied that improves fairness because every time we apply multiplicative decrease, we get closer to this X1 equals X2 line.

AIMD

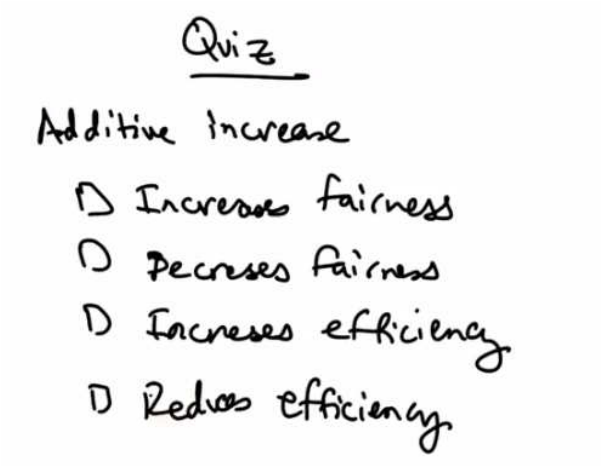


The result is the additive increase multiplicative decrease congestion control algorithm. The algorithm is distributed, meaning that all the senders can act independently, and we've just shown using the phase plots that it's both fair and efficient. To visualize this sending rate over time, the sender's sending rate looks roughly as shown. We call this the TCP sawtooth behavior or simply the TCP sawtooth. TCP periodically probes for available bandwidth by increasing its rate using additive increase. When the sender reaches a saturation point by filling up a buffer in a router somewhere along the path, it will see a packet loss, at which point it will decrease its sending rate by half. You can thus see that a TCP sender sends at a sending rate shown by the dotted green line that is halfway between the maximum window size at which the sender sends, and half that rate which it backs off to when it sees a loss. You can see that between the lowest sending rate and the highest is w_m over 2 plus 1 round trips. Now, given that rate we can compute the number of packets between periods of packet loss and compute the loss rate from this. The number of packets sent for every packet lost is the area of this triangle. So the lost rate is on the order of the square of the maximum window divided by some constant. Now, the throughput is the average rate, 3 4ths w_max divided by the RTT. Now if we want to relate the throughput to the loss rate, where we call the loss rate p and the throughput lambda, we simply need to solve for w_m.



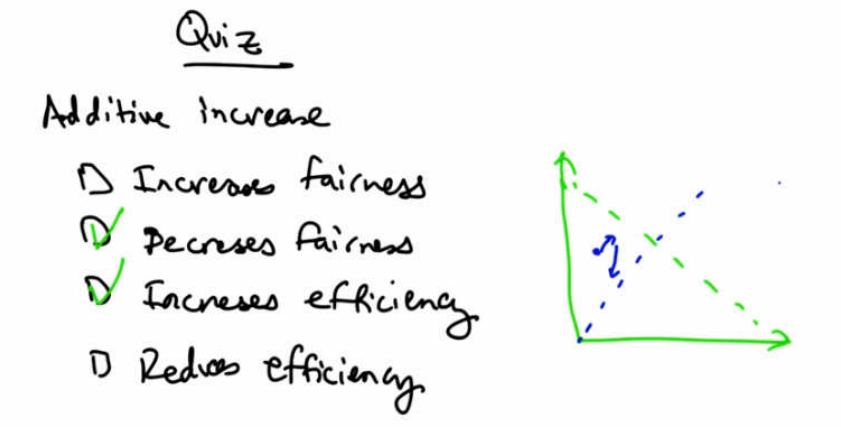
And I'm just going to get rid of the constant. So a loss occurs once for this number of packets, so the loss rate is simply 1 over that quantity. And then when we solve for w_m and plug in for throughput, we see that the throughput is inversely proportional to both the round trip time and the square root of the loss rate.

Additive Increase Quiz



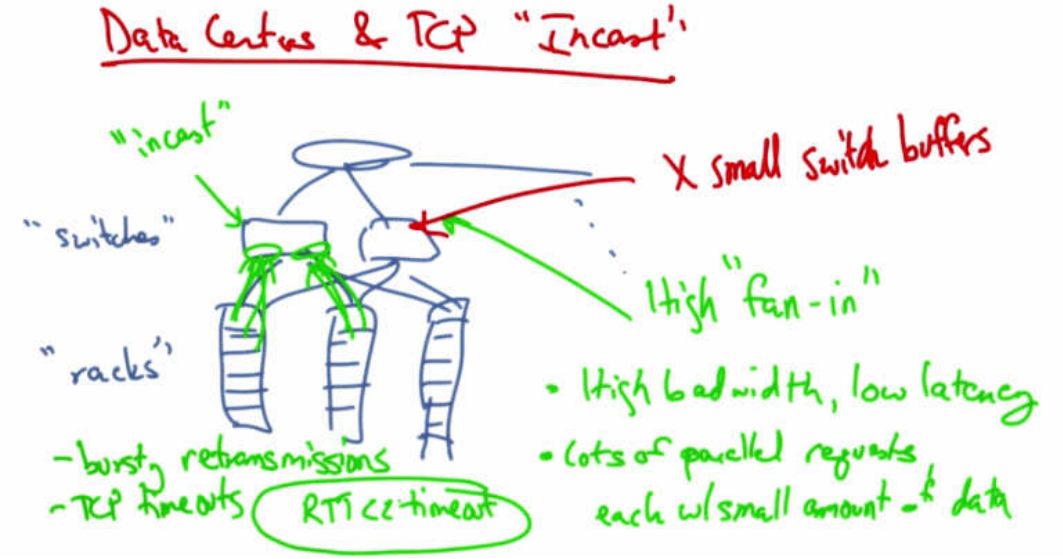
So as a quick quiz, and returning to our phase plot, does additive increase increase or decrease fairness? And does it increase, or decrease, or reduce, efficiency? Please check all that apply

Additive Increase Solution



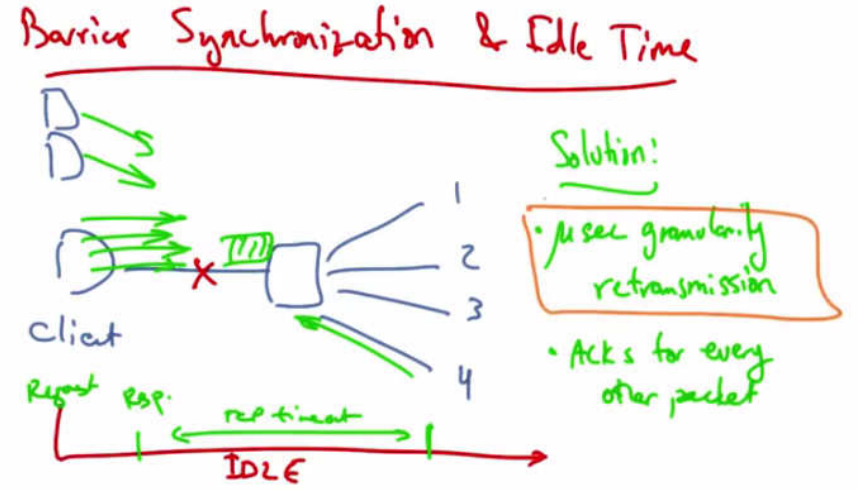
Additive increase increases efficiency because it gets us closer to that efficiency line parallel to the X1 equals X2 line. It technically also decreases fairness because the centers are both increasing their rates, so we're relatively further away from this X1 equals X2 line. In contrast, remember that multiplicative decrease reduces efficiency by moving us further away from this green dotted line, but it increases fairness by moving us closer to the blue dotted line.

Data Centers and TCP Incast



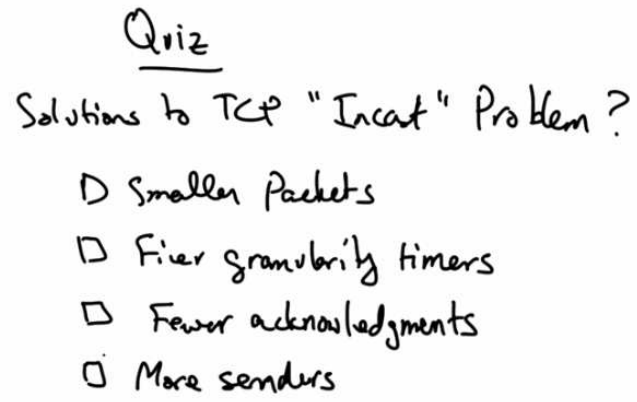
We'll now talk about TCP congestion control in the context of modern datacenters. And we'll talk about a particular TCP throughput collapse problem called the TCP incast problem. A typical data center consists of a set of server racks, each holding a large number of servers, the switches that connect those racks of servers, and the connecting links that connect those switches to other parts of the topology. So the network architecture is typically made up of some sort of tree and switching elements that progressively are more specialized and expensive as we move up the network hierarchy. Some of the characteristics of a data center network include a high fan in. There is a very high amount of fan in between the leaves of the tree and the top of the root workloads are high bandwidth and low latency, and many clients issue requests in parallel, each with a relatively small amount of data per request. The other constraint that we face is that the buffers in these switches can be quite small. So when we combine the requirements of high bandwidth and low latency for the applications, the presence of many parallel requests coming from these servers. and the fact that the switches have relatively small buffers, we can see that potentially there will be a problem. The throughput collapse that results from this phenomenon is called the TCP Incast problem. Incast is a drastic reduction in application throughput that results when servers using TCP all simultaneously request data, leading to a gross underutilization of network capacity in many-to-one communication networks like a datacenter. The filling up of the buffers here at the switches result in bursty retransmissions that overfill the switch buffers. And these bursting retransmissions are cause by TCP timeouts. The TCP timeouts can last hundreds of milliseconds. But the roundtrip time in a data center network is typically less than a millisecond. Often just hundreds of microseconds. Because the roundtrip times are so much less than TCP timeouts, the centers will have to wait for the TCP timeout before they retransmit an application. Throughput can be reduced by as much as 90% as a result of link idle time.

Barrier Synchronization and Idle Time



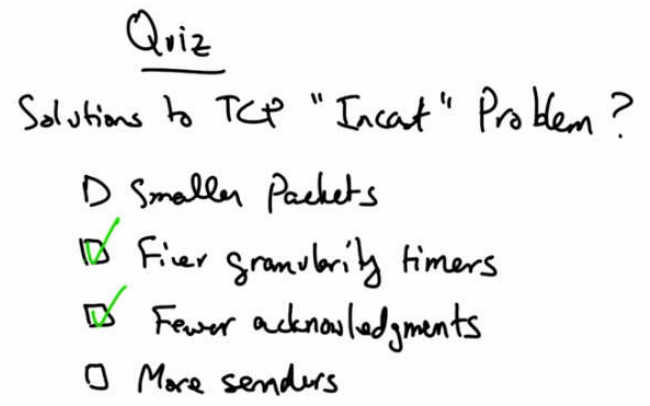
A common request pattern in data centers today is something called barrier synchronization whereby a client or an application might have many parallel threads, and no forward progress can be made until all the responses for those threads are satisfied. For example, a client might send a synchronized read with four parallel requests. But, suppose that the fourth is dropped. At this point we have a request sent at time zero, then we see a response less than a millisecond later, and at this point, threads one to three complete but TCP may time out on the fourth. In this case, the link is idle for a very long time while that fourth connection is timed out. The addition of more servers in the network induces an overflow of the switch buffer, causing severe packet loss, and inducing throughput collapse. One solution to this problem is to use fine grained TCP retransmission timers, on the order of microseconds, rather than on the order of milliseconds. Reducing the retransmission timeout for TCP thus improves system throughput. Another way to reduce the network load is to have the client acknowledge every other packet rather than every packet, thus reducing the overall network load. The basic idea here, and the premise, is that the timers need to operate on a granularity that's close to the round-trip time of the network. In the case of a data center that's hundreds of microseconds or less.

TCP Incast Quiz



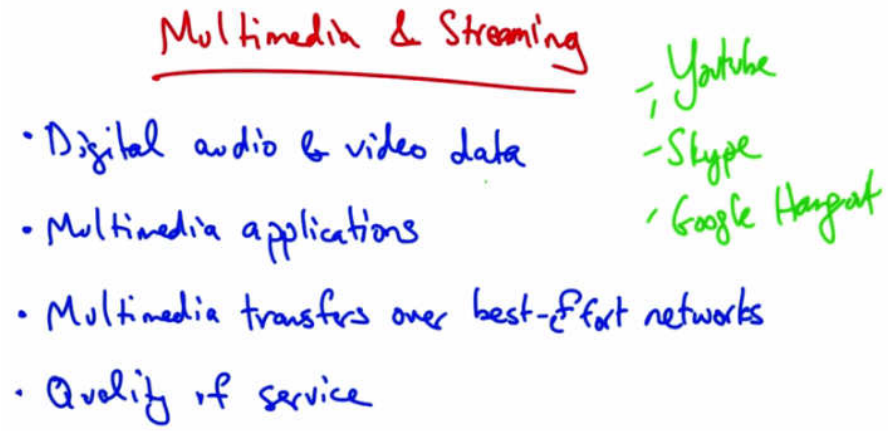
As a quick review what are some solutions to the TCP incast problem? Having senders send smaller packets? Using finer granularity TCP timeout timers. Having the clients send fewer ackowledgements or having fewer TCP senders?

TCP Incast Solution



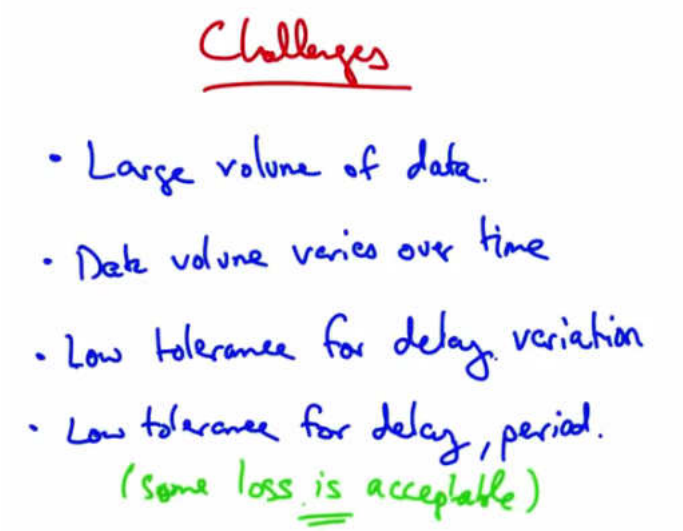
Using finer granularity timers and having the clients acknowledge only every other packet, as oppose to every packet, are possible solutions to the TCP Incast problem.

Multimedia and Streaming



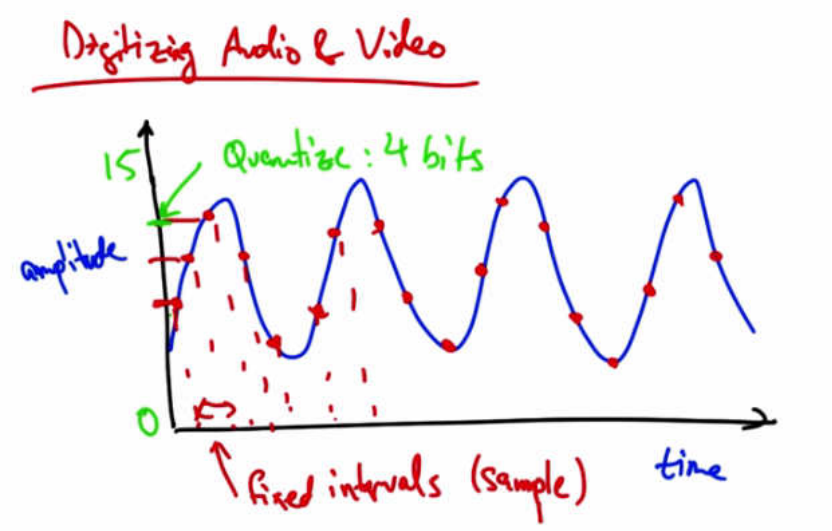
In this lesson, we'll talk about multimedia and streaming. We'll talk about digital audio and video data, multimedia applications (in particular, streaming audio and video for playback), multimedia transfers over a best-effort network (in particular, how to tolerate packet loss delay and jitter), and quality of service. So we use multimedia and streaming video very frequently on today's internet. YouTube streaming videos are an example of multimedia streaming as are applications for video or voice chat such as Skype or Google Hangout. In this lecture we'll talk about the challenges for streaming these types of applications over best effort networks as well has how to solve those challenges. We'll also talk about the basics of digital audio and video data. First of all, let's talk about the challenges for media streaming.

Challenges



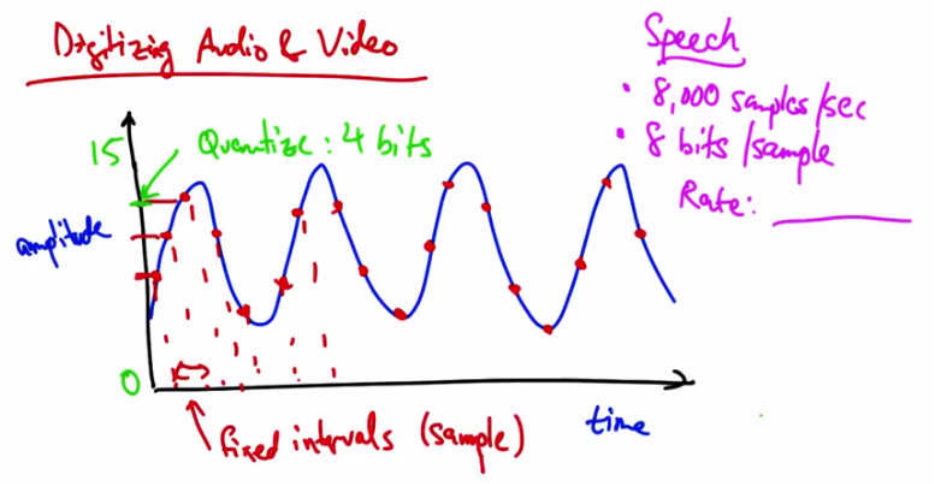
One challenge is that, there's a large volume of data. Each sample is a sound or an image and there are many samples per second. Sometimes because of the way data is compressed, the volume of data that's being sent may vary over time. In particular, the data may not be set at a constant rate. But in streaming, we want smooth playout, so the variable volume of data can pose challenges. Users typically have a very low tolerance for delay variation. Once playout of a video starts for example, you want that video to keep playing. It's very annoying if once you've started playing, that the video stops. The users might have a low tolerance for delay period, so in cases like games or Voice over IP, delay is typically just unacceptable, although users can tolerate some loss. Before we get into how the network solves these challenges. Let's talk a little bit about digitizing audio and video.

Digitizing Audio and Video



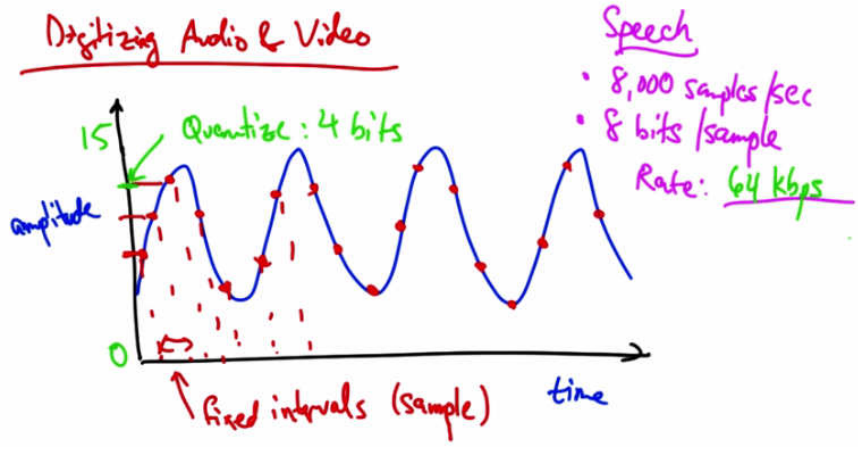
Suppose we have an analog audio signal that we'd like to digitize, or send as a stream of bits. What we can do is sample the audio signal at fixed intervals, and then represent the amplitude of each sample with a fixed number of bits. For example, if our dynamic range was from 0 to 15, we could quantize the amplitude of this signal such that each sample could be represented with four bits.

Digitizing Audio and Video Quiz 1



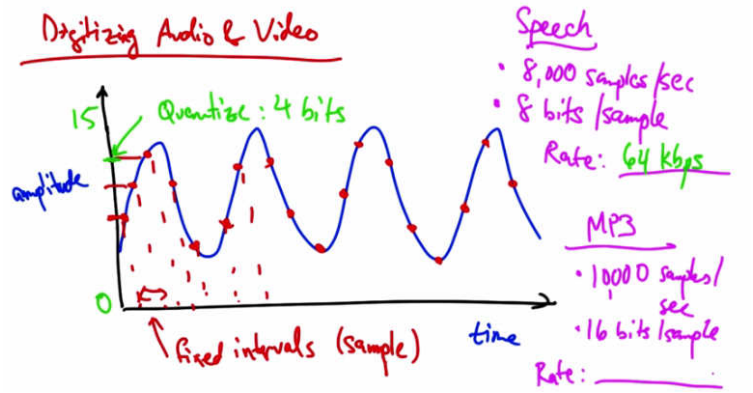
Let's take a couple of examples. So with speech you might take 8000 samples per second, and you might have 8 bits per sample. So what is the sampling rate in this case? Please give your answer in kilobits per second.

Digitizing Audio and Video Solution 1



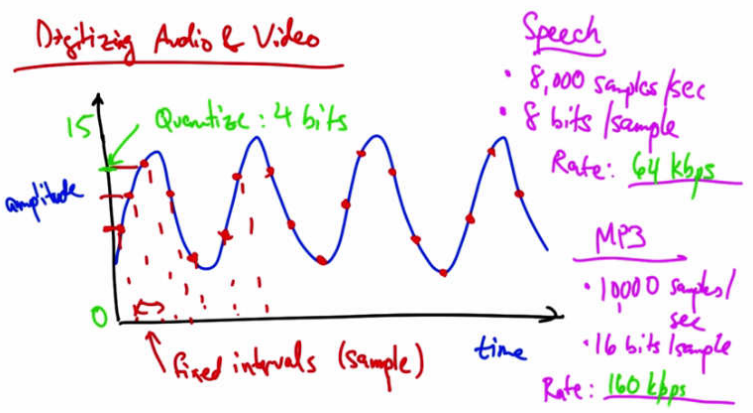
At 8,000 samples per second, and eight bits for every sample, the rate of digitized speech would be 64 kbps, which is a common bit rate for audio.

Digitizing Audio and Video Quiz 2



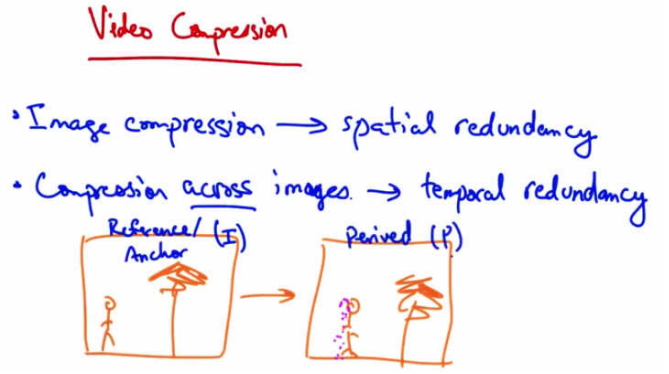
Suppose we have a MP3 with 10,000 samples per second and 16 bits per sample. What's the resulting rate in this case in kbps?

Digitizing Audio and Video Solution 2

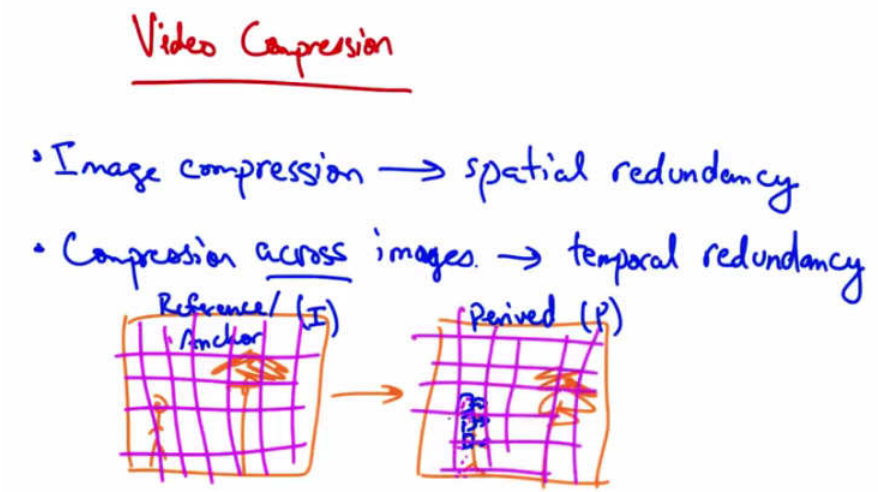


The resulting rate in this case is, a 160 kbps.

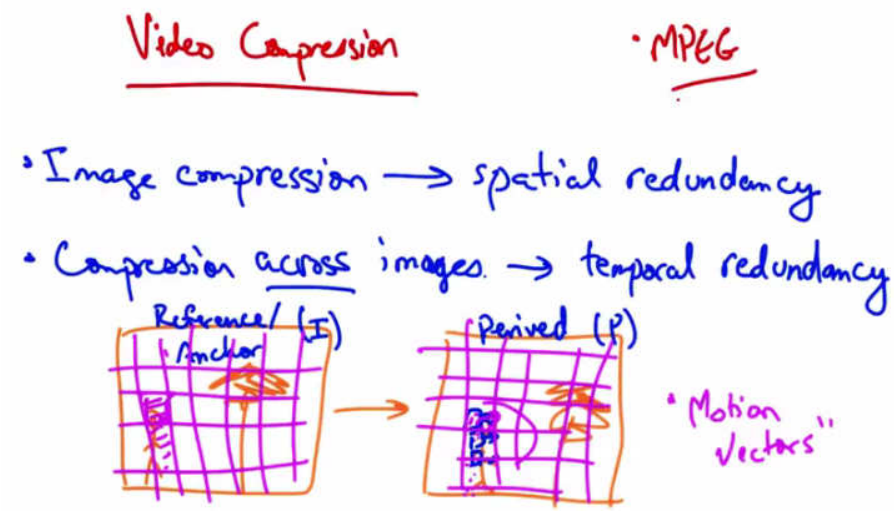
Video Compression



Video compression works in slightly different ways. Each video is a sequence of images and each image can be compressed with spatial redundancy, exploiting aspects that humans tend not to notice. Also there is temporal redundancy. Between any two video images, or frames, there might be very little difference. So if this person was walking towards a tree, you might see a version of the image that's almost the same, except with the person shifted slightly to the right. Video compression uses a combination of static image compression on what are called reference frames, or anchor frames (sometimes called I frames), and derived frames, sometimes called P frames. The P frame can be represented as the I frame, compressed.

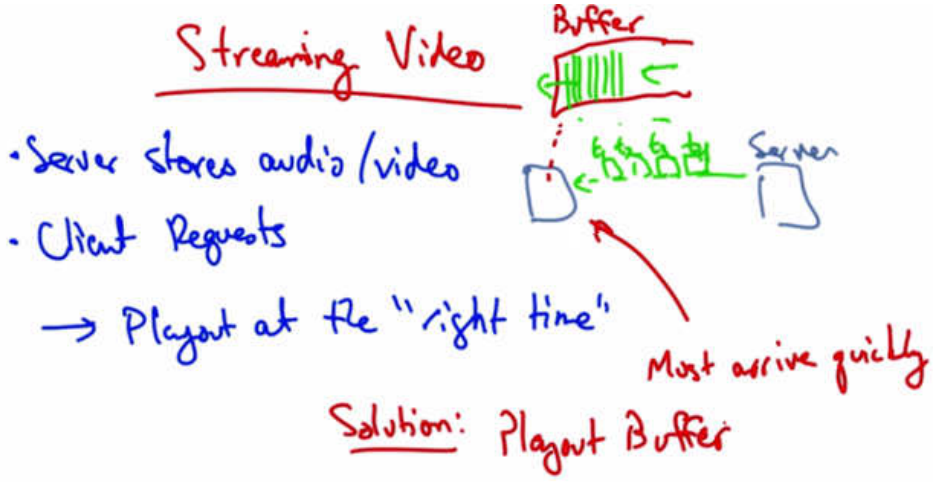


If we take the I frame and divide it into blocks, we can then see that the P frame is almost the same except for a few blocks here that can be represented in terms of the original I frame blocks, plus a few motion vectors.



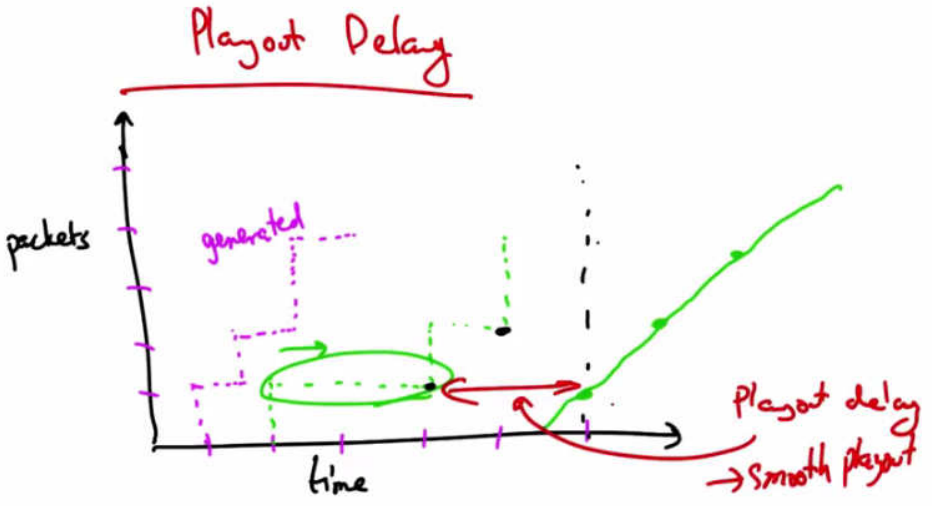
A common video compression format that's used on the internet is called MPEG.

Streaming Video



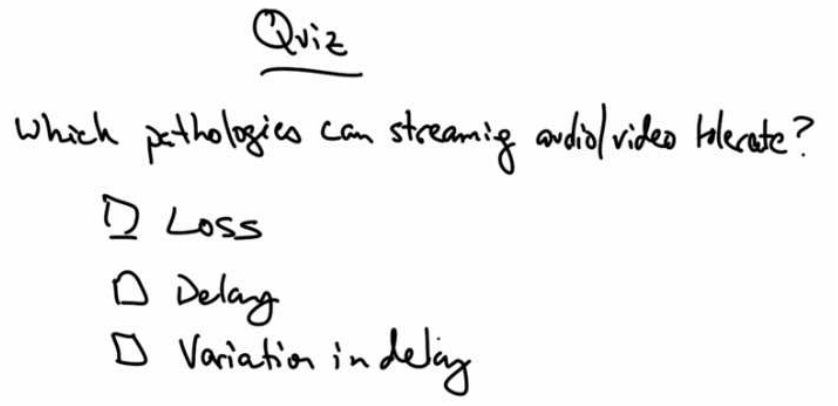
In a streaming video system where the server streams stored audio and video, the server stores the audio or video files, the client requests the files, and plays them as they download. It's important to play the data at the right time. The server can divide the data into segments and then label each segment with a time stamp indicating the time at which that particular segment should be played, so the client knows when to play that data. The data must arrive at the client quickly enough, otherwise the client can't keep playing. The solution is to have a client use what's called a playout buffer, where the client stores data as it arrives from the server, and plays the data for the user in a continuous fashion. Thus, data might arrive more slowly or more quickly from the server, but as long as the client is playing data out of the buffer at a continuous rate, the user sees a smooth playout. A client may typically wait a few seconds before it starts playing the stream to allow data to be built up in this buffer to account for cases when the server might have times where it is not sending at a rate that's sufficient to satisfy the client's playout rate.

Playout Delay



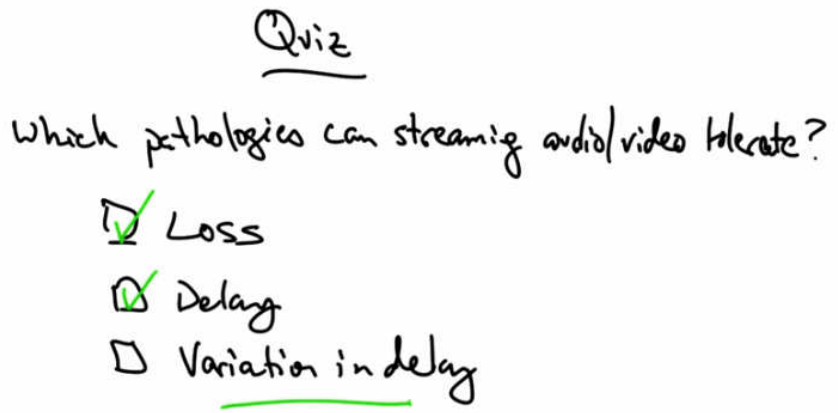
Looking at this graphically, we might see packets generated at a particular rate and the packets might be received at slightly different times, depending on network delay. These types of delays are the types that we want to avoid when we playout. So if we wait to receive several packets and fill the buffer before we start playing, say to here, then we can have a playout schedule that is smooth regardless of the erratic arrival times that may result from network delays. So this playout delay or buffering allows the client to achieve a smooth playout. Some delay at the beginning of the playout is acceptable. Startup delays of a few seconds are things that users can typically tolerate, but clients cannot tolerate high variation in packet arrivals if the buffer starves or if there aren't enough packets in the buffer. Similarly, small amount of loss or missing data does not disrupt the playback, but retransmitting a lost packet might actually take too long and result in delays or starvation of the playout buffer.

Streaming Quiz



So as a quick review, which of these pathologies can streaming audio and video tolerate? Packet loss, delay, variation in delay, or jitter?

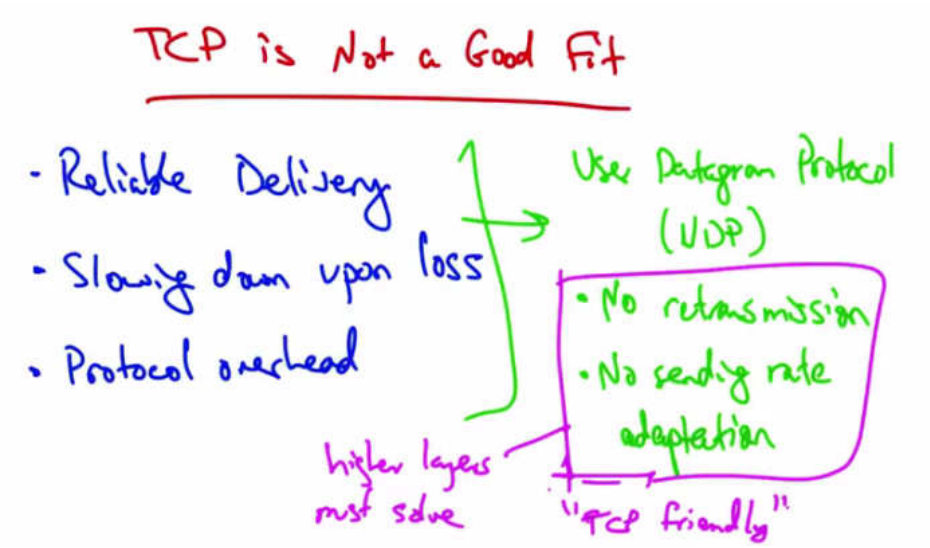
Streaming Solution



Some delay at the beginning of a packet stream is acceptable. And similarly, some small amount of missing data is okay. We can tolerate small amounts of missing data that result in slightly

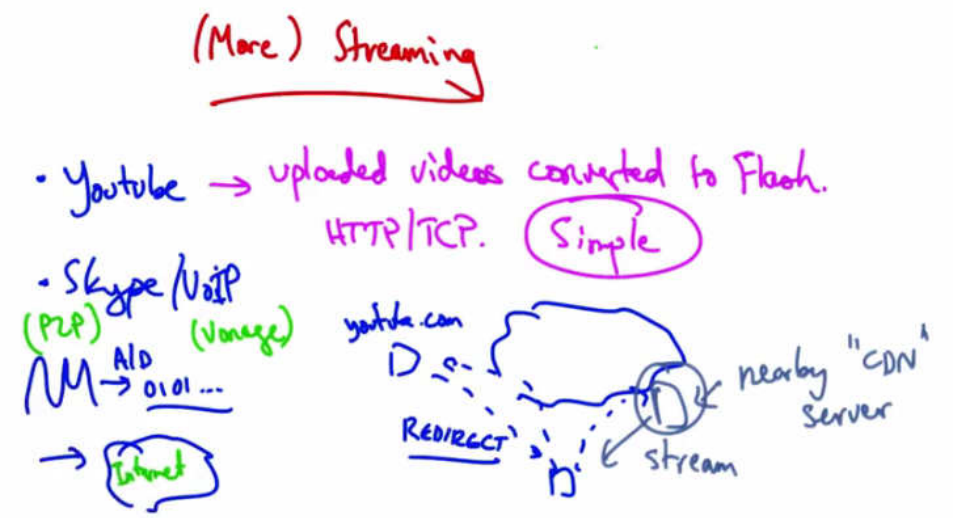
reduced quality of the audio or video stream. On the other hand, a receiver is not very good at tolerating variability in packet delay in the packet stream, particularly if the client buffer is starved.

TCP is Not a Good Fit



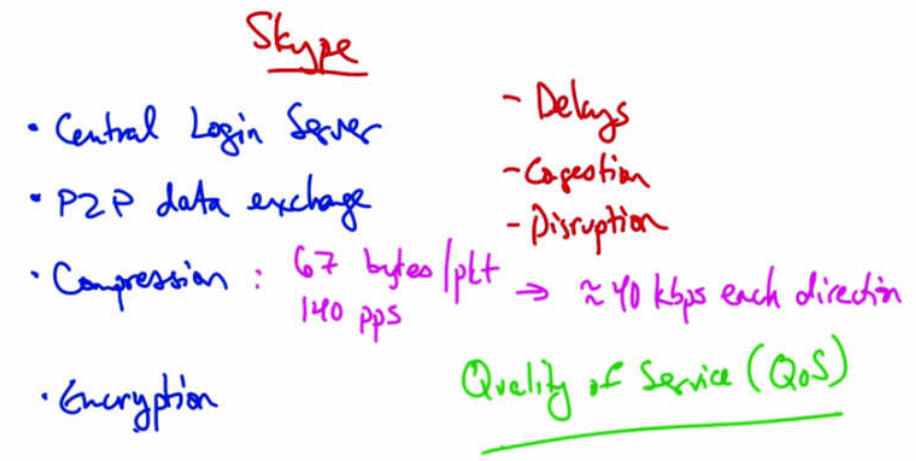
It turns out that TCP is not a good fit for congestion control for streaming video, or streaming audio. TCP retransmits lost packets, but retransmissions may not always be useful. TCP also slows down its sending rate after packet loss, which may cause starvation of the client. There's also a fair amount of overhead in the protocol. A TCP header of 20 bytes for every packet is large for audio samples and sending acknowledgements for every other packet may be more feedback than is needed. Instead, one might consider using UDP. UDP does not retransmit lost packets and it does not automatically adapt the sending rate. It also has a smaller header. Because UDP does not automatically retransmit packets or adjust the sending rate, many things are left to higher layers, potentially the application, such as when to transmit the data, how to encapsulate it, whether to retransmit, and whether to adapt the sending rate, or to adapt the quality of the video or audio encoding. So higher layers must solve these problems. In particular, the sending rate still needs to be friendly or fair to other TCP senders, which may be sharing the link. There are a variety of video streaming and audio streaming transport protocols that are built on top of UDP that allows senders to figure out when and how to retransmit lost packets and how to adjust sending rates.

(More) Streaming



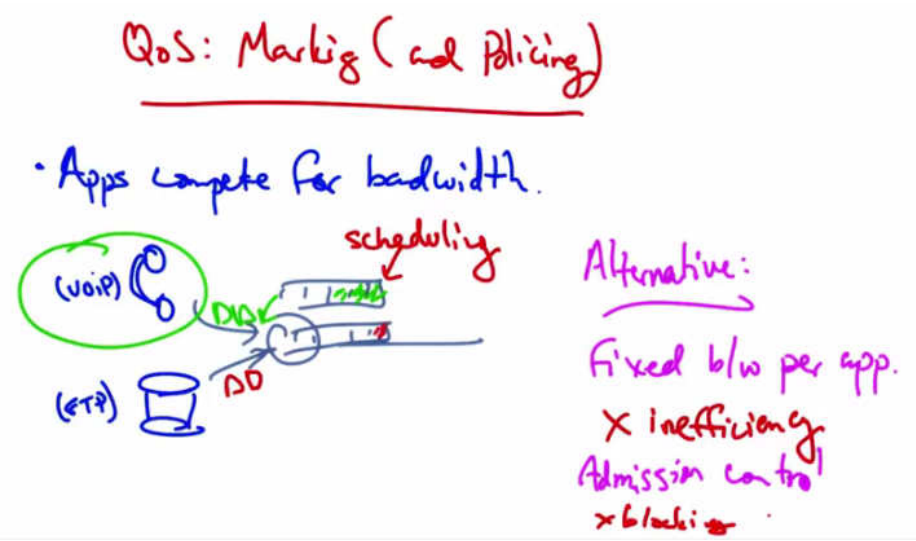
We're going to talk a little bit more about streaming and in particular, specific applications and how they work. We'll talk about a streaming video application, YouTube, and a Voice over IP (or streaming audio) application, Skype. With YouTube, all uploaded videos are converted to Flash or html5 and nearly every browser has a Flash plug-in. Thus, every browser can essentially play these videos. HTTP and TCP are implemented in every browser and the streams easily get through most firewalls. So, in this case, even though we've talked about why TCP is suboptimal for streaming, the designers of YouTube decided to keep things simple, potentially at the expense of video quality. When a client makes an HTTP request to youtube.com, it is redirected to a content distribution network server located in a content distribution network, such as Limelight or perhaps even YouTube's own content distribution network. We will talk about content distribution networks later on in this course. When the client sends an HTTP get message to this CDN server, the server responds with a video stream. Similarly with Skype, or Voice Over IP, your analog signal is digitized through an A to D conversion and this resulting digitized bit stream is sent over the internet. In the case of Skype, this A to D conversion happens by way of the application. And in the case of Voice over IP, this conversion might be performed with some kind of phone adapter that you actually plug your phone into. An example of that might be Vonage. In VoIP with an analogue phone, the adapter converts between the analogue and digital signal. It sends and receives data packets and then communicates with the phone in a standard way. Skype, on the other hand, is based on what's known as peer-to-peer technology where individual users using Skype actually route voice traffic through one another. We will talk more about peer-to-peer content distribution later in this course.

Skype



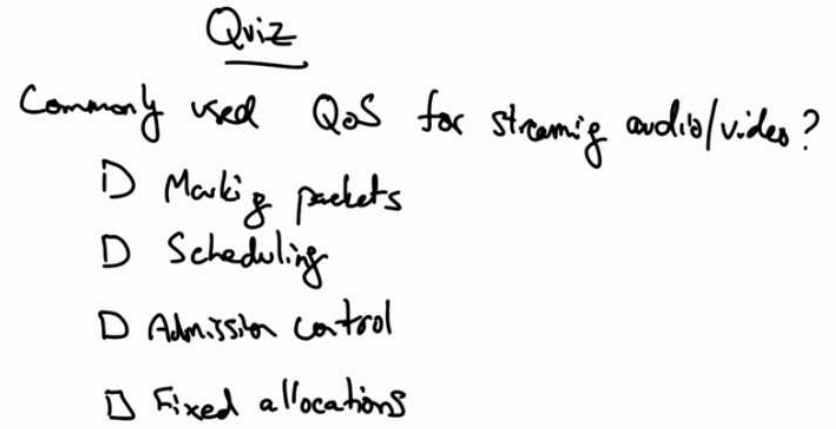
So Skype has a central log-in server but then uses pier to pier to exchange the actual voice streams, and compresses the audio to achieve a fairly low bit rate. At around 67 bytes per packet and 140 packets per second, we see Skype uses about 40 kilobites per second in each direction. Data packets are also encrypted in both directions. Good audio compression and avoiding hops through far away hosts can improve the quality of the audio, as can forward error correction. Ultimately, though, the network is a major factor. Long propagation delays, high congestion, or disruptions as a result of routing changes can all degrade the quality of a voice over IP call. To ensure that some streams achieve acceptable performance levels, we sometimes use what's called Quality of Service. One way of doing Quality of Service is through explicit reservations. But, another way is to simply mark certain packet streams as higher priorities than others. Let's first take a look at marking and policing of traffic sending rates.

Marking (and Policing)



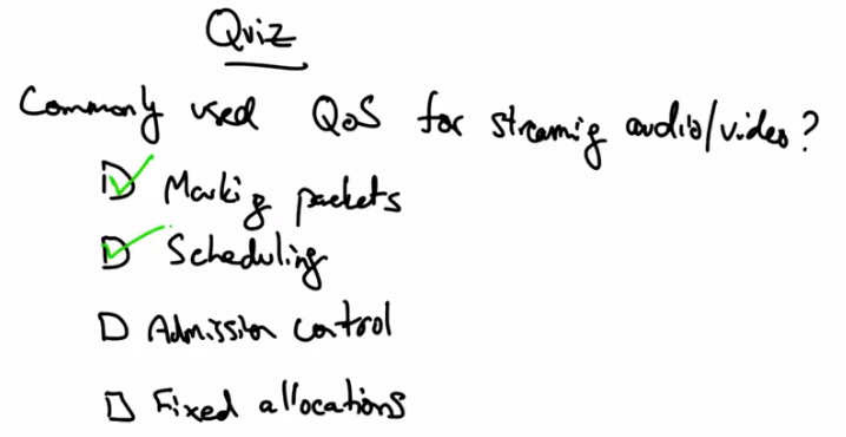
So we know from before that applications compete for bandwidth. Consider a Voice over IP application and a file transfer application that want to share a common network link. In this case, we'd like the audio packets to receive priority over the file transfer packets since the user's experience can be significantly degraded by lost or delayed audio packets. In this case, we want to mark the audio packets as they arrive at the router so that they receive a higher priority than the file transfer packets. You might imagine implementing this with priority queues where the VOIP packets were put in one queue and the file transfer packets are put in a separate queue that is served less often than a high priority queue. An alternative is to allocate fixed bandwidth per application, but the problem with this alternative is that it can result in inefficiency if one of the flows doesn't fully utilize its fixed allocation. So the idea, in addition to marking and policing, is to apply scheduling. One way of applying scheduling is to use what's called weighted fair queuing, whereby the queue with the green packets is served more often than the queue with the red packets. Another alternative is to use admission control, whereby an application declares its needs in advance, and the network may block the application's traffic if the application can't satisfy the needs. A busy signal on a telephone network is an example of admission control. But can you imagine how annoying it would be if you attempted to load a web page and were blocked. This blocking, or the user experience that results from it, is a negative consequence of admission control and is one of the reasons it's not commonly applied for internet applications.

QoS Quiz



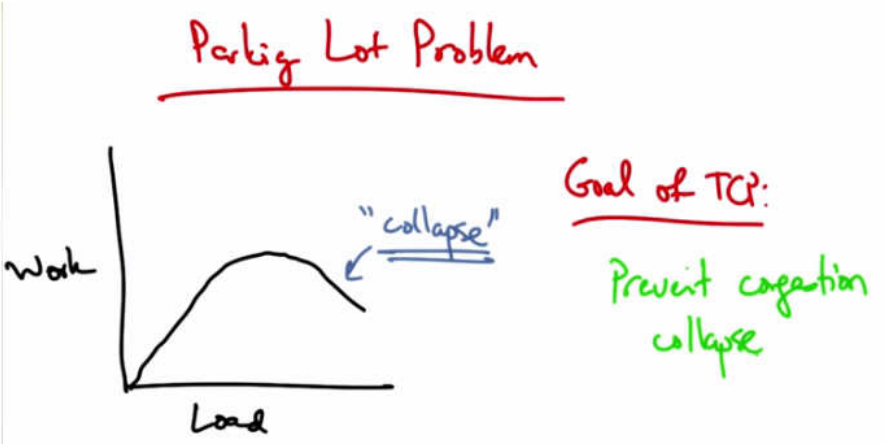
So as a quick quiz what are some commonly used Quality of Service techniques for streaming audio and video? Marking packets and putting them into queues with different priorities, scheduling the service of these queues at different rates depending on the priority of the application, blocking flows with admission control if the network can't satisfy their rates, or performing fixed bandwidth allocations? Please check all that apply.

Qos Solution

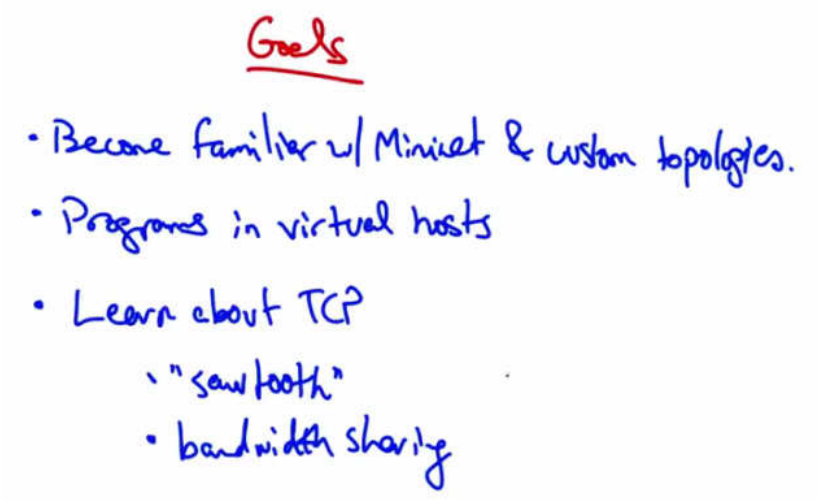


A common way of applying QoS for streaming applications is to mark the packets at a higher priority, put those packets into a higher priority queue, and then schedule that queue so that it's serviced more regularly, or more frequently, than traffic or applications that are lower priority.

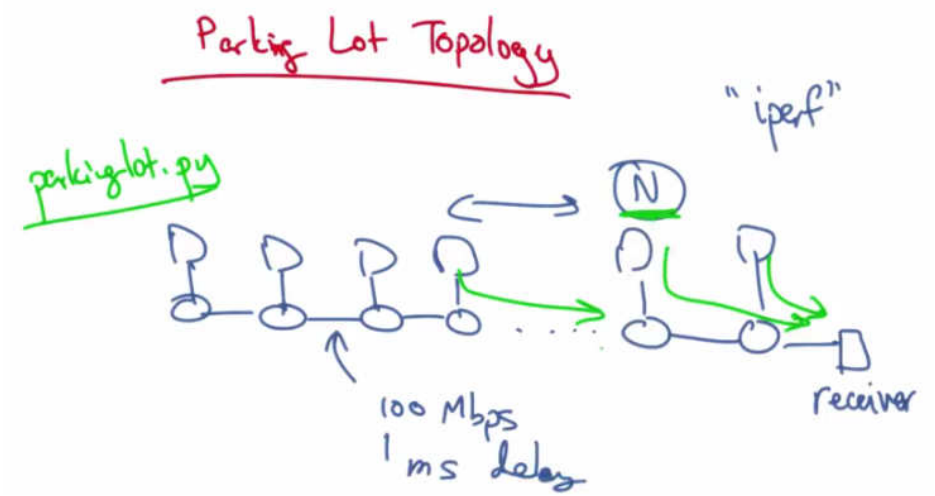
Parking Lot Problem



Recall that congestion collapse occurs when packets consume valuable network resources only to get dropped later at a downstream link. And although the network is forwarding packets, none of them actually reach the destination and no useful communication occurs, resulting in what we called congestion collapse. We talked about congestion collapse in an earlier lesson in this course.



Recall that the goal of TCP is to prevent congestion collapse. In this assignment, you will become familiar with the Mininet environment, creating custom network topologies, running programs in virtual hosts that generate TCP traffic, and learn about TCP congestion control, an the TCP sawtooth. You'll also see how bandwidth is shared across multiple flows.



We are going to explore these concepts in a particular topology that we'll just call a parking lot. So in the assignment, you'll create the topology below with the following parameters. N will be the number of receivers connected via this switch like topology, the data rate of each link will be 100 megabits per second, and the delay of each link will be one millisecond. Now you're going to use iperf to generate simultaneous TCP flows from each of the senders to the lone receiver. Then you're going to plot the time series of throughput versus time for each of these senders for each of your experiments as you vary the value of N. Your plot should run for 60 seconds. We have given an initial file called parkinglot.py. Your goal is to fill in the gaps. You'll need to both fill in the part that sets up the topology using a parameter of N, and then you'll need to fill in the part that actually generates the TCP flows between the senders and the receiver using iperf, and monitor the throughput of each of these flows.

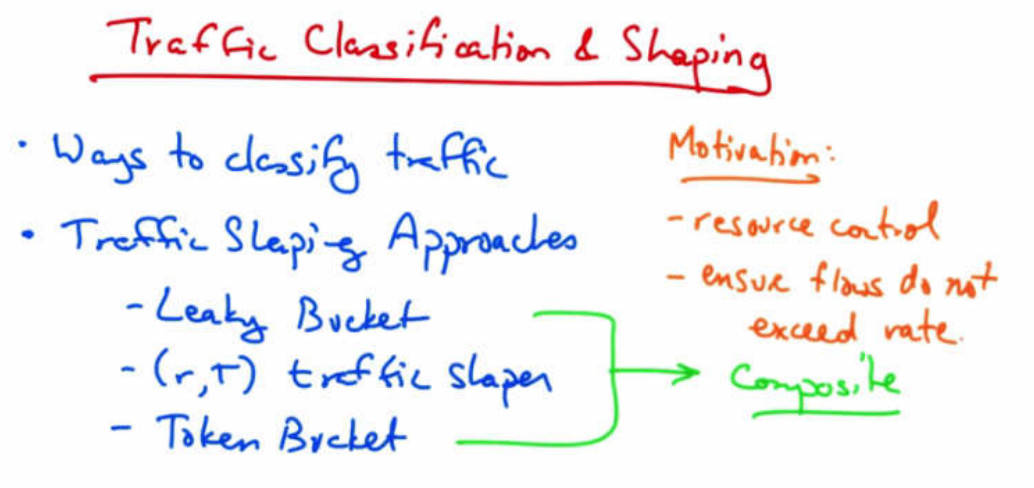
CS 6250: Computer Networking

Lecture 7: Rate Limiting & Traffic Shaping

Lesson 2 Intro

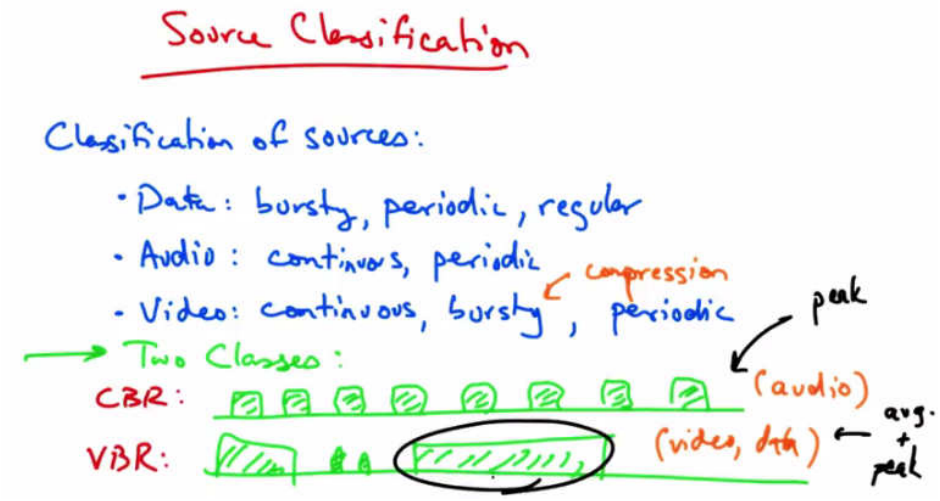
TCP congestion control responds to interpreted network events like packet loss, but does not provide a high-level control mechanism for congestion. In this section, we'll look at traffic shaping and network measurement, which are important tools for operating the network.

Traffic Classification and Shaping



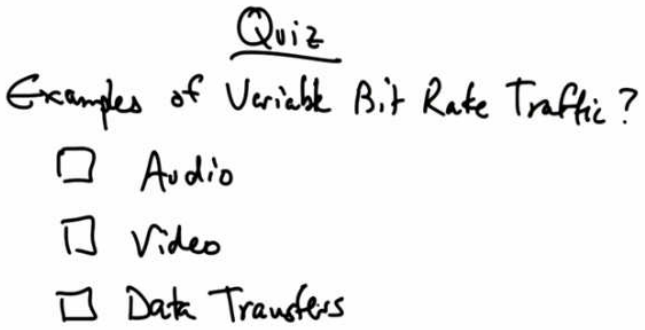
In this lesson we will talk about traffic classification and shaping. We'll first talk about different ways to classify traffic, then we'll talk about different traffic shaping approaches, then we'll talk about a traffic shaper called a leaky bucket traffic shaper, then we'll talk about an (r, T) traffic shaper. Then we'll talk about a token bucket traffic shaper, and finally, we'll talk about how to combine a token bucket shaper with a leaky bucket shaper to build what's called a composite shaper. The motivation here is to control network resources and ensure that no traffic flow exceeds a particular pre-specified rate.

Source Classification



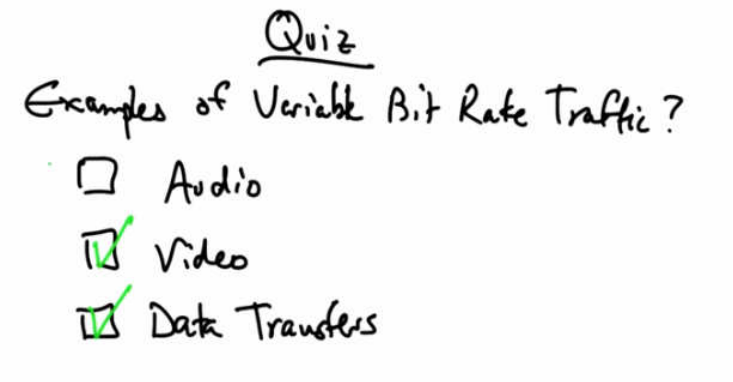
Traffic sources can be classified in a number of ways. Data traffic might be bursty, it might be weakly periodic, or it might also be regular. Audio traffic is typically continuous and strongly periodic. Video traffic is continuous, but it's often bursty due to the nature of how video is often compressed, as we saw in a previous lecture, and it may also be periodic. Typically, we think of taking these sources and classifying them into two kinds of traffic. One is a constant bit rate source or a CBR source. In a constant bit rate source of traffic, traffic arrives at regular intervals, and packets are typically the same size as they arrive, resulting in a constant bit rate of arrival. Audio is an example of a constant bit rate source. Many other sources of traffic are variable bit rate or VBR. Video and data are often variable bit rate. Typically when we shape CBR traffic, we shape it according to a peak rate. Variable bit rate traffic is shaped according to both an average rate, and a peak rate, where the average rate might actually be a small fraction of the peak rate. You can see that at certain times the peak rate might well exceed the average rate. Let's now talk about how to perform traffic shaping in a number of different ways.

VBR Quiz



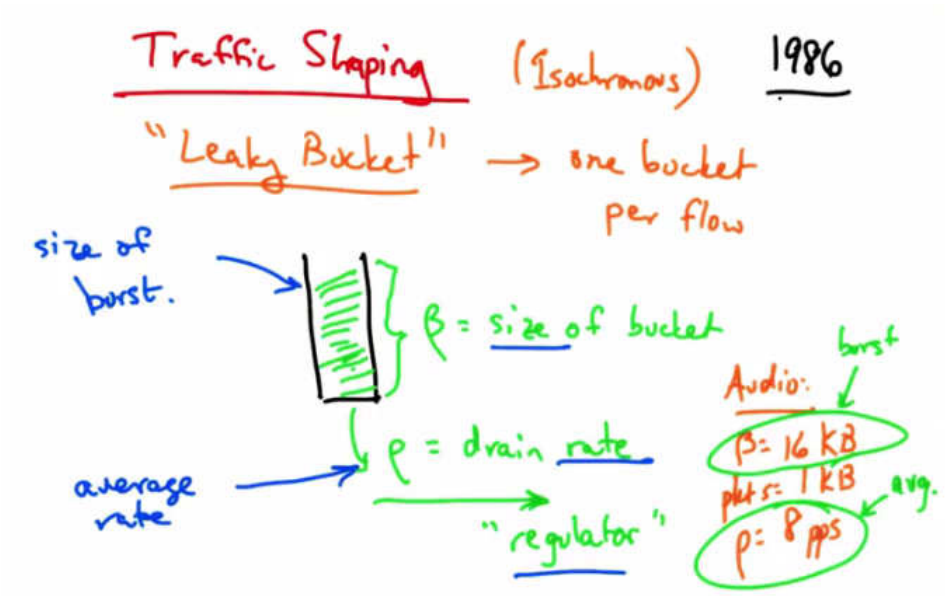
So what are some examples of variable bit rate traffic? Audio streams, Video streams, or Data Transfers? Please check all that apply.

VBR Solution



Video streams and data transfers can be variable bit rate or bursty. Audio tends to be constant bit rate with each packet being of a small, fixed packet size.

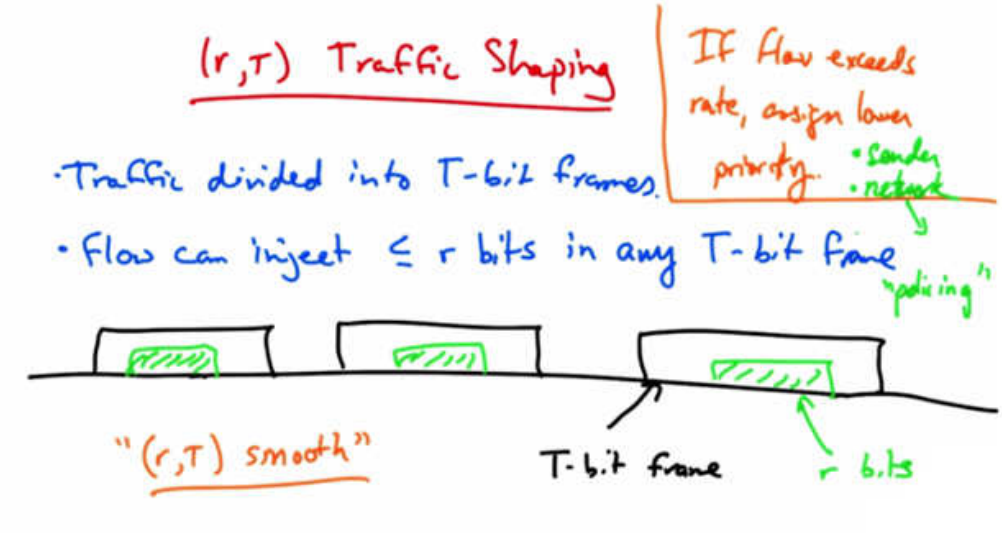
Leaky Bucket Traffic Shaping



One way of shaping traffic is with what's called a Leaky Bucket Traffic Shaper, where each flow has its own bucket. In a Leaky Bucket Traffic Shaper, data arrives in a bucket of size beta and drains from the bucket at rate rho. The parameter rho controls the average rate. Data can arrive faster or slower into the bucket, but it cannot drain at a rate faster than rho. Therefore, the maximum average rate that traffic can be sent is this smooth rate, rho. The size of the bucket controls the maximum burst size that a sender can send for a particular flow. So even though the average rate cannot exceed rho, at times, the sender might be able to send at a faster rate, as long as the total size of the burst does not exceed the size of the bucket. Or does not overflow the bucket. The leaky bucket allows flows to periodically burst, and the regulator at the bottom of the leaky bucket ensures that the average rate does not exceed the drain rate of the bucket. For

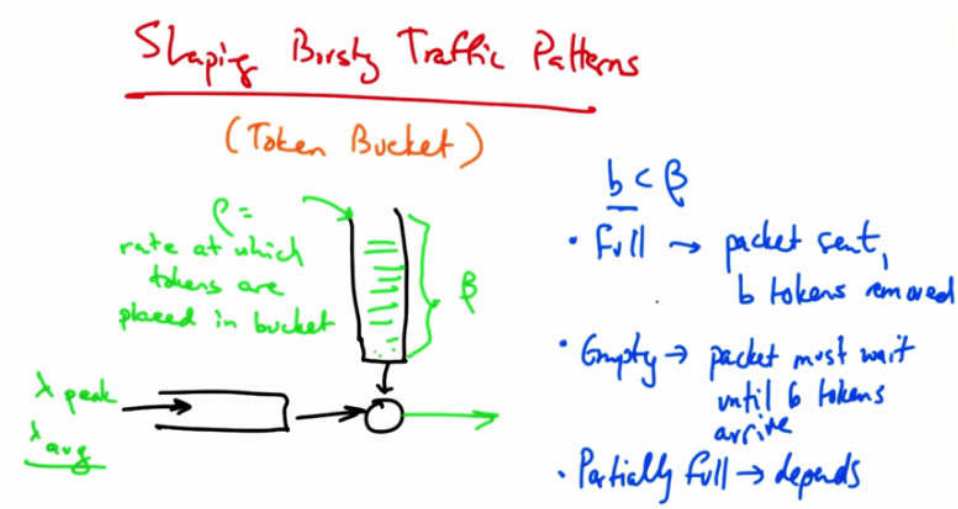
example, for an audio application one might consider setting the size of the bucket to be 16 kilobytes. So packets of one kilobyte would then be able to accumulate a burst of up to 16 packets in the bucket. The regulator's rate of eight packets per second, however, would ensure that the audio rate would be smooth to an average rate not to exceed 8 kilobytes per second or 64KBps. Setting a larger bucket size can accommodate a larger burst rate. Setting a larger value of rho can accommodate or enable a faster packet rate. The leaky bucket traffic shaper was developed in 1986 and soon to follow was a technique called (r, T) traffic shaping.

(r, T) Traffic Shaping



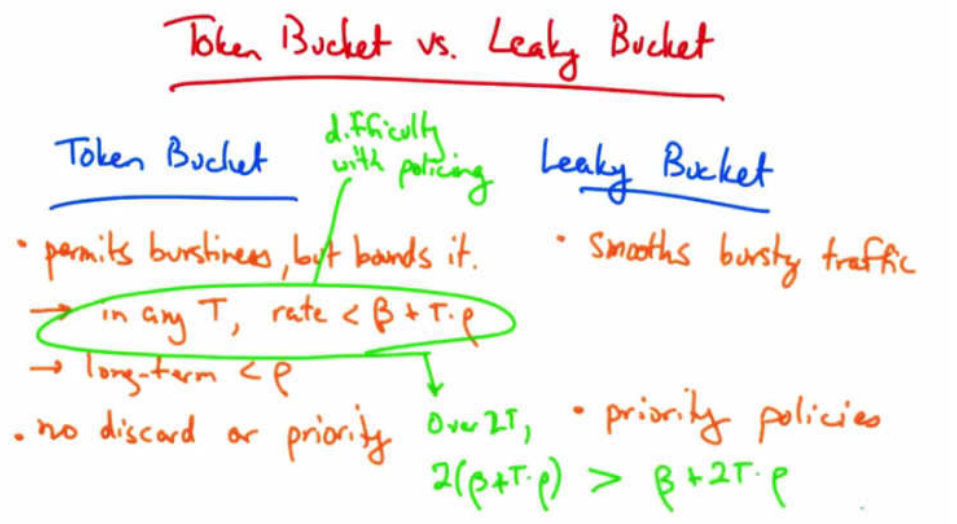
In (r, T) traffic shaping, traffic is divided into T-bit frames, and a flow can inject less than or equal to r bits in any T-bit frame. If the sender wants to send more than one packet of r bits, it simply has to wait until the next T-bit frame. A flow that obeys this rule has what is known as an (r, T) smooth traffic shape. In the case of (r, T) smooth traffic shaping, one cannot send a packet that's larger than r bits long. Unless T is very long, the maximum packet size may be very small. So the range of behaviors is typically limited to fixed rate flows. Variable flows have to request data rates that are equal to the peak rate, which is incredibly wasteful if you have to configure the shaper such that the average must support whatever peak rate the variable rate flow may send. The (r, T) traffic shaper is slightly relaxed from a simple leaky bucket because rather than sending one packet every time unit, the flow can send a certain number of bits every time unit. Now there's a question of what to do when a flow exceeds a particular rate. And typically what's done is that if a flow exceeds its rate, the excess packets in that flow are given a lower priority, and if the network is heavily loaded or congested, the packets from a flow that exceeds a rate may be preferentially dropped. Priorities might be assigned at the sender, or at the network. At the sender, the application may mark its own packet, since the application knows best which packets may be less important. In the network, the routers may mark packets with a lower priority, which is sometimes called policing.

Shaping Bursty Traffic Patterns



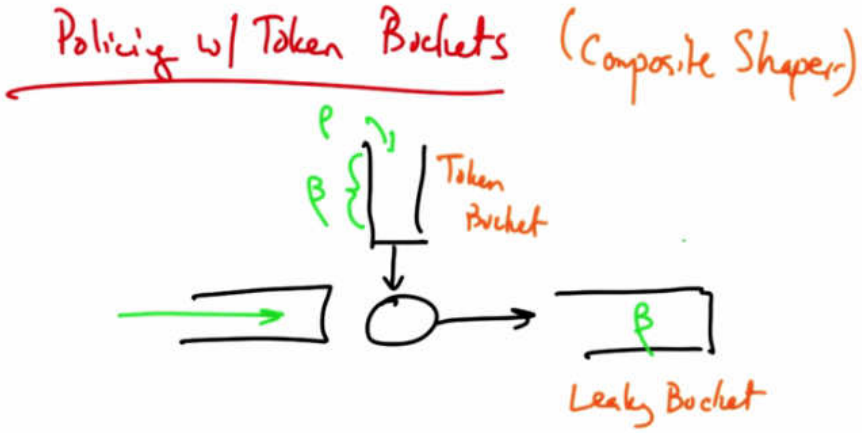
Sometimes we may want to shape bursty traffic patterns allowing for bursts to be sent on the network, but still ensuring that the flow does not exceed some average rate. For this we might use what's called a token bucket. In a token bucket, Tokens arrive in a bucket at a rate rho, and beta is again the capacity of the bucket. Now, traffic may arrive at an average rate lambda_average, and a peak rate lambda_peak. Traffic can be sent by the regulator as long as there are tokens in the bucket. To consider the difference between a token bucket and a leaky bucket, consider sending a packet of size b That's less than beta. If the token bucket is full, the packet is sent, and b tokens are removed. If the bucket is empty though, the packet must wait until b tokens drip into the bucket. If the bucket is partially full, well, then it depends. If the number of tokens in the bucket exceed little b, then the packet is sent immediately. Otherwise we have to wait until there are little b tokens in the bucket before we can send the packet.

Token Bucket vs Leaky Bucket



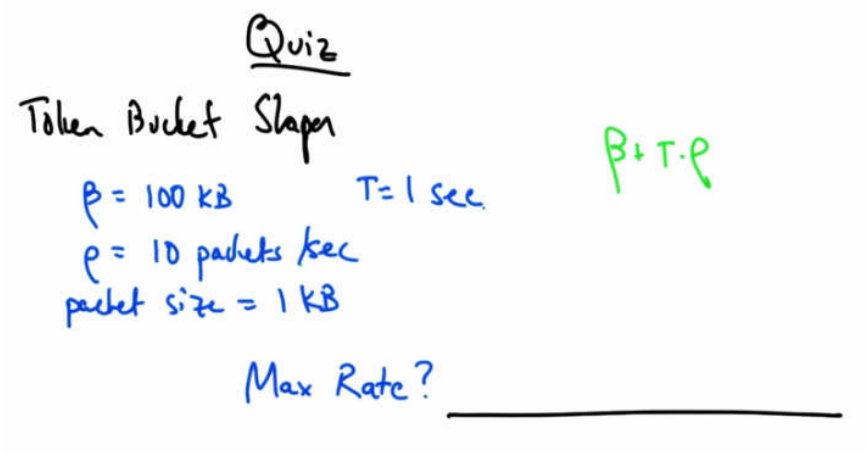
Let's compare the difference between a token bucket and a leaky bucket. The token bucket permits traffic to be bursty, but it bounds it by the rate rho. On the other hand, a leaky bucket simply forces the bursty traffic to be smoothed. The bound in a token bucket is as follows. If our bucket size is beta, then we know that in any interval T, then the rate is always less than beta, that is, the maximum number of tokens that can be accumulated in the bucket, plus the rate at which tokens accumulate, times that time interval. We also know that the long term rate will always be less than rho. Token buckets have no discard or priority policies, whereas leaky buckets typically implement priority policies for flows that exceed the smoothing rate. Both are relatively easy to implement, but the token bucket is a little bit more flexible since it has some additional parameters that we can use to configure burst size. One of the limitations of token buckets is the fact that in any traffic interval of length T, the flow can send beta plus T times rho tokens of data. If a network tries to police the flows by simply measuring their traffic over intervals of length T, the flow can cheat by sending this amount of data in each interval. Consider, for example, an interval of twice this length. Well, if the flow can send beta plus T times rho in each interval, then over 2T the flow can consume 2 times beta plus tau times rho tokens. But actually this is greater than how much the flow is actually supposed to be able to send which is beta plus 2T times rho. So policing traffic being sent by token buckets is actually rather difficult. So, token buckets allow for long bursts, and if the bursts are of high priority traffic, they are difficult to police and may interfere with other high priority traffic. So there's some need to limit how long a token bucket sender can monopolize the network.

Policing With Token Buckets



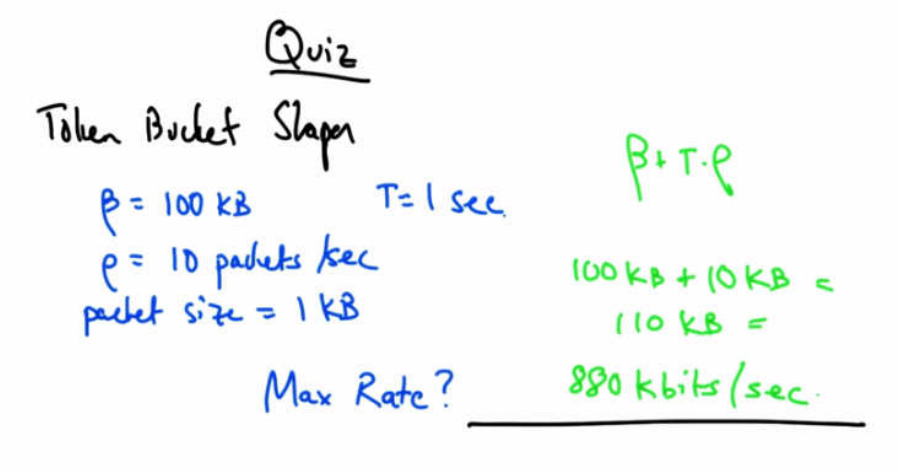
So, to apply policing to Token Buckets, what's often done is to use what's called a Composite Shaper, which is to combine a Token Bucket Shaper with a Leaky Bucket. The combination of the Token Bucket Shaper with the Leaky Bucket Shaper allows for good policing. Confirming that the flow's data rate does not exceed the average data rate allowed by the smooth Leaky Bucket is easy, but, the implementation is more complex since each flow now requires two counters and two timers. One timer and one counter for each bucket.

Token Bucket Shaper Quiz



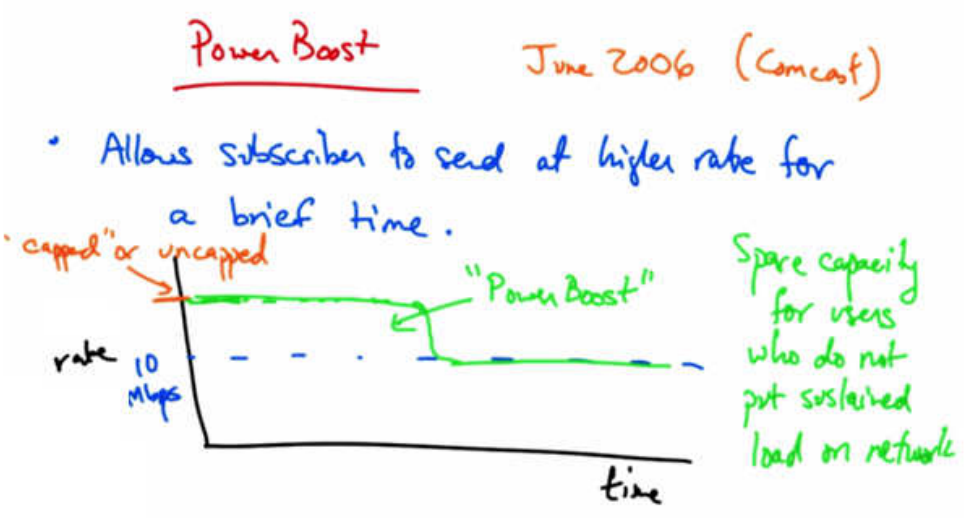
So as a quick quiz, suppose that we have a token bucket shaper, and suppose that the size of the bucket is 100 kilobytes, that rho is ten packets per second, and that packets are one kilobyte. Assume also that we are talking about an interval of one second. Remember than in any given interval, a flow can never send more than beta plus tau times rho bits of data. Please give your answer in kilobits per second. Keeping in mind that one byte is eight bits.

Token Bucket Shaper Solution

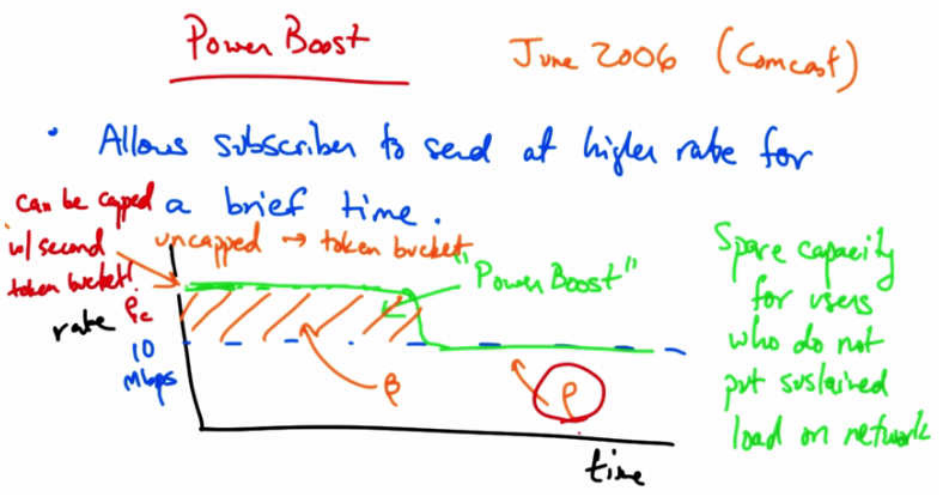


So the maximum rate would be 100 kilobytes times 1 second plus 10 packets per second times 10 kilobytes, or 110 kilobytes, which is 880 kilobits in one second.

Power Boost

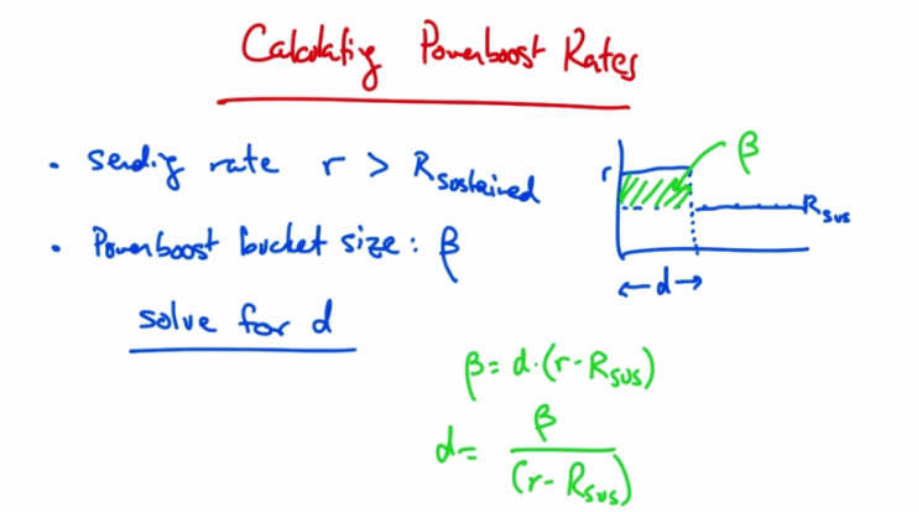


In this lesson we'll talk about Power Boost which is a traffic shaping mechanism that was first deployed in commercial broadband networks in June 2006 by Comcast. The Power Boost allows a subscriber to send at a higher rate for a brief period of time. So if you subscribed at a rate of ten megabits per second, then Power Boost might allow you to send at a higher rate for some period of time before being shaped back to the rate at which you were subscribed at. So, Power Boost targets the Spare Capacity in the network for use by subscribers who don't put a sustained load on the network. There are two types of Power Boosts. If the rate at which the user can achieve during this burst window is set to not exceed a particular rate. Then we say that the policy is capped Power Boost, otherwise the policy, or the shaping, is called uncapped Power Boost. Now in the uncapped setting, the configuration is simple and as we described in the last lesson. The area here is the Power Boost bucket size. That's the maximum amount of traffic that can be sent that exceeds the sustained rate. The maximum sustained traffic rate is simply Rho, as we've defined it before.



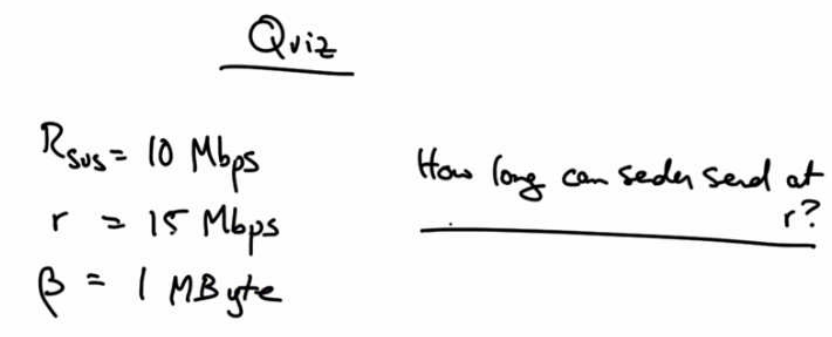
Now suppose that we wanted to cap the rate that the sender could send during the power boost window. Well all we need to do in that case is to simply apply a second token bucket with another value of Rho. That token bucket limits the peak sending rate for Power Boost eligible packets to the rate Rho C, where Rho C is larger than Rho. Remember that this value of Rho also affects how quickly tokens can refill in the bucket, so it also plays the role in the maximum rate that can be sustained during a power boost window.

Calculating Power Boost Rates



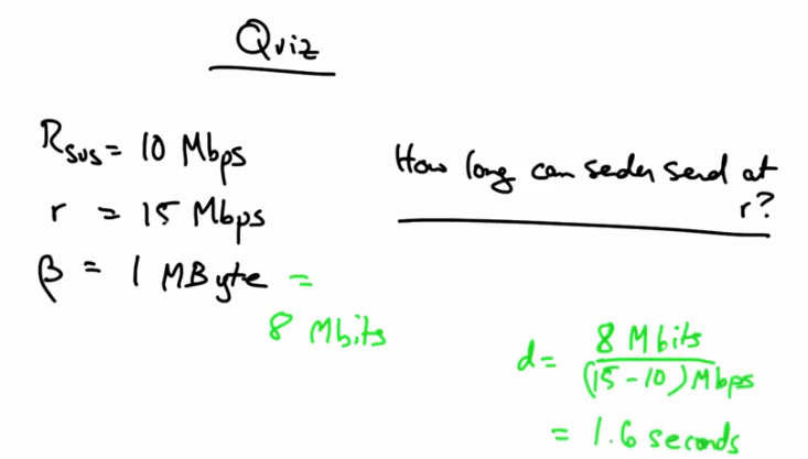
Suppose that a sender is sending at some rate r, which is bigger than the sustained rate R that they are allowed to be sending it, and suppose that our bucket size is beta. Then how long can a sender send at the rate r that exceeds the sustained rate? In other words, what is the value of d? We know that the bucket size, beta, as shown in the shaded green area, is simply d times r minus the sustained rate R. So a sender can send at the rate, little r, that exceeds the sustained rate, R, for beta divided by r minus R sustained. Now based on what we've learned here, let's just take a quick quiz.

Power Boost Quiz



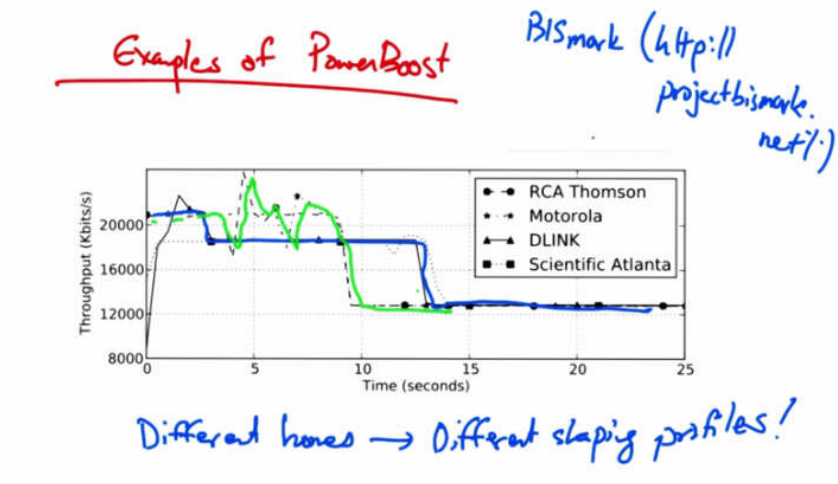
Suppose that the sustained rate that a subscriber subscribes to is ten megabits per second, but they like to burst at a rate of fifteen megabits per second. Suppose that the bucket size is one megabyte or eight megabits. How long can the sender send at the higher rate? Please give your answer in decimal form in seconds.

Powerboost Solution



One megabyte is eight megabits, and from our previous calculation, we know that the duration should be eight megabits, over five megabits per second, or 1.6 seconds.

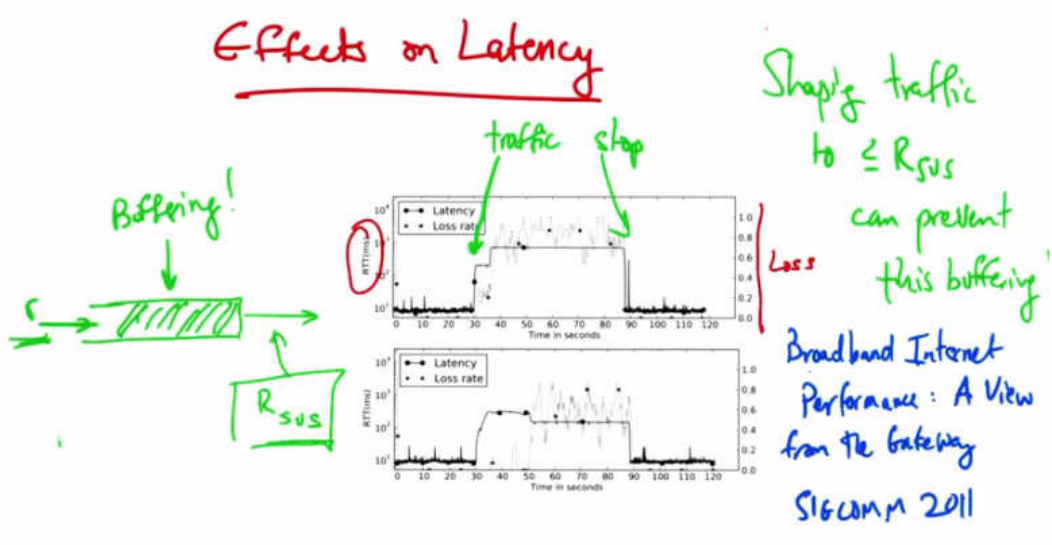
Examples of Powerboost



In the Bismark Project at Georgia Tech, which you can go check out at http://projectbismark.net, we've done some measurements of Comcast Power Boosts in different home networks. Here are some real world examples of Power Boost's traffic shaping profile in four different home networks, each with a different cable modem as shown in the caption. You can see that different homes exhibit different shaping profiles. Some have a very steady pattern whereas others have a

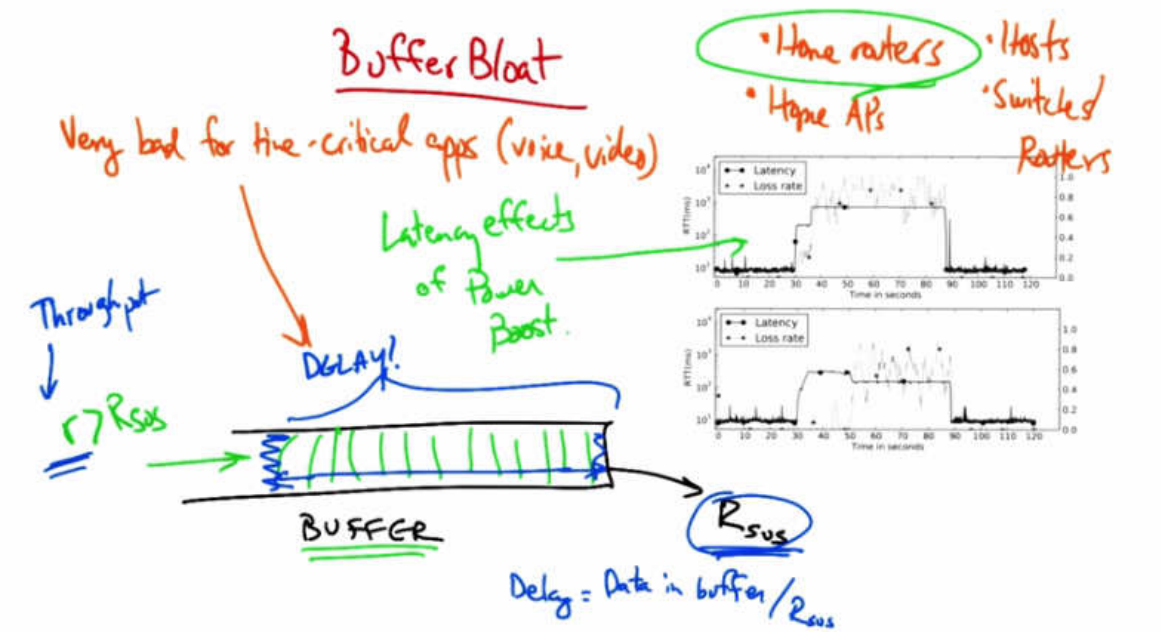
more erratic pattern. Interestingly you can see in some cases that there appear to be two different tiers of higher throughput rates.

Effects on Latency



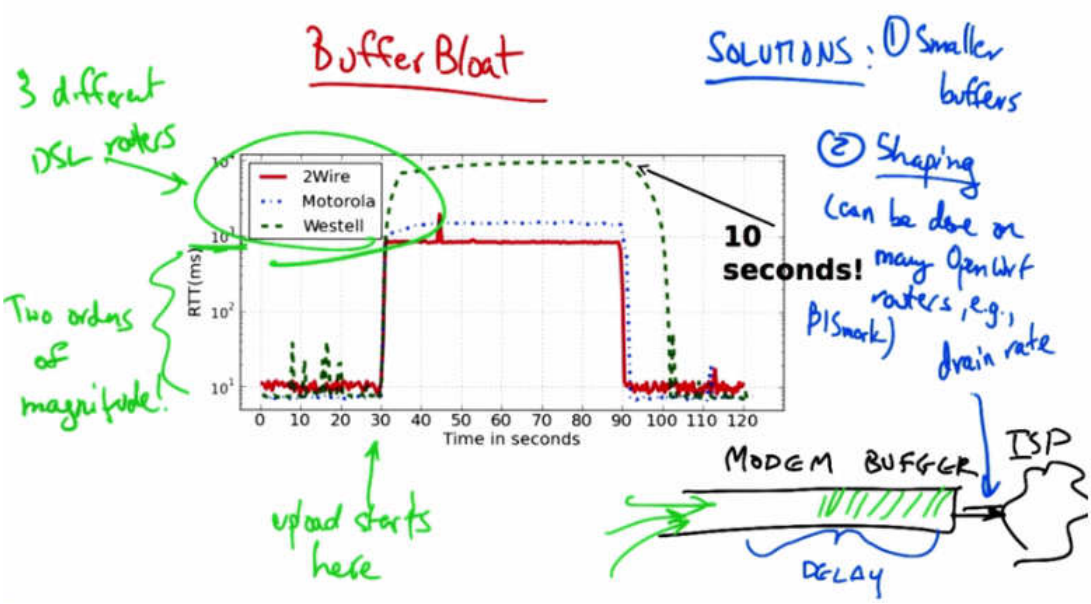
Power Boost also has effect on the latency that uses perceive, as well as the loss rate. Here we've shown Power Boost latency effect for two different users. The latency is shown in terms of round trip time in milliseconds and the loss rates are shown on the right side of the Y axis. Latencies are also shown in a log scale. In this particular experiment, we start sending traffic here and we stop sending traffic here, in both cases. We can see that, even though power boost allows you to just send at a higher traffic rate, actually users may experience high latency and loss over the duration that they're sending at a higher rate. The reason for this is that the access link may not be able to support the higher rate. So if a sender can only send at R sustained for an extended period of time but is allowed to burst at a rate r for some shorter period of time, then buffers may fill up and the resulting buffers may introduce additional delays in the network since packets are being buffered up rather than dropped. TCP senders can continue to send at higher rates, such as little r, without seeing any packet loss even though the access link may not be able to send at that higher rate. As a result, packets buffer up and users see higher latency over the course of the power boost interval. To solve this problem, you might imagine instead that a sender might shape it's rate never to exceed the sustained rate, big R. If it did this, then it could avoid seeing these latency effects. So, certain senders who are more interested in keeping latency under control than they are in sending at bursty volumes, may wish to run a traffic shaper in front of a power boost enabled link. So shaping traffic to a rate of less than this sustained rate R can prevent this buffering. More details about power boost and the experiments that we've run to keep latency under control in its presence are available in a paper we wrote called Broadband Internet Performance, A View from the Gateway. Which appeared in Sigcomm 2011.

Buffer Bloat



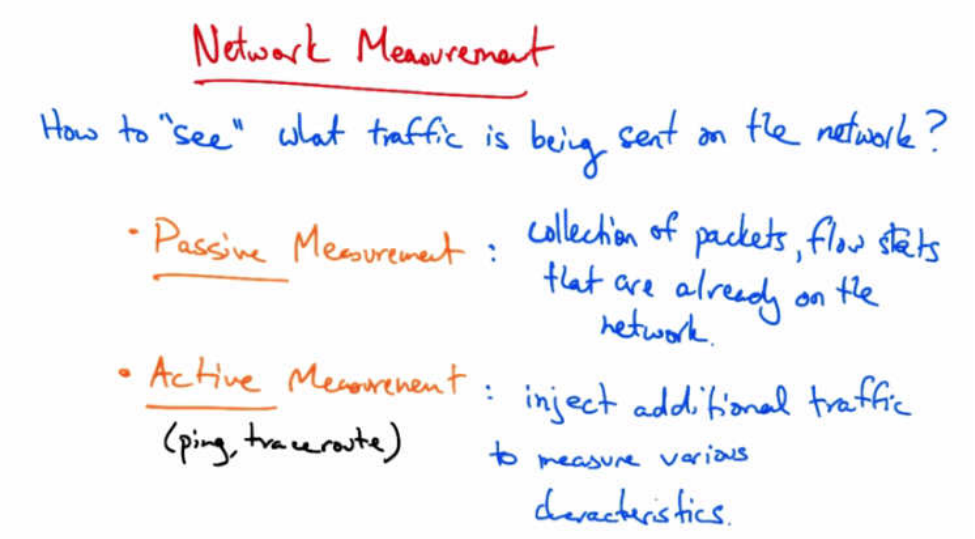
In this lesson, we will talk just briefly about buffer bloat. We saw an example of buffer bloat in the last lesson where we explored the latency effects of power boost. In the example we explored, the sender could send at a rate r that was bigger than the sustained rate R without seeing packet loss. Now if there's a buffer in the network that can support this higher rate, what we'll see is that buffer will start filling up with packets. But this buffer can still only drain at the sustained rate R. So even though the sender might be able to send at a faster rate for a brief period of time in terms of throughput, all of those packets that the sender sent at that faster rate are queued up in line waiting to be sent. As these packets are waiting in this buffer, they'll see higher delays than they would see if they simply arrived at the front of the queue and could be sent immediately. The delay that the packet will see in the buffer is the amount of data in the buffer divided by the rate that the buffer can drain. These large buffers can introduce delays that ruin the performance for time-critical applications such as voice and video. These large buffers actually show up all over the place: in home routers, in home WiFi devices or access points, in hosts on device drivers, and also in switches and routers. Let's take an example of buffer bloat that we observed in home routers as part of the Bismarck study that I described in the last lesson.

Buffer Bloat Example



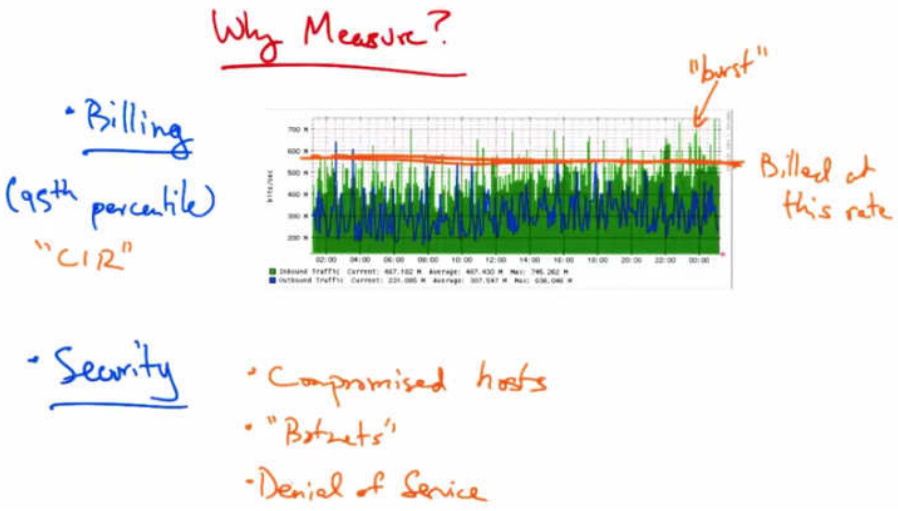
In the example we've shown here, we have three different DSL routers. The y axis shows the round trip time, or the latency to a nearby server in milliseconds, and is again shown in a log scale. We started an upload at the time 30 seconds shown on the plot. Now you can see that different modems experience a huge increase in latency when we start this upload. Some of them experience a latency of as much as one second, up from a typical latency of about ten milliseconds. One particular modem saw a round trip latency of as high as ten seconds during uploads. Now to remind you what's going on here, is that the modem itself has a buffer. Your ISP may be upstream of that buffer, and your access link may be draining that buffer at a certain rate. TCP senders in your home will send until they see lost packets, but if the buffer's large, the senders won't actually see those lost packets until this buffer has already filled up. The senders continue to send at increasingly faster rates until they see a loss. As a result, packets that are arriving at this buffer see increasing delays, and senders continue to send at faster rates, because without packet loss they don't have a signal to slow down. There's several solutions to the buffer bloat problem. One is obviously to use smaller buffers, but given that we have a lot of deployed infrastructure, simply reducing the buffer size in deployed routers, modems, switches, home Wi-Fi devices, and so forth, is a tall order. The other thing that we can do is to use the traffic shaping methods that we have learned about. Consider that the buffer drains at a particular rate, which in this case is the rate of the uplink to the ISP. If we shape traffic such that traffic coming into the access link never exceeds the uplink that the ISP has provided us, then the buffer will never fill. Thus, by shaping traffic at the home router such that the rate that traffic is sent to the ISP never exceeds the rate of the uplink, the modem buffer will never actually fill up. This type of shaping can be done on many open WRT capable routers, including the Bismark routers that we've developed here at Georgia Tech.

Network Measurement



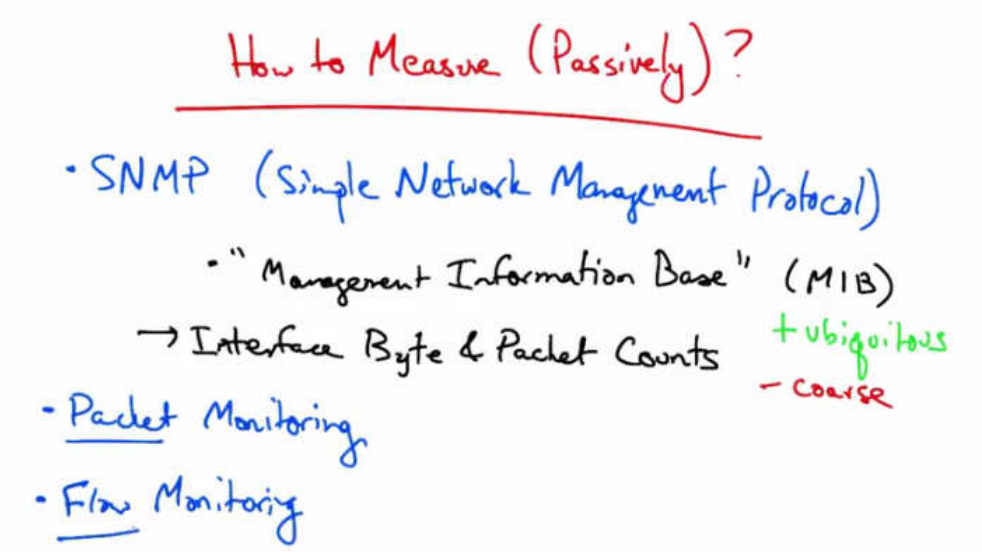
In this lesson we'll be talking about network measurement, or how to see what traffic is being sent on the network. There are two types of network measurement. One is passive measurement. In passive measurement we collect packets, flow statistics, and so forth, of traffic that is already being sent on the network. So, this might include packet traces, flow statistics, or application level logs. In active measurement, we inject additional traffic into the network to measure various characteristics of the network. So we've seen some examples of active measurement already, such as in the previous lessons where we actively sent traffic on the network to measure speeds of downloads. Other common active measurement tools include those such as ping, and traceroute. Ping is often used to measure the delay to a particular server, and traceroute is often used to measure the network level, or the IP level path between two hosts on the network.

Why Measure



So why do we want to measure the traffic on the network? One reason might be billing. So for example, we might want to charge a customer based on how much traffic they've sent on a network. In order to do so, we need to passively measure how much traffic that customer is sending. Here's an example of measurements of inbound and outbound traffic volumes on a link on the Georgia Tech campus network. The Y axis is shown in bits per second, and the X axis is the time of day. Now, a user might be billed based on how much traffic they send on the network. A common mode of billing is called 95th Percentile billing, where a customer pays for what's called a committed information rate, or CIR, and throughput is measured every five minutes. The customer, then, may be billed on the 95th percentile of these five minute samples. So if we were to bill on the 95th percentile of inbound traffic, we might approximate that 95th percentile by the orange line I've drawn here. And the customer might be billed at this rate, even though they're allowed to sometimes burst at higher rates. Another common reason to measure is security. For example, network operators may want to know the type of traffic that's being sent on the network so they can detect rogue behavior. A network operator may want to measure traffic on the network to detect compromised hosts or the presence of Botnets or Denial of Service attacks, two phenomena that we'll talk about later on in the course. For the rest of this lesson, since we focused a lot on performance measurement already, I will mainly focus on passive traffic data measurement.

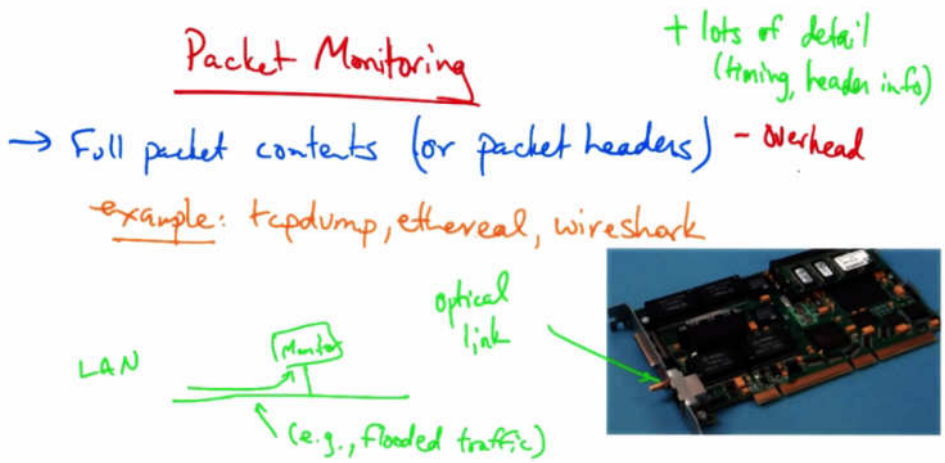
How to Measure (Passively)



Let's talk about how to perform passive Network Traffic Management. One way to do this is using the Packet and Byte Counters provided by the Simple Network Management Protocol. Many network devices provide what's called a Management Information Base, or a MIB that can be polled or queried for particular information. One common use for SNMP is to poll a particular Interface on a Network Device for the number of Bytes or Packets that it sent. By periodically polling, we can then determine the rate at which Traffic is being sent on a link by simply taking the difference in these Packet and Byte Counters over particular intervals. The advantage of SNMP is that it's fairly ubiquitous. It's supported on essentially all Networking Equipment and

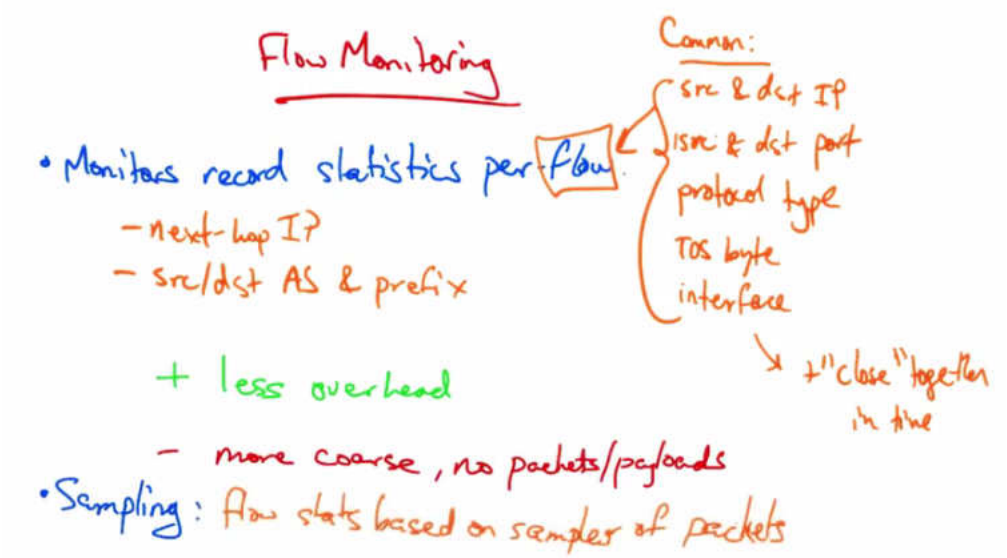
there are many products for polling and analyzing SNMP data. On the other hand, it's fairly coarse and you cannot express complex queries on the data. It's coarse in the sense that because we are just polling Byte or Packet Counts on the Interface, we can't ask specific questions such as how much traffic has been sent by a particular host or by a particular flow. Two other ways to measure passively are by monitoring at a packet granularity, whereby monitors can see full packet contents or at least headers, or at a flow level where a monitor may see specific statistics about individual flows in the network. Let's now talk a little bit about packet and Flow Monitoring.

Packet Monitoring



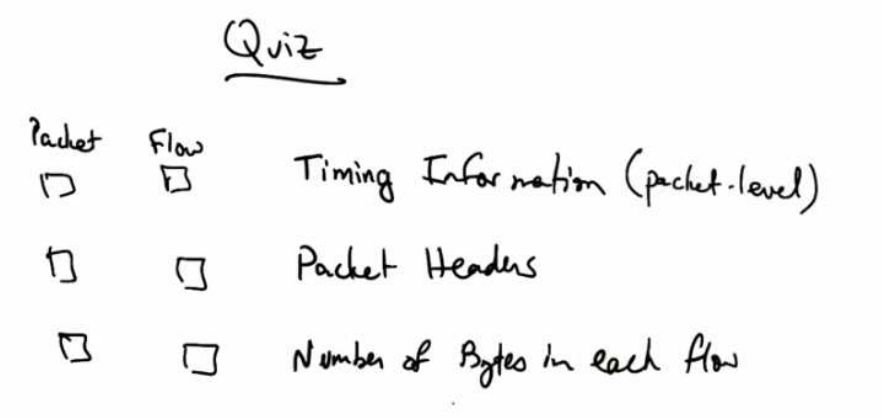
So in packet monitoring, a monitor might see the full packet contents, or at least the packet headers that traverse a particular link. Common ways of performing packet monitoring that you may have tried yourself include tcpdump, ethereal, or wireshark. And in some of the exercises, you'll get a chance to explore packet monitoring with one of these tools. Sometimes packet monitoring is performed using expensive hardware that can be mounted in servers alongside the router that forward traffic through the network. In these cases, an optical link in the network is sometimes split so that traffic can be both sent along the network and sent to the monitor. Even though packet monitoring sometimes requires this expensive hardware on very high speed links, what you do when you run tcpdump or wireshark or ethereal is essentially the same thing. Your machine acts as a monitor on the local area network. And if any packets happen to be sent towards your network interface, the network interface records those packets. Now on a switch network, you wouldn't see many packets that weren't destined for your own mac address. But on a network where there's a lot of traffic being flooded, you might see quite a bit more traffic destined for an interface that you're using to monitor. So the advantages of packet monitoring is that it provides lots of detail. You can see timing information and information in the packet headers. Unfortunately, a disadvantage is that it's fairly high overhead. It is very hard to keep up with high speed links and often requires a separate monitoring device such as the monitoring card that we've shown here. What if we are happy with a little less detail than packet monitoring can provide, but we can't afford its overhead? In that case, there is actually another approach that we can use called flow monitoring.

Flow Monitoring



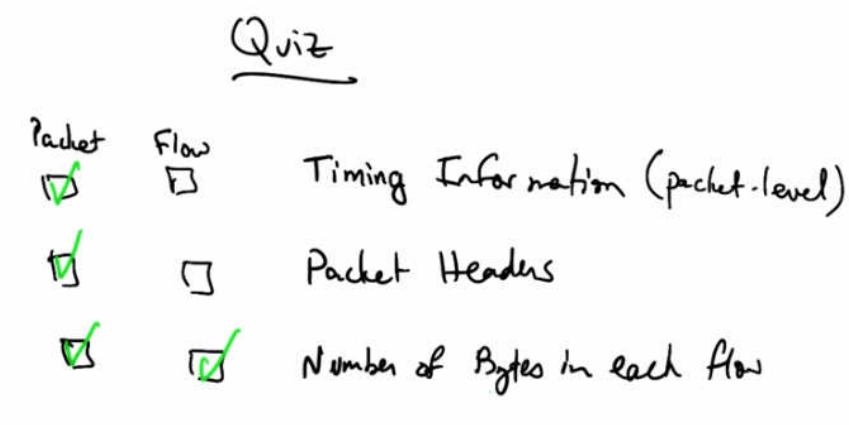
In flow monitoring, a monitor which might actually be running on the router itself, records statistics per-flow. A flow consists of packets that share a common source and destination IP address, source and destination port, protocol type, TOS byte, and interface on which the packets arrive. A flow monitor can then record statistics for a flow that's defined by the group of packets that share these features. The flow records may also contain additional information, such as the next hop IP address and other information related to routing, such as the source and destination AS on which those packets appear to be coming from and going to, based on the routing tables, as well as the prefix that those packets matched in the routing table. Flow monitoring is much less overhead than packet monitoring, but it's also much more coarse than packet monitoring because the monitor does not see individual packets or payloads. Therefore, it's impossible to get certain information from flow monitoring such as packet timing information. In addition to grouping packets into flows based on the fact that they share common elements in their headers, typically packets are grouped into flows if they occur close together in time. So, for example, if packets that share common sets of header fields do not appear for a particular time interval, such as 15 or 30 seconds, the router simply declares the flow to be over, and sends a flow record to the monitor based on the group of packets that it's seen up to that point. Sometimes, to reduce monitoring overhead, flow level monitoring may also be accompanied with sampling. Sampling builds flow statistics based only on samples of the packets. So, for example, flows may be created based on one out of every ten or 100 packets, or a packet might be sampled with a particular probability and flow statistics might only be tabulated based on the packets that end up being sampled randomly from the total set of packets.

Passive Traffic Monitoring Quiz



So as a quick review, which of the following passive traffic monitoring methods, packet or flow sampling, can provide the following information? Timing information about packets, packet headers, and the number of bytes that each flow sends? Please check all boxes that apply.

Passive Traffic Monitoring Solution



Only packet monitoring can provide timing information on a packet level, or packet headers. But both methods can actually provide the number of bytes in each flow. By definition, flow records record the number of bytes in each flow as an aggregate statistic, but if you had packet-level information, you could, of course, compute the statistic yourself.

CS 6250: Computer Networking

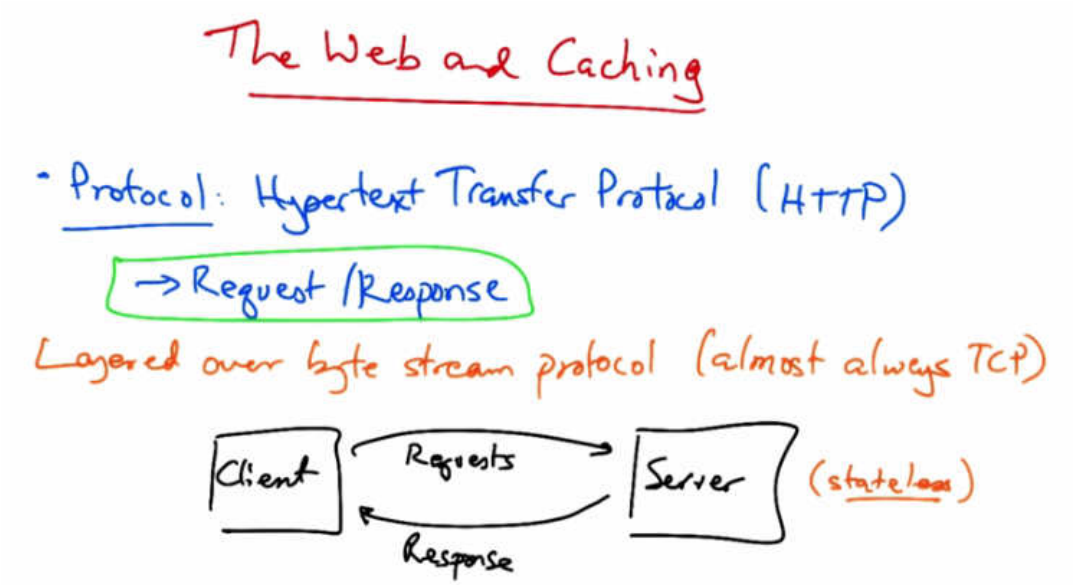
Lecture 8: Content Distribution

Lesson 3 Intro

We've covered congestion control, which works within network protocols; traffic shaping, which is a high level network tool; and now we'll look at content distribution, an internet-wide tool that enables websites and network operators to deliver data quickly and efficiently.

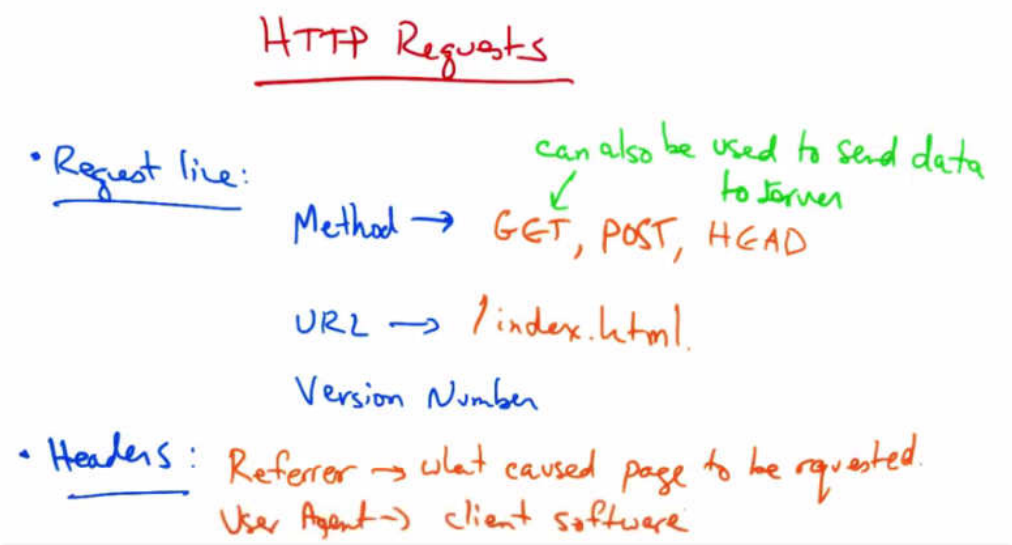
And to wrap up this section of the course, your project will export TCP in its slow start state.

The Web and Caching



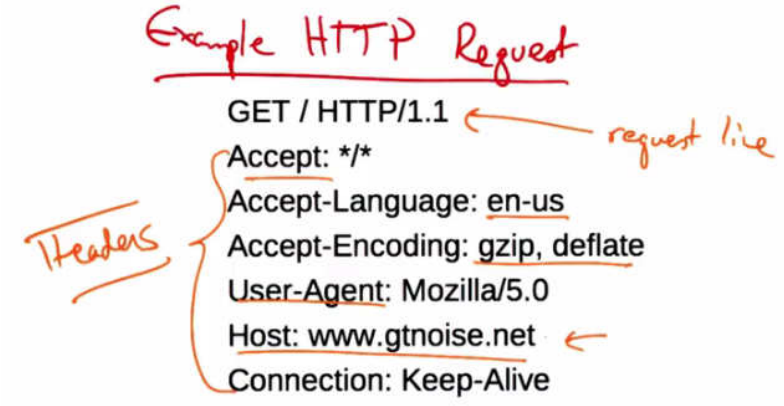
In this lesson, we'll talk about the web and how web caches can improve web performance. We will study, in particular, the hyper-text transfer protocol, or HTTP, which is an application layer protocol to transfer web content. It's the protocol that your web browser uses to request web pages, and it's also the protocol that the responses (or the web pages, or the objects that are returned as part of a webpage) are returned to your browser. Your web browser makes requests for web pages, and the pages and the objects in the page come back as responses. HTTP is typically layered on top of a byte stream protocol, which is almost always TCP. The client sends a request to a server asking for web content and the server responds with the content often encoded in text. The server maintains no information about past client requests. Thus we say the server is stateless. Let's take a quick look into the format of HTTP requests, and responses.

HTTP Requests



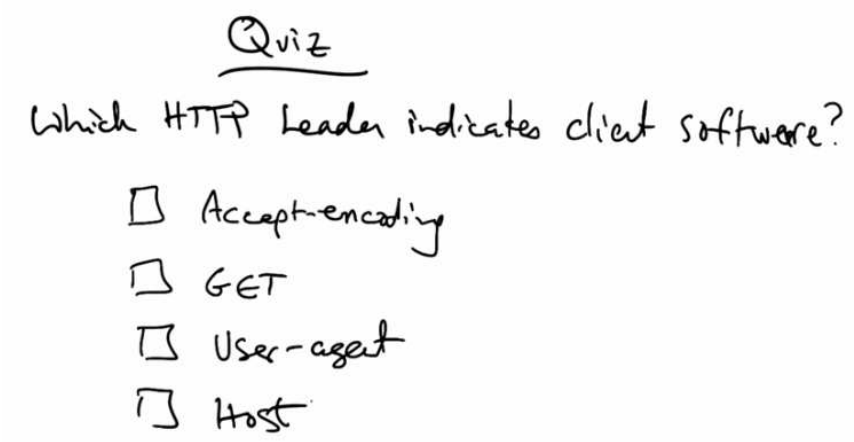
Let's first take a look at the contents of an HTTP Request. First there's the Request Line which typically indicates first, a method of request, where typical methods get to return the content associated with the URL; a Post, which sends data to the server; and a Head Request which returns, typically, only the headers of the Get Response, but not the content. It's worth noting that a Get Request can also be used to send data from the content to the server. The request line also includes the URL, which is relative, and may be something like index.HTML, and it also includes the version number of the HTTP protocol. The request also contains additional headers, many of which are optional. These include the referrer, which indicates the URL that caused the page to be requested. For example, if an object is being requested as part of embedded content in another page, the referrer might be the page that's embedding the content. Another example header is the user agent, which is the client software that's being used to fetch the page. For example, you might fetch a page using a particular version of Chrome or Firefox, and the user agent informs the server which client software is being used.

Example HTTP Request



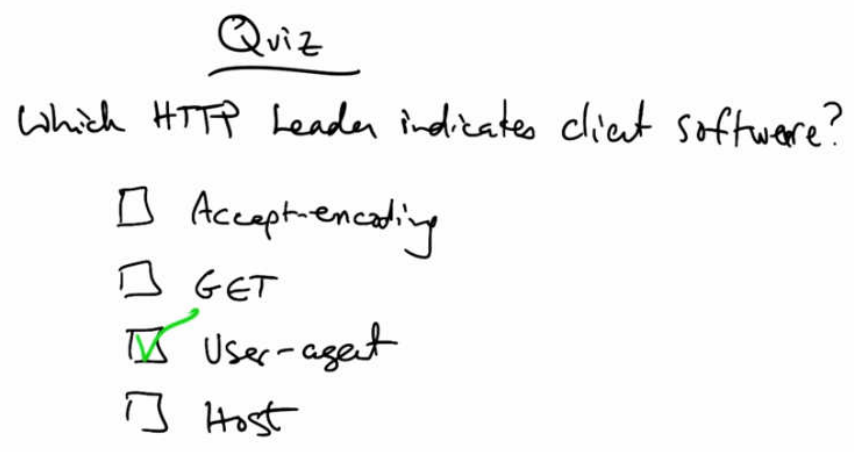
Let's take a look at an example HTTP request now. You can see here the request line, and here are some headers. Accept indicates that the client's willing to accept any content type, that it would like the content to be returned in English, that it can accept pages that are encoded in particular compression formats. We talked about the user agents. So in this case, it's a Mozilla 5.0 browser. Here's the host that the request is being made to. This is particularly useful in cases where a particular web server IP address might be hosting multiple websites on the same server.

HTTP Header Quiz



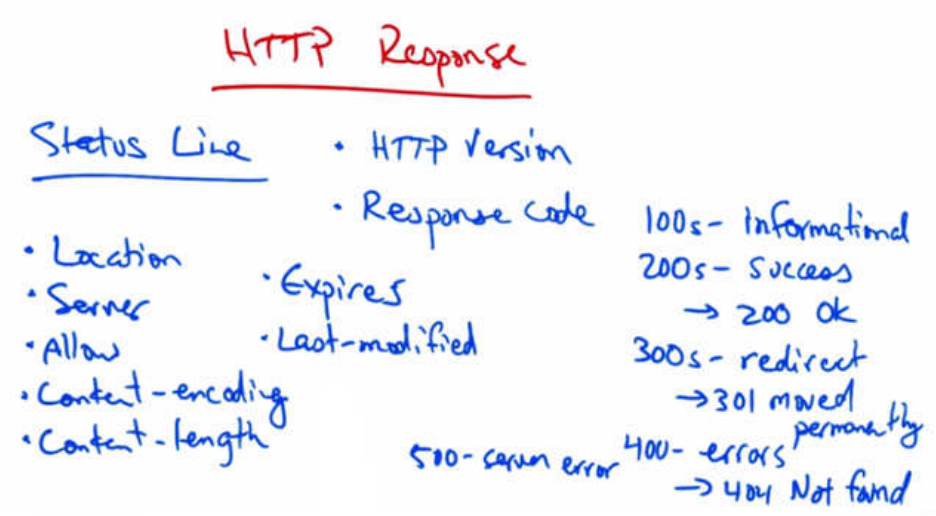
As a quick quiz, which HTTP header field indicates the client software that's being used to make the request? Accept encoding, get, user-agent, or host? Please pick the best answer.

HTTP Header Solution



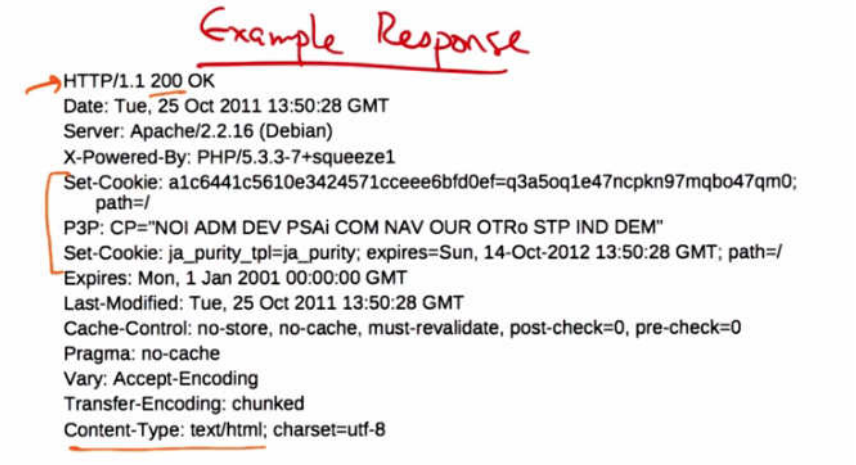
The user agent field in the HTTP request header indicates the client software that's being used to make the request.

HTTP Response



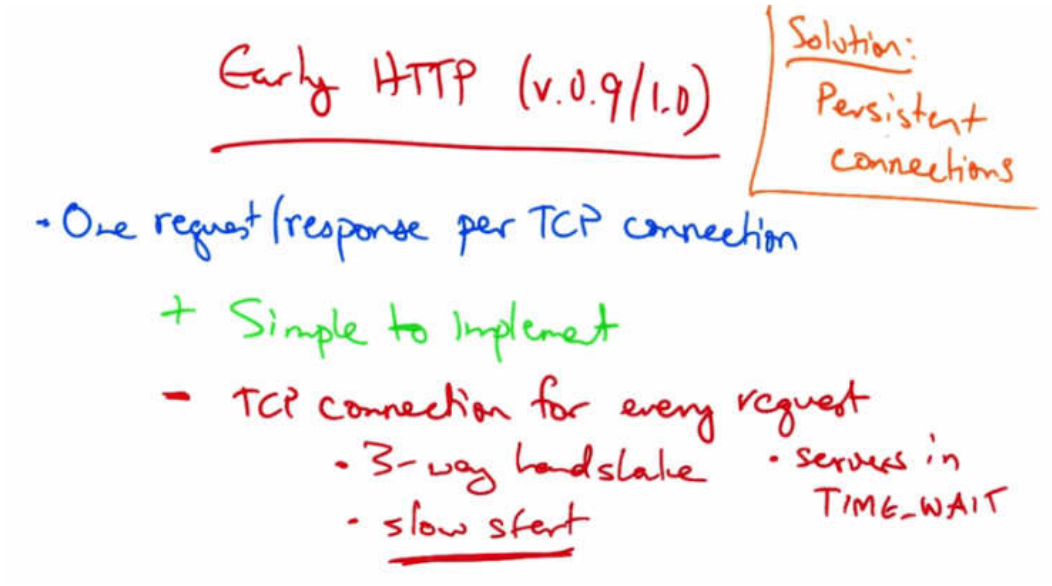
Let's now take a look at the anatomy of an HTTP response. A response includes a status line, which includes the HTTP version, and a response code, a where the response code may indicate a number of possible outcomes. 100 response codes are typically informational, 200s indicate success. So an example 200 response code is a common server response that indicates okay. 300 response codes indicate redirection. For example, a 301 response code indicates that the page has moved permanently. 400s are errors, a well known one being 404, which is not found, and 500s indicate server errors. Other headers include the location, which may be used for redirection; a server, which indicates server software; allow, which indicates the HTTP methods that are allowed, such as get, head and so forth; content-encoding, which describes how the content is encoded (for example, if it's compressed); content length, which indicates how long the content is in terms of bytes; expires, which indicates how long the content can be cached; and last-modified, which indicates the last time the page was modified.

Example Response



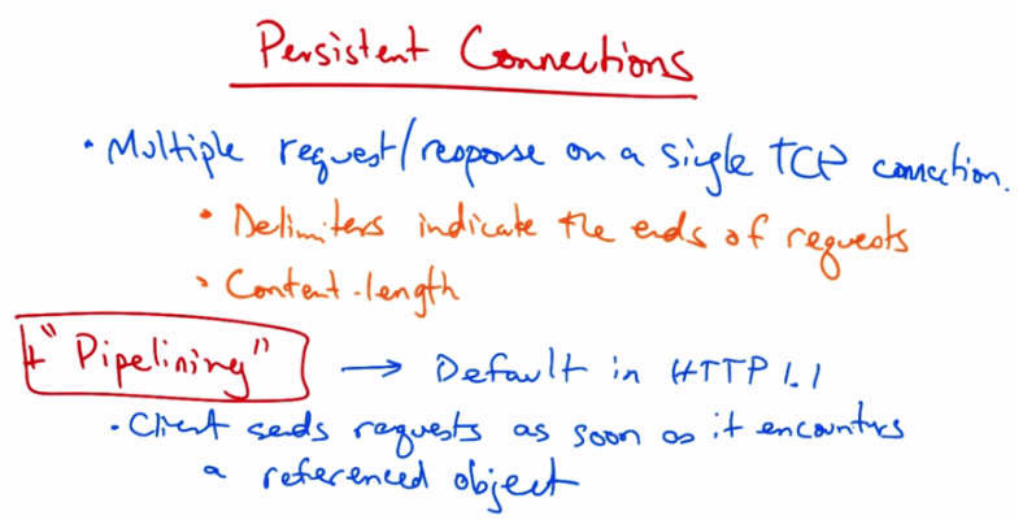
Let's take a quick look at an example response. Here is the status line indicating the HTTP version number and a response code, OK; the date the response was sent; the server that served the request; some cookies that are used to set some state on the client (for example, whether or not the client is logged in or not); when the page expires; when it was last modified; and some more instructions about how the page can or cannot be cached. There's also a content type header to indicate that the response is coming back in HTML format.

Early HTTP



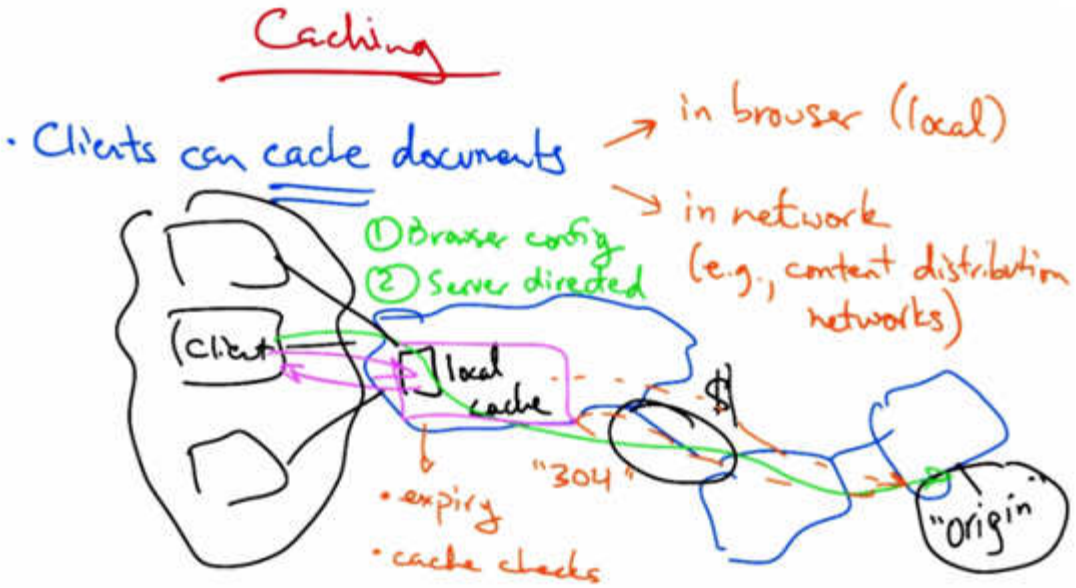
Now, early versions of HTTP actually only had one request or response for every TCP connection. On the plus side, this is simple to implement. But the main drawback is that it requires a TCP connection for every request, thereby introducing a lot of overhead and slowing transfer. First of all, we need a TCP three-way handshake for every request, and TCP must start in slow start every time the connection opens. This is exacerbated by the fact that short transfers are very bad for TCP because TCP is always stuck in slow start and never gets a chance to actually ramp up to steady state transfer. Also, as TCP connections are terminated after every request is completed, the servers have many connections that are forced to keep TCP connections in time-wait states until the timers expire, thus resulting in additional resources that the server needs to keep reserved even after the connections have completed. So a solution to increase efficiency and account for many of these drawbacks is to use something called persistent connections.

Persistent Connections

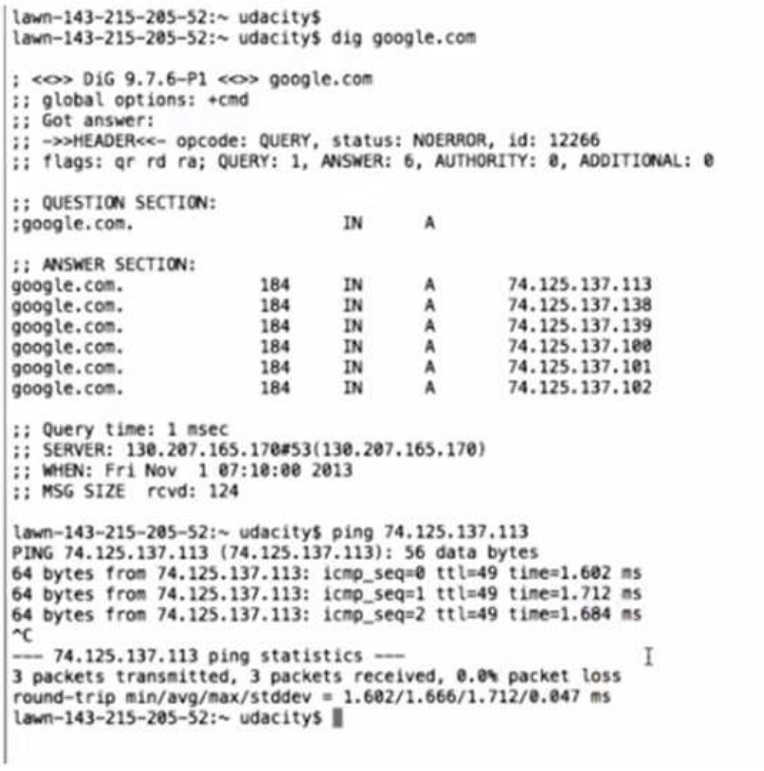


In persistent connections, multiple HTTP requests and responses are multiplex onto a single TCP connection. Delimiters at the end of an HTTP request indicates the end of a request and the content length allows the receiver to identify how long a response is. So, the server actually has to know the size of the transfer in advance. Persistent connections can also be combined with something call pipelining. In pipelining, a client sends the next request as soon as it encounters a referenced object. So there's as little as one round trip time for all referenced objects before they began to be fetched. Persistent connections with pipelining is the default behavior in HTTP 1.1.

Caching

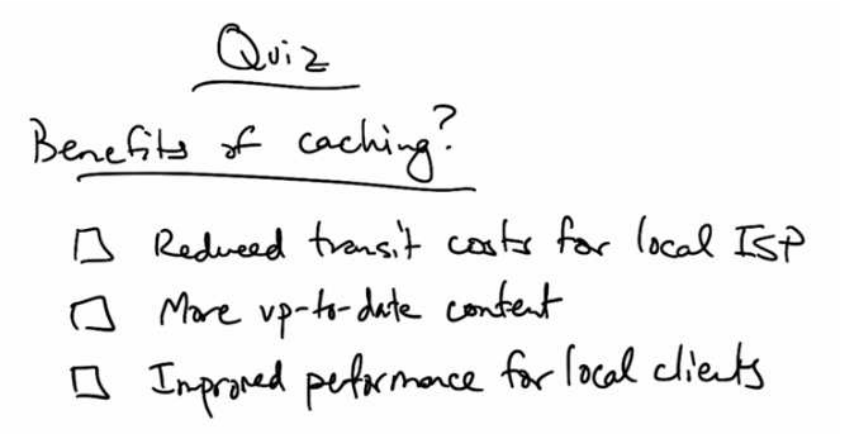


To improve performance, clients often cache parts of a webpage. Caching can occur in multiple places. Your browser can cache some objects locally on your very machine. Caches can also be deployed in the network. Sometimes your local ISP may have a web cache, and later we'll also look at how content distribution networks are a special type of web cache that can be used to improve performance. To see how caching can improve performance, consider the case where the origin web server may host the content for a particular website, but it's particularly far away. Now, we already know that TCP throughput is inversely proportional to round-trip times. So, the further away that this web content is, the slower the web page will load, both because latency is bigge, and because throughput is lower. If, instead, the client could fetch content from the local cache, performance could be drastically improved by fetching content from a more nearby location. Caching can also improve the performance when multiple clients are requesting the same content. In this case, not only do all of the local clients benefit from the content being cached locally, but the ISP also saves costs on transit, because it doesn't have to pay to keep transferring the same content over these expensive links. Instead, it can simply serve the content to the clients locally. To ensure that clients are seeing the most recent version of a page, caches periodically expire content, based on the expire setter that we already saw. Caches can also check with the origin server to see whether the original content has been modified. If the content has not been modified, the origin server would respond to a cache check request with a 304, or a not modified, response. Clients can be directed to a cache in multiple ways. One is with browser configuration. So you can open your browser and explicitly configure the browser to point to a local cache so that all HTTP requests first are directed to the local cache before the request is forwarded to the origin. In the second approach, the origin server, or the service hosting the content, might actually direct your browser to a cache. This can be done with a special reply to a DNS request.



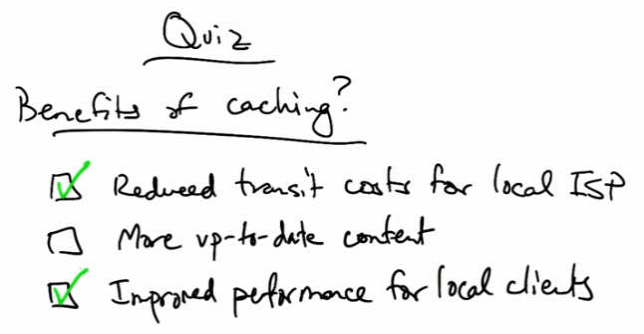
We can see these effects, for example, when we do a DNS look up for Google.com. The response returns a number of IP addresses, and when I ping the IP address, we see that the resulting IP address is only one millisecond away, which indicates that that server is not far away, but is in fact very likely on a local network, probably even the Georgia Tech campus network in this case.

Caching Quiz



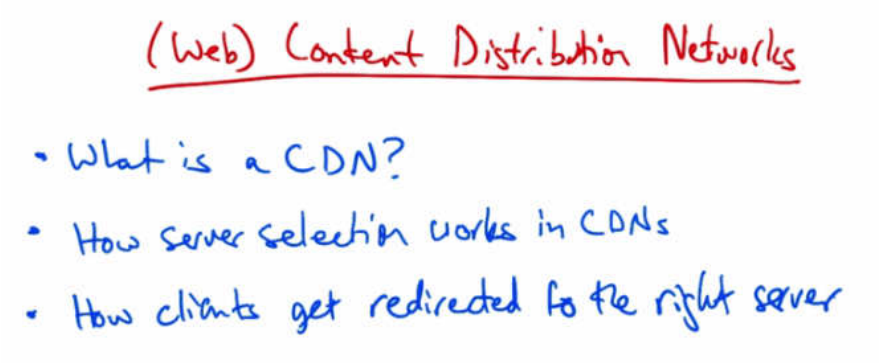
As a quick quiz, what are some of the benefits of HTTP caching? Reduced transit costs for the local ISP, more up to date content, or improved performance for local clients? Please check all that apply.

Caching Solution

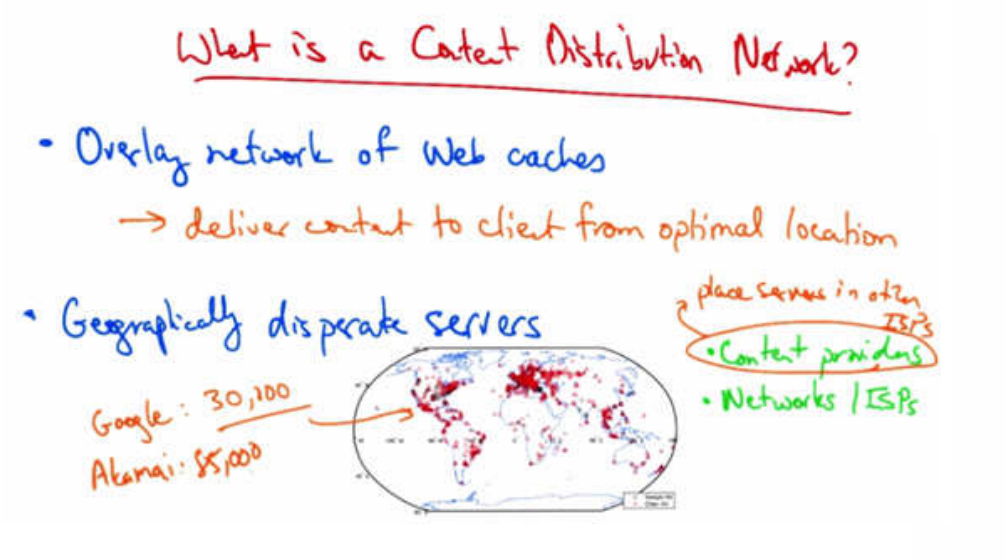


Web caching can reduce transit costs for the local ISP by preventing every HTTP request from needing to go to the origin server. Because the content's also closer to the client, clients should also see improved performance.

CDNs



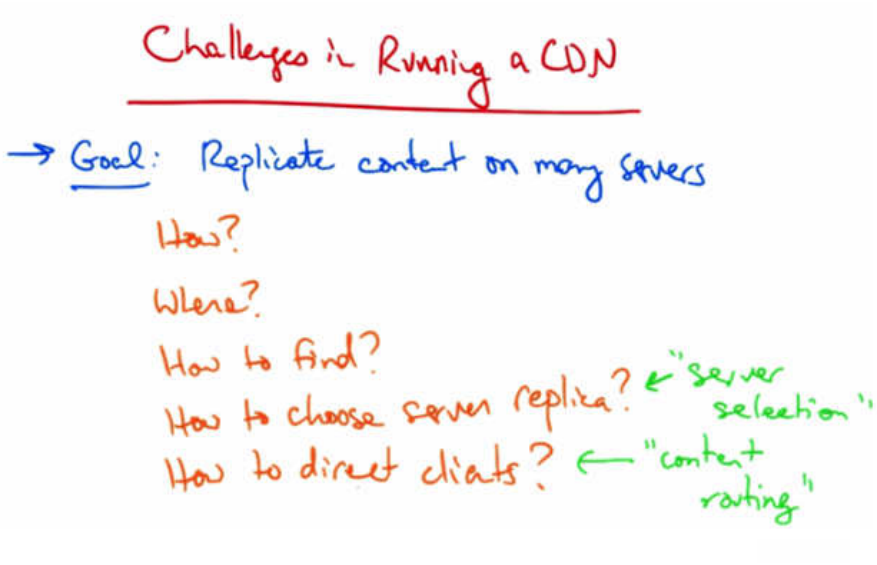
Let's now talk a little bit about web content distribution networks, or CDNs. We'll first talk about what a CDN is and why a content provider might want to use one. We'll then talk about how service selection works in CDNs and how clients get redirected to the right server.



So, first of all, what is a content distribution network? It's an overlay network of web caches that's designed to deliver content to a client from the optimal location. Now, in many cases optimal means geographically closest, but sometimes optimal is not the geographically closest cache, and we'll see some examples of when that's the case. CDNs are made of distinct geographically disparate groups of servers, where each group can serve all the content on the CDN. These CDNs can often be quite extensive. Here is a global map depicting the deployment of the Google cache servers around the world, as mapped in a recent project by researchers at the University of Southern California. As you can see, these Web caches can be quite extensive and in many cases there's a concerted effort to place caches as close as possible to users. Some CDNs are owned by content providers such as Google and others are owned and operated by networks such as Level 3, Limelight, and AT&T. Still others such as Alcamai operate independently. Non network CDNs, such as Alcamai and Google can typically place servers in other autonomous

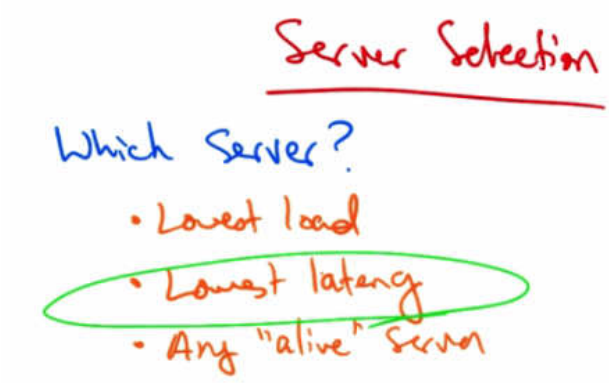
systems or ISPs. The number of cache nodes in a large content distribution network can vary. For example, in the Google Network, the USC researchers found that there were about 30,000 unique front-end cache nodes. As of about two years ago, the Alcamai Edge platform reported about 85,000 unique caching servers in nearly 1,000 unique networks around the world in 72 countries.

Challenges in Running a CDN



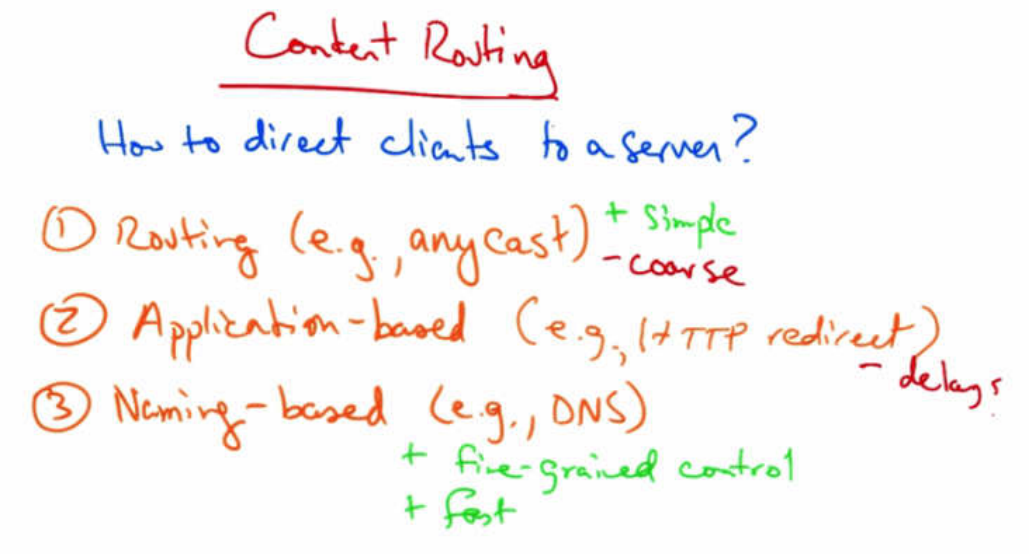
Operating a CDN presents many different challenges, and the underlying goal is to replicate content on many servers so that the content is replicated close to the clients. Yet this leaves many open questions including, how to replicate the content, where it should be replicated, how clients should find the replicated content, how to choose the appropriate server replica (or cache) for a particular client, and how to direct clients towards the appropriate replica once it's selected. This problem is commonly known as server selection, and this problem is sometimes called content routing. Let's take a look at each of these problems in a little bit more detail.

Server Selection



The fundamental problem with server selection is determining which server to direct the client to. One could do this based on a number of criteria, such as the least loaded sever, the one with the lowest network latency, or simply to any alive server to help provide fault times. Content distribution networks typically aim to direct clients towards servers that provide the lowest latency for the reasons that we talked about before, since latency plays a hugely significant role in the web performance that clients see.

Content Routing



Content routing concerns how to direct clients to a particular server. One might do this in a number of ways. One could use the routing system. For example, Anycast. So one could number all of the replicas with the same IP address and then rely on routing to take the client to the closest replica based on the routes that the internet routers choose. Routing-based redirection is simple but it provides the service providers with very little control over which servers the clients ultimately get redirected to because the redirection is at the whims of internet routing. Another way to do redirection is application based. For example, by using an HTTP redirect. This is effective but it requires the client to first go to the origin server to get the redirect in the first place, increasing latency. The third and most common way that service selection is performed is as part of the naming system using DNS. In this approach, a client looks up a particular domain name, such as Google.com, and the response contains an IP address of a nearby cache. Naming base redirection provides significant flexibility in directing different clients to different server replicas. So, in summary, routing based redirection is simple but it's very coarse. Application based routing is also fairly simple but it incurs significant delays which operators really care about, as well as users. Naming based redirection provides fine-grained control and it's also fast.

Naming Based Redirection

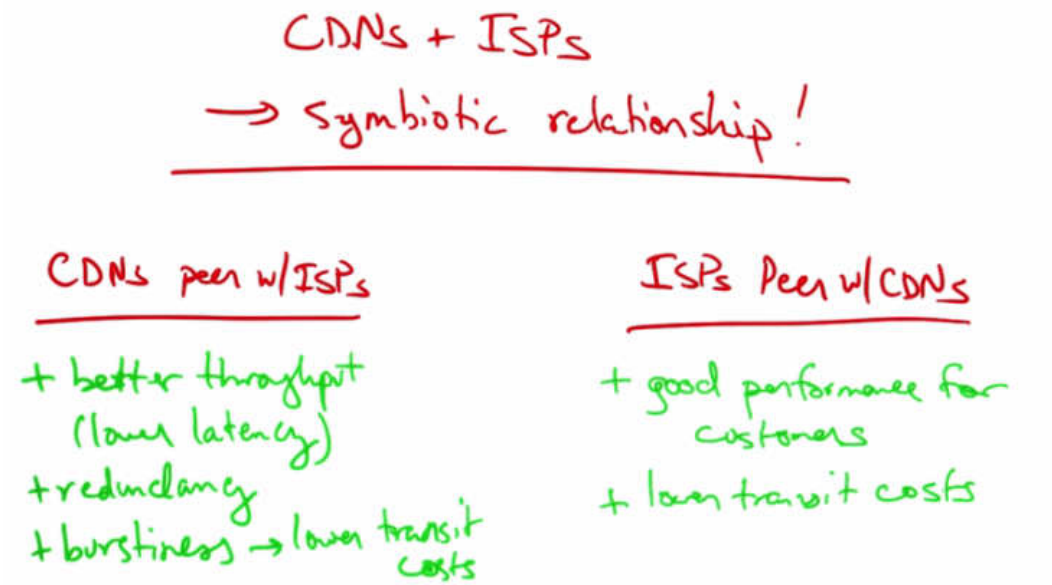
```
[NYC]% host www.symantec.com
www.symantec.com    CNAME   a568.d.akamai.net
a568.d.akamai.net   A       207.40.194.46
a568.d.akamai.net   A       207.40.194.49

[Boston]% host www.symantec.com
www.symantec.com    CNAME   a568.d.akamai.net
a568.d.akamai.net   A       81.23.243.152
a568.d.akamai.net   A       81.23.243.145
```

Let's take a closer look at how naming base redirection works. In the example shown here, I've looked up symantec.com from two different locations. You can see that when we look up the domain name, we don't get an A record immediately, but rather we get a CNAME, or a canonical name, which tells us to look up the following domain name in Alcamai. When we look up that domain name, we see two corresponding IP addresses. Notice that when we perform the same look up from Boston, we also get redirected to Alcamai through the CNAME, but we get two different IP addresses that are presumably more local to the Boston area. So, depending on where the client looks up the domain name, it receives different IP addresses at different locations in the network. This is how operators use DNS to redirect clients to nearby replicas.

```
lawn-143-215-205-52:~ udacity$ ping youtube.com
PING youtube.com (74.125.137.190): 56 data bytes
64 bytes from 74.125.137.190: icmp_seq=0 ttl=49 time=1.596 ms
64 bytes from 74.125.137.190: icmp_seq=1 ttl=49 time=1.506 ms
64 bytes from 74.125.137.190: icmp_seq=2 ttl=49 time=1.521 ms
64 bytes from 74.125.137.190: icmp_seq=3 ttl=49 time=1.724 ms
64 bytes from 74.125.137.190: icmp_seq=4 ttl=49 time=1.501 ms
64 bytes from 74.125.137.190: icmp_seq=5 ttl=49 time=1.483 ms
64 bytes from 74.125.137.190: icmp_seq=6 ttl=49 time=1.604 ms
64 bytes from 74.125.137.190: icmp_seq=7 ttl=49 time=1.521 ms
64 bytes from 74.125.137.190: icmp_seq=8 ttl=49 time=1.771 ms
64 bytes from 74.125.137.190: icmp_seq=9 ttl=49 time=1.539 ms
64 bytes from 74.125.137.190: icmp_seq=10 ttl=49 time=1.503 ms
64 bytes from 74.125.137.190: icmp_seq=11 ttl=49 time=1.432 ms
64 bytes from 74.125.137.190: icmp_seq=12 ttl=49 time=1.539 ms
^C
--- youtube.com ping statistics ---
13 packets transmitted, 13 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 1.432/1.557/1.771/0.092 ms
lawn-143-215-205-52:~ udacity$ dig -x 74.125.137.190

; <<>> DiG 9.7.6-P1 <<>> -x 74.125.137.190
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 42868
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;190.137.125.74.in-addr.arpa.   IN      PTR

;; ANSWER SECTION:
190.137.125.74.in-addr.arpa. 18697 IN   PTR     yh-in-f190.1e100.net.

;; Query time: 5 msec
;; SERVER: 130.207.165.170#53(130.207.165.170)
;; WHEN: Fri Nov  1 07:36:03 2013
;; MSG SIZE  rcvd: 79

lawn-143-215-205-52:~ udacity$
```

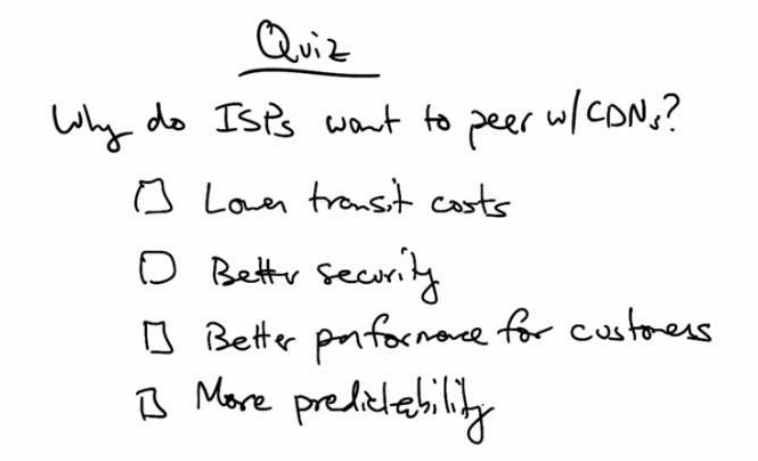
As another example you can see when I ping youtube.com, I get very low latencies. And when I do a reverse lookup on this IP address, I in fact see that the content was posted on Google CDN.

CDNs and ISPs



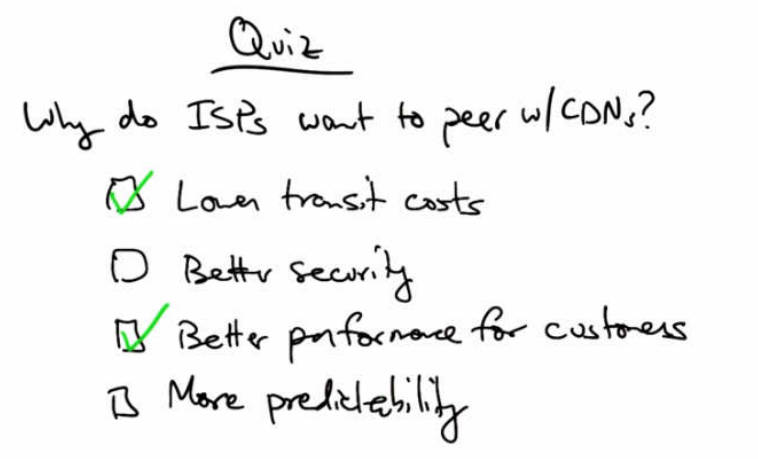
It turns out that content distribution networks and ISPs have a fairly symbiotic relationship when it comes to peering relationships. CDNs like to peer with ISPs because peering directly with ISPs where a customer's located provides better throughput since there are no intermediate AS hops and network latency is lower. Having more vectors to deliver a content increases reliability, and during large request events, having direct connectivity to multiple networks where the content is hosted allows an ISP to spread its traffic across multiple transit links, thereby potentially reducing the 95th percentile and lowering its transit costs. On the other hand, there are other good reasons for ISPs to peer with CDNs. First of all, providing content closer to the ISP's customers allows the ISP to provide the customers with good performance for a particular service. For example, you could already see that Georgia Tech has placed a Google cache node in its own network, resulting in very low latencies to Google, and thereby happy customers. You can imagine that providing good performance to popular services is a major selling point for ISPs. Another reason that ISPs like to peer with CDNs or host cache nodes locally is to lower their transit costs. You can imagine that if there are a huge demand for a particular video on YouTube and all the requests and responses were going over expensive transit links, then the ISPs cost would be potentially prohibitively high. On the other hand, peering with the CDN, or hosting a local cache node, prevents all of that traffic from traversing expensive links, thus reducing costs.

CDNs and ISPs Quiz



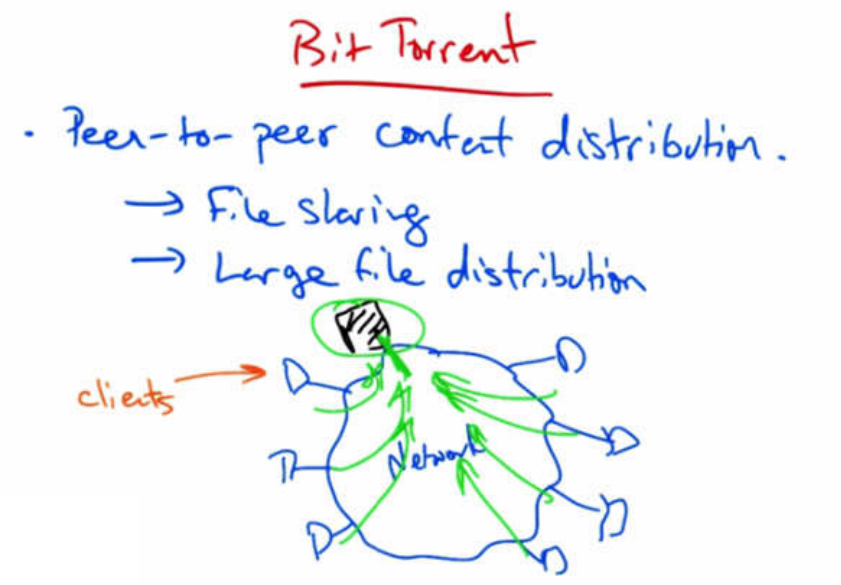
As a quick quiz, why do ISPs want to peer with CDNS? Lower transit costs, better security, better performance for its customers, or more predictability? Please check all that apply.

CDNs and ISPs Solution

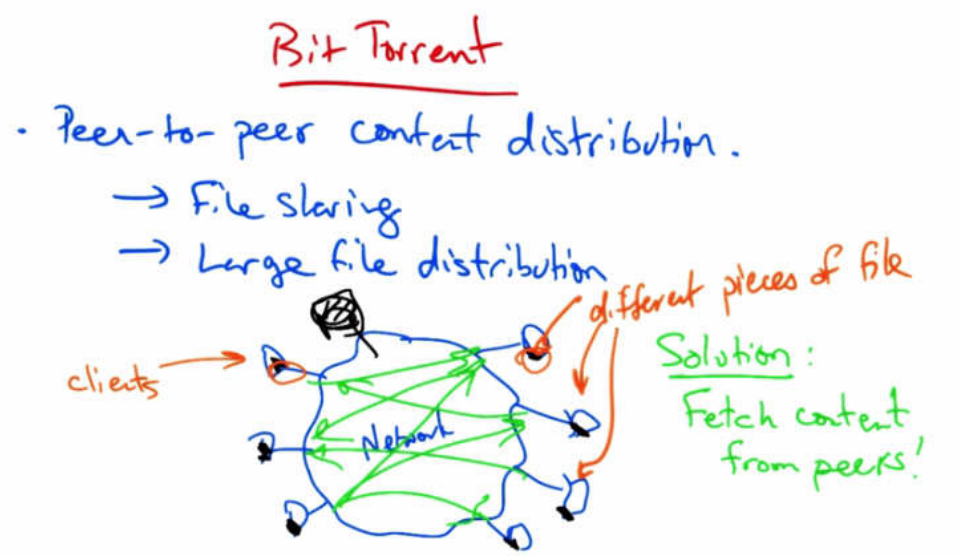


ISPs typically want to peer with CDNs to lower the transit costs and to provide better performance for their customers. CDNs don't inherently provide better security or more predictability. And, in fact, some ways of redirecting clients to servers may actually reduce predictability.

Bit Torrent



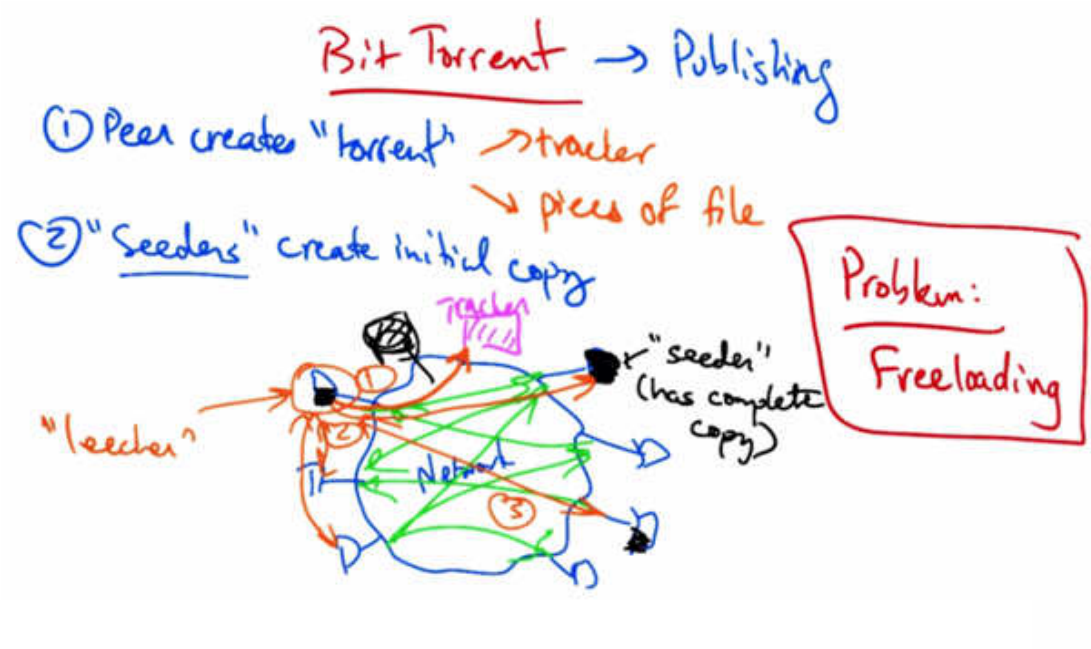
Okay, we're now going to talk about Bit Torrent, which is a peer to peer content distribution network that is commonly used for file sharing and distribution of large files. Okay, suppose we have a network with a bunch of clients, all of whom want a particular file and the file might be particularly big. Now, those clients could all fetch the same file from the source, or the origin. But the problems with that, of course, are that the origin may be overloaded and the act of making this request for a large file from the same location on the network may also create congestion or overload at the network where the content is being hosted.



So, a solution is to fetch content from other peers. Rather than having everyone fetch the content from the origin, we can take the original file and chop it into many different pieces and replicate different pieces on different peers in the network, as soon as possible. So the idea is that each

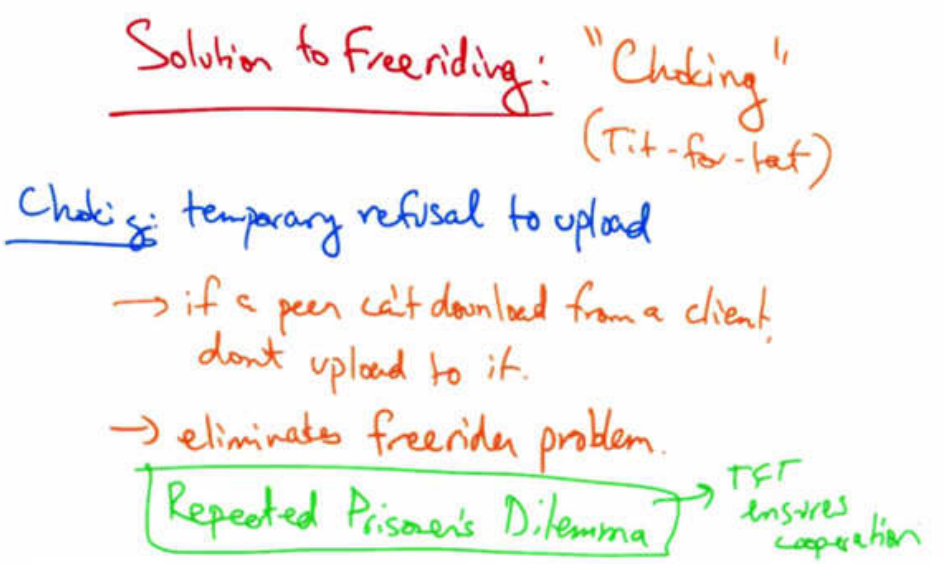
peer is assembling the file, but it's assembling it by picking up different pieces of the file. And then it can retrieve the pieces that it doesn't have from the remaining peers in the network. By trading different pieces of the same file, everyone eventually gets the full file. The idea is that hopefully we'll be able to assemble the entire file at the end by the time all of the clients have swapped.

Bit Torrent Publishing



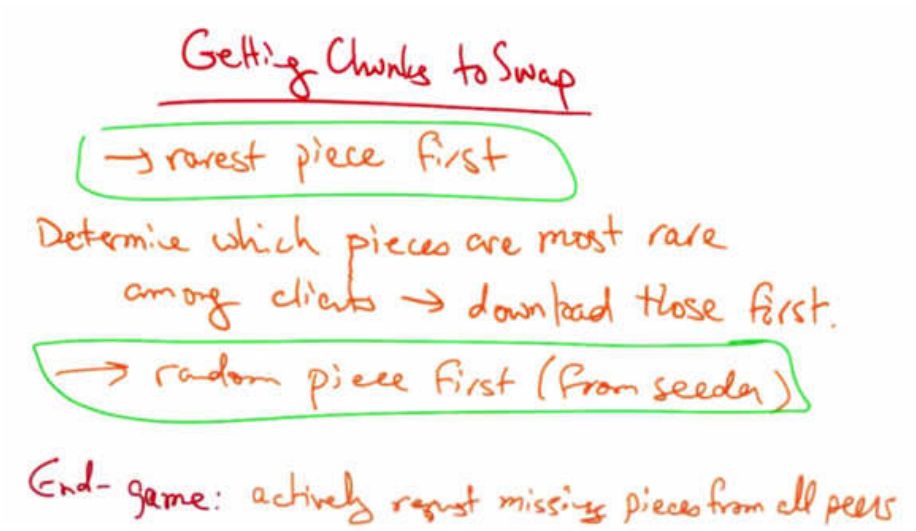
Bit Torrent has several steps for publishing. First, a peer creates what's called a torrent which contains metadata about tracker and all of the pieces of the file in question as well as a checksum for each piece of the file at the time the torrent was created. Now some peers in the network need to maintain a complete initial copy of the file. Those peers are called seeders. Now to download a file, a client first contacts the tracker which provides this metadata about the file, including a list of seeders that contain an initial copy of the file. Next, the client starts to download parts of the file from the seeder. Once the client starts to accumulate some initial chunks, hopefully those chunks were different than those that other clients in the network that are also trading the file have. At this point clients can begin to swap chunks. As clients begin swapping distinct chunks with one another, the idea is that eventually, after enough swapping, everyone gets a copy of the complete file. Clients that contain incomplete copies of the file are called leechers. The tracker allows peers to find each other and it also returns a random list of peers that any particular leecher can use to swap chunks of the file. Previous, peer to peer file-sharing systems used similar swapping techniques, but a problem that many of them faced, and which Bit Torrent solved, is called free-loading, whereby a client might leave the network as soon as it finished downloading a copy of the file, not providing any benefit to other clients who also want the file.

Solution to Free-riding



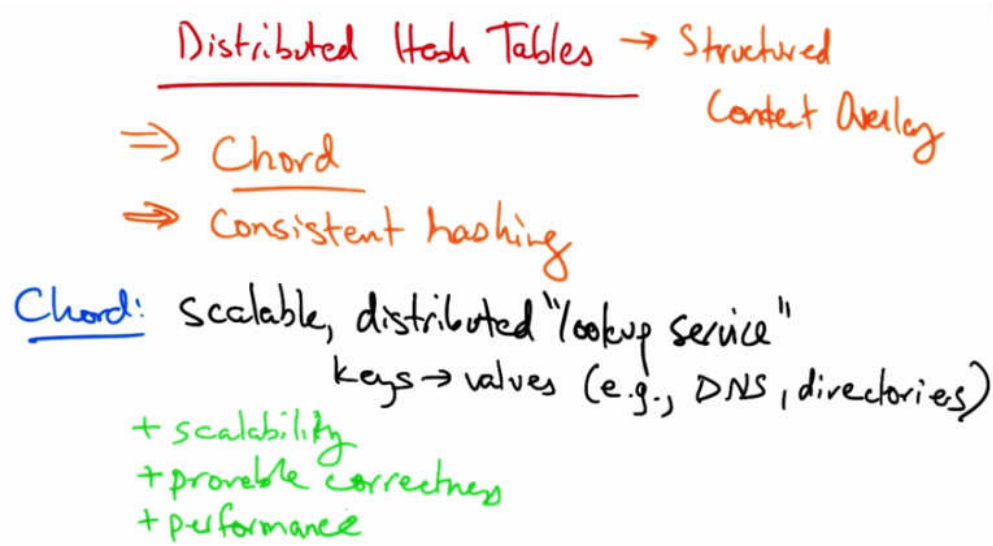
Bit Torrent's solution to free-riding is called choking, which is a type of game theoretic strategy, called tit for tat. Choking is a temporary refusal to upload chunks to another peer that is requesting them. Downloading, of course, occurs as normal. But if a node is unable to download from any particular peer, it simply doesn't upload to that peer. This ensures that nodes cooperate and eliminates the free-rider problem. If you're interested in the game theory behind why this strategy ensures cooperation, I encourage you to go read about the repeated prisoner's dilemma problem where a tit-for-tat strategy, such as that which is shown here, ensures cooperation among mutually distrustful parties.

Getting Chunks to Swap



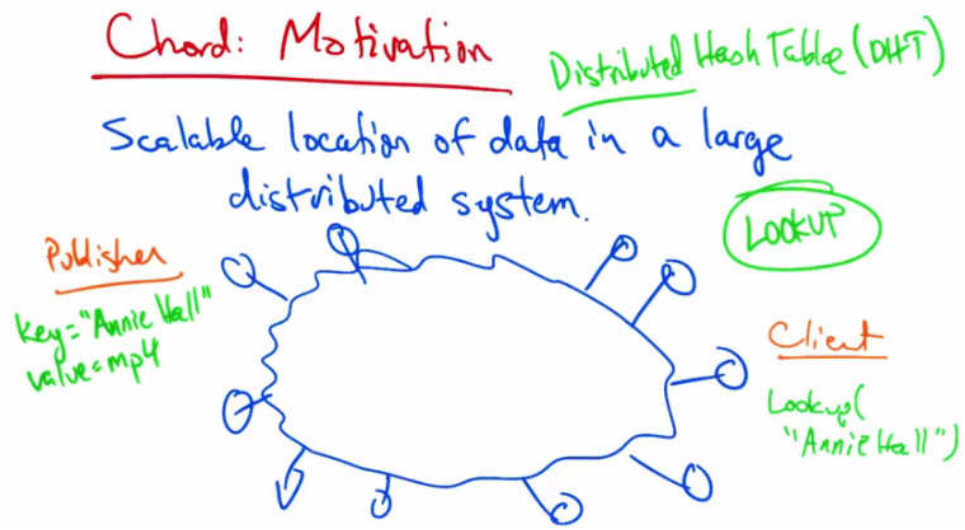
One of the problems that Bit Torrent needs to solve is ensuring that each client gets chunks to swap with other clients. If all the clients received the same chunks, then no-one would want to trade with one another and everyone would have an incomplete copy of the file. To solve this problem, Bit Torrent clients use a policy called rarest piece first. Rarest piece first allows a client to determine which pieces are the most rare among clients, and download the rarest pieces of the file first. This ensures that the most common pieces are left till the end to download and that a large variety of pieces are downloaded from the seeder. Additionally, a client has nothing to trade and it's important to get a complete piece as soon as possible. Rare pieces are typically available at fewer peers initially. Downloading a rare piece is initially maybe not a good idea. So one policy that clients use is to select a random piece of the file and download it from a seeder. In the end game the client actively requests any missing pieces from all peers, and redundant requests are cancelled when the missing piece arrives. This ensures that a single peer with the slow transfer rate doesn't prevent the download from completing.

Distributed Hash Tables



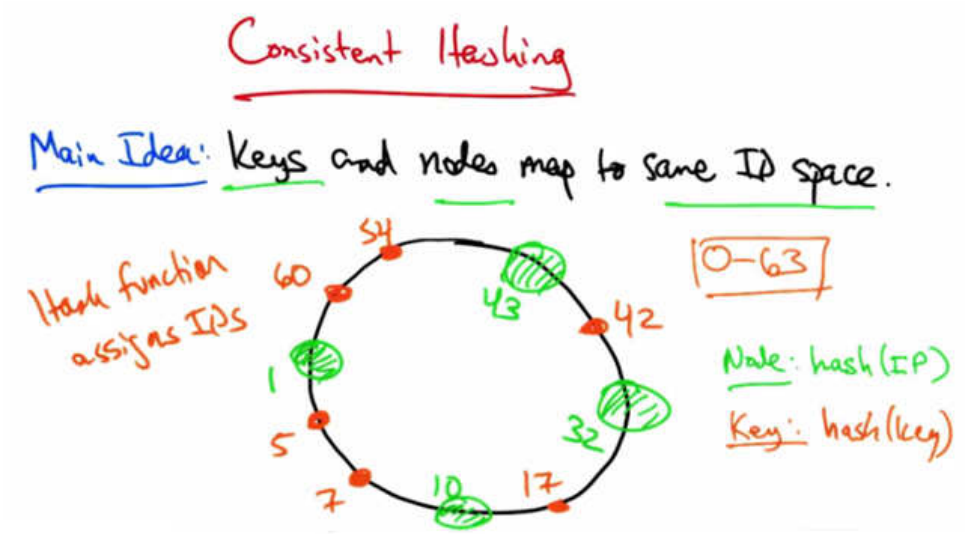
In this lesson we will talk about distributed hash tables, which enable a form of content overlay called a structured overlay. We'll talk about a particular distributed hash table called Chord and an underlying mechanism that enables it, called consistent hashing. Chord is a scalable, distributed lookup service. A lookup service is simply any service that maps keys to values. Examples of lookup services on the internet include DNS and directory services. Chord has some desirable properties, including scalability, provable correctness, and reasonably good performance that's also fairly easy to reason about.

Chord Motivation

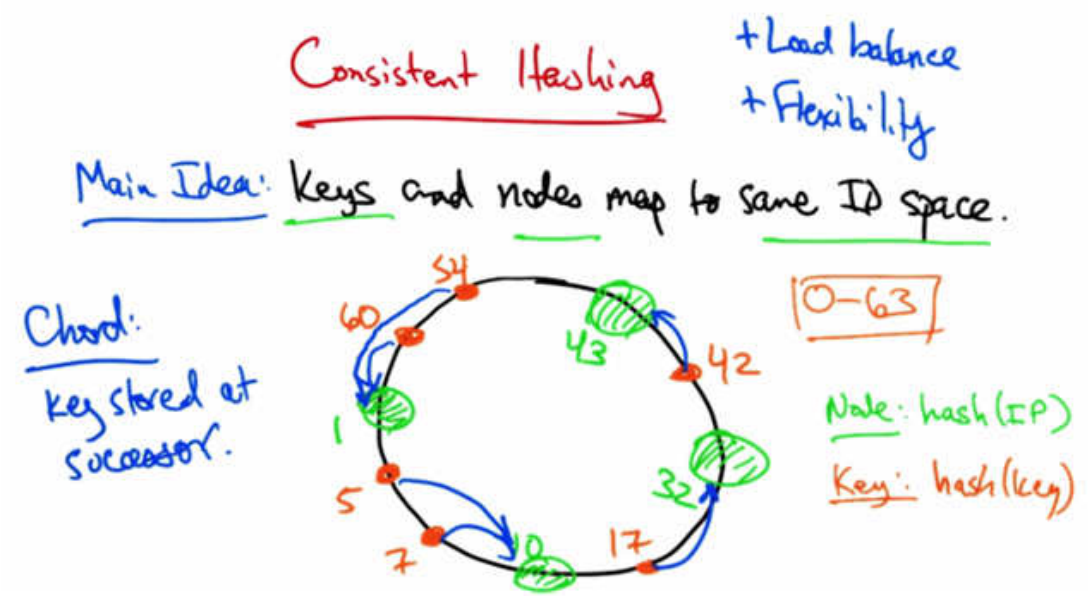


The main motivation of Chord is scalable location of data in a large distributed system. So a publisher might want to publish the location of a particular piece of data, such as an MP4 with a particular name, such as Annie Hall. It needs to figure out where to publish this data in a place that the client can find it so that when the client performs a look up for Annie Hall, it's directed to the right location that is hosting the data. The key problem that we need to solve here is look up and you can see that the function that needs to be provided is just a simple hash table, but the thing that makes this problem interesting is that the hash table isn't located in one place but that it's distributed across the network. So what we're trying to build is what's called a distributed hash table or a DHT. The way that we're going to build this is using a mechanism called consistent hashing.

Consistent Hashing

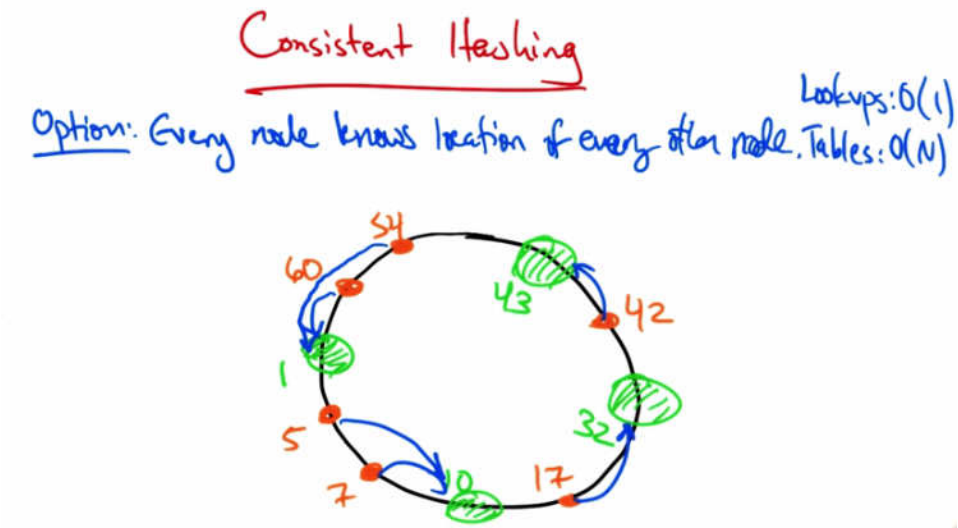


In consistent hashing, the main idea is that the keys and the nodes map to the same ID space. So what we're going to do is create a metric space, such as a ring, and we'll put nodes on this ring, and the idea is that these nodes each have some ID. Now the keys should also map to the ID space. So in this case, just for the sake of example, let's suppose that we have a six bit ID space, so ID's might range from zero to 63. Now you can see that the nodes have ID's, and the keys also have ID's in the same space. A consistent hash function will assign the nodes and the keys and identifier in this space. A hash function such as SHA-1 might be used to assign these identifiers. In the case of nodes, the ID might be a hash of the IP address. In the case of keys, the ID might simply just be the hash of a key. Both of these hash operations create ID's that are uniformly distributed in the ID space. The question now is how to map the key ID's to the node ID's, so that we know which nodes are responsible for resolving the look ups for a particular key.

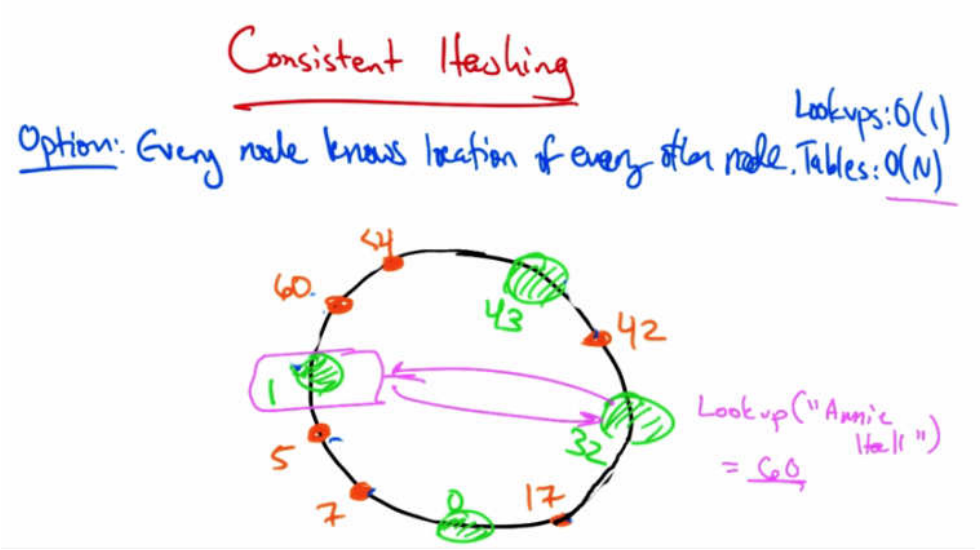


The idea in chord is that a key is stored at its successor, which is the node with the next highest ID. So, for example, the key corresponding to the key ID of 60 would be stored at the node with the node ID of one, similarly for the key with the key ID of 54. Forty-two would be stored at the node with the node ID of 43, 17 at the node with 32, seven and five at the node with ID of 10, and so on. Consistent hashing offers the properties of load balance, because all nodes receive roughly the same number of keys and flexibility because when a node joins or leaves the network, only a fraction of the keys need to be moved to a different location. You can actually prove that the solution is optimal, meaning that the minimal number of keys need to be remapped to maintain load balance when a node joins or leaves the network.

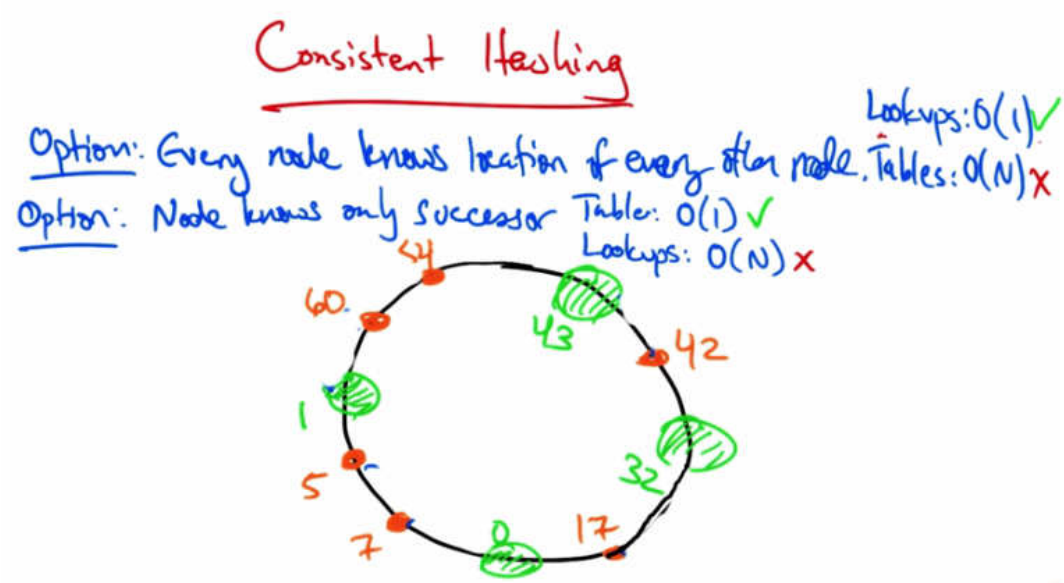
Implementing Consistent Hashing



Let's talk a little bit about how to implement consistent hashing. One option is for every node to know the location of every other node. In this case, lookups are fast. In fact, they are order one, but the routing tables are large. In particular, because every node needs to know the location of every other node in the network,, the routing table must be order N, where N is the number of nodes in the network.

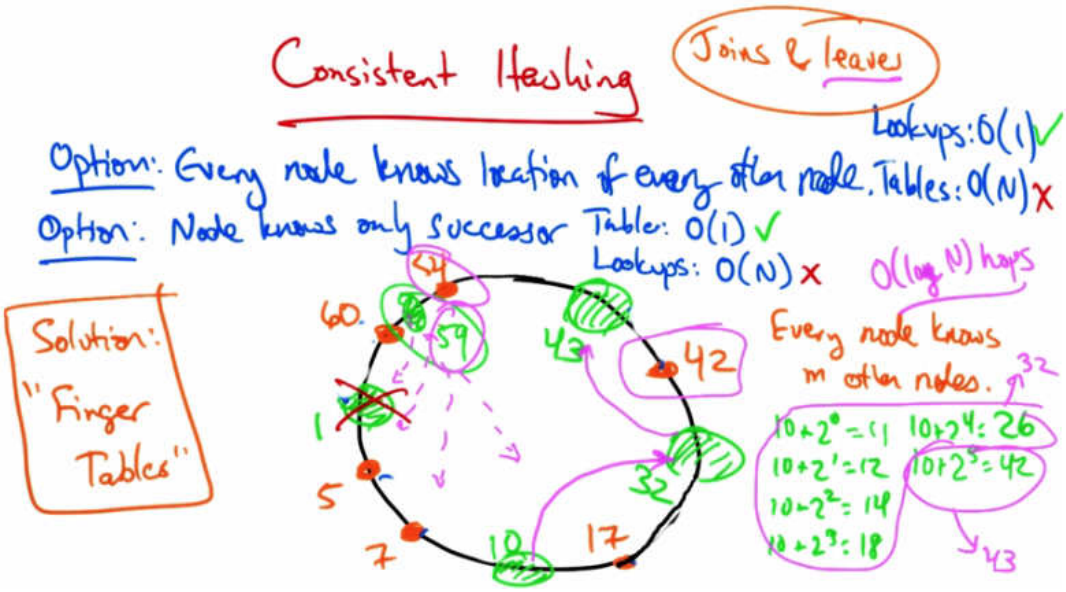


So, for example, if node 32 wanted to look up the location of Annie Hall, that value might hash to 60, and if every node maintains a routing table entry for every other node, 32 would know that the key corresponding to ID 60 was located at node one. So the look up, would be order one, but the table ,would be order N.



Another option is that each node only knows the location of its immediate successor in the ring. So, for example, node 32 would know the location of node 43, but of no other node. This results in a small table, of size order one. But locating the content, as before, would require order N lookups. So in summary, if every node knows the location of every other node, then lookups have good performance at the expense of larger tables. If every node only knows its successor, then routing tables can be small, but every lookup operation is order N.

Finger Tables



A solution that provides the best of both worlds is called finger tables, where every node knows m other nodes in the ring and the distance of the nodes that it knows increases exponentially. So, for example, node 10 would maintain mappings for 10 plus 2 to the 0, 10 plus 2 to the 1, and so

forth, where finger i Points to the successor of n plus 2i. So finger 0 would point to the successor of 11, which is 32; Finger 1 would also point to 32 and so forth. Finger 5 would point to 43. Now every node knows its immediate successor. So what you want to do is find the predecessor for a particular ID and then ask for the successor of that ID. So let's suppose that node 10 wanted to look up a key corresponding to the id of 42. It can use the finger tables to find the predecessor of that node, which in this case, is 32. Its finger tables have the mapping of that nodes location as well. It then can ask node 32 for its successor. At this point, we can move forward around the ring looking for the node whose successor's ID is bigger than the ID of the data, which in this case is node 43. Due to the structure of the finger table, these lookups require order of log n hops. This results in efficient lookups, order log n messages per look up, and the size the finger table is order of log n state per node. Another consideration that we have to take into account is what happens when nodes join and leave the network. When a new node joins, we first have to initialize the fingers of this new node. Then we must update the fingers of existing nodes so that they know that they can point to the node with the new ID. And finally, the third step is to transfer the keys from the successor to the new node. In this case, the key that we must transfer from the successor, one, is the data with ID of 54. In this case, each node's successor is maintained and the successor of any particular ID k is always responsible for k. A fallback for handling leaves is to ensure that any particular node not only keeps track of its own finger table, but also of the fingers of any successor, so that if a node should fail at any time, then the predecessor node in the ring also knows how to reach the nodes corresponding to the entries in the failed nodes finger table.

CS 6250: Computer Networking

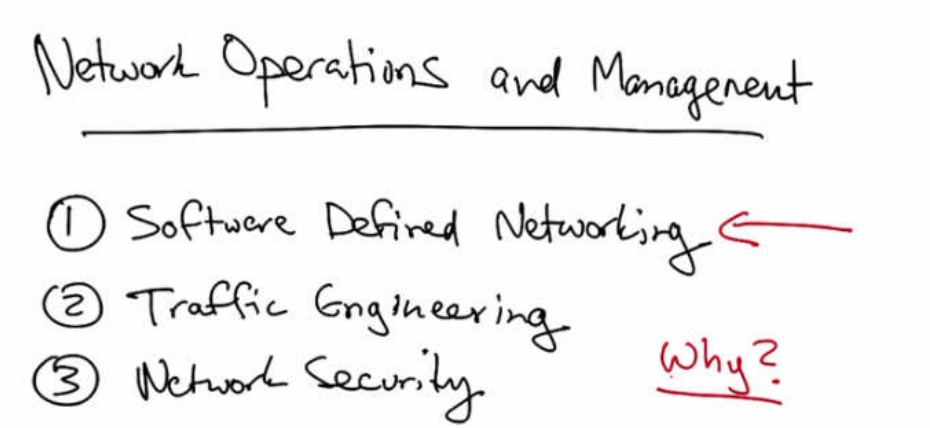
Lecture 9: Software Defined Networking Introduction

Operations and Management Overview

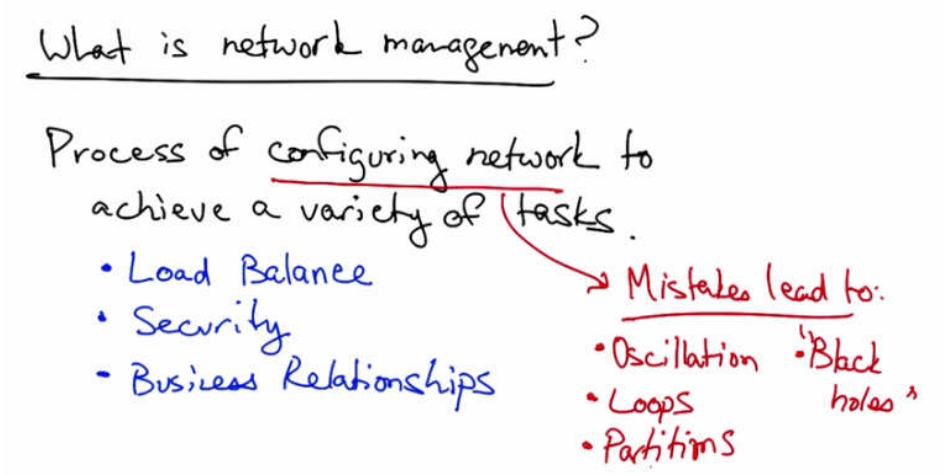
Welcome to the final third of the course. In the first section, you reviewed the basic building blocks of the internet, and in the second section, you learned how networks deal with large amounts of network traffic. In the final section, you'll learn how network operators manage their networks. More importantly, these topics will introduce you to the forefront of networking research.

That's right. We're going to cover software defined networking, traffic engineering, and network security. Let's get started.

Network Management Overview

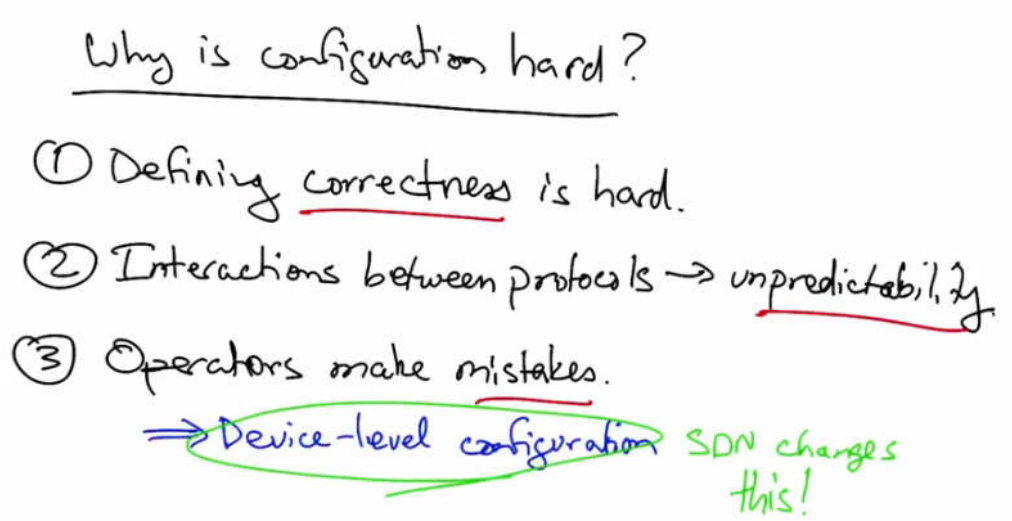


Welcome to the third course in CS 6250 where we will be discussing network operations and management. This segment in the course has three lessons. The first lesson is focused on software-defined networking and its role in making network operations and management easier. The second module covers traffic engineering, which is the process by which network operators reconfigure the network to balance traffic demands across the network. The third lesson covers network security. We will start with a lesson on Software Defined Networking. But, before we jump into the details, I'd like to motivate, a little bit—why? In particular, I plan to tell you about the role of network operators in running the network.



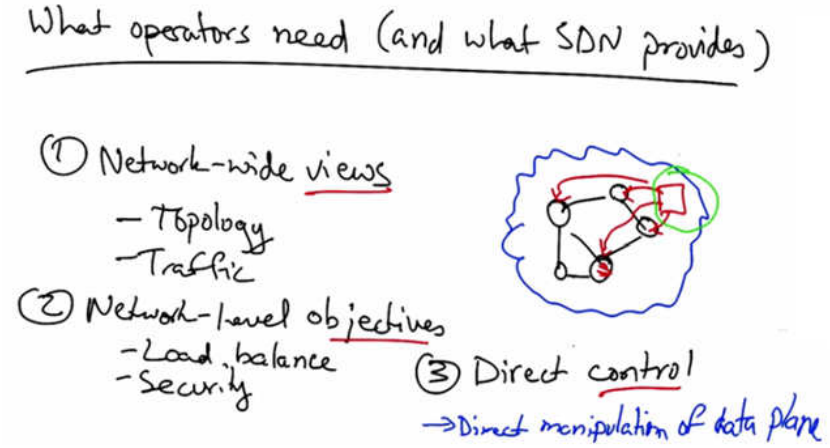
So what is network management? Network management is the process of configuring the network to achieve a variety of tasks. Network configuration achieves a variety of tasks including balancing traffic load across the network, achieving various security goals, and satisfying business relationships that may exist between the network that's being configured and neighboring networks, such as the network's upstream Internet service provider. A key aspect to network management is configuring the network. Unfortunately, if the network is not configured correctly, many things can go wrong. Configuration mistakes can lead to problems such as persistent oscillation, whereby routers can't agree on a route to a destination; loops, where packets get stuck in between two or more routers and never actually make it to the destination; partitions, whereby the network is split into two or more segments that are not connected; and black holes, where packets reach a router that does not know what to do with the packet and drops it as opposed to sending it on to its ultimate destination.

Why is Configuration Hard

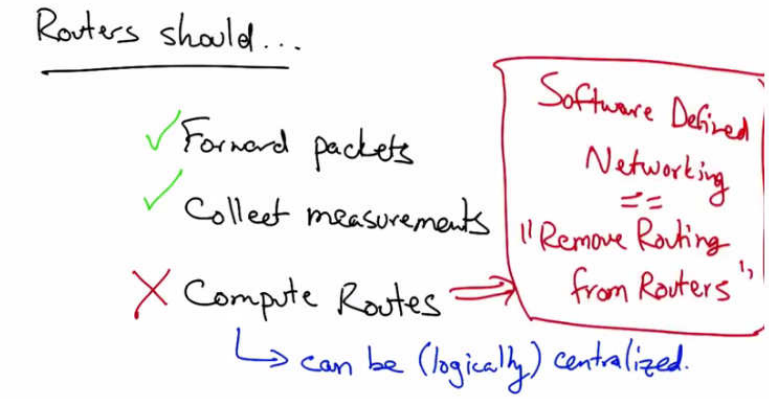


So why is configuration hard to get right? First, it's difficult to define what we mean by correct behavior in the first place. Second, the interactions between multiple routing protocols can lead

to unpredictability. Furthermore, each autonomous system on the internet is independently configured, and the interaction between the policies of these autonomous systems can lead to unintended, or unwanted behavior. The third reason that configuration is hard is that operators simply make mistakes. Configuration is difficult, and network policies are very complex. Furthermore, Network configuration has historically been distributed across hundreds or more network devices across the network, where each device is configured with vendor-specific low-level configuration. We'll see in the first part of this course how Software Defined Networking or SDN changes this by centralizing the network's configuration in a logically centralized controller.

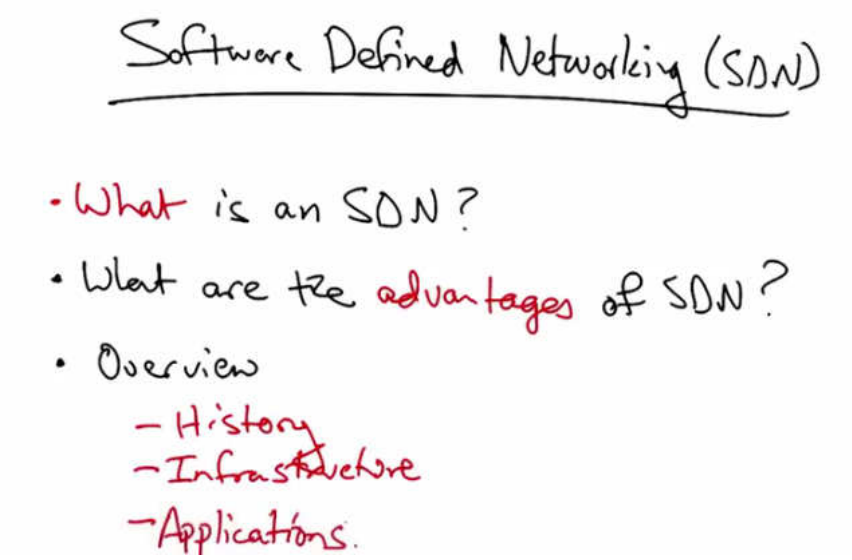


At a very high level, Software Defined Networking provides exactly the primitives that operators need to run the network better. In particular, SDN provides operators three things. The first is network-wide views of both topology and traffic. The second is the ability to satisfy network level objectives such as those that we talked about before including load balance, security, and other high level goals. The third thing that software defined networking provides (that network operators need) is direct control. In particular, rather than requiring network operators to configure each device individually with indirect configuration, SDN allows an operator to write a control program that directly affects the data plane. So rather than having to configure each device individually and guess or infer what might happen, software-defined networking allows a network operator to express network level objectives and direct control from a logically centralized controller.

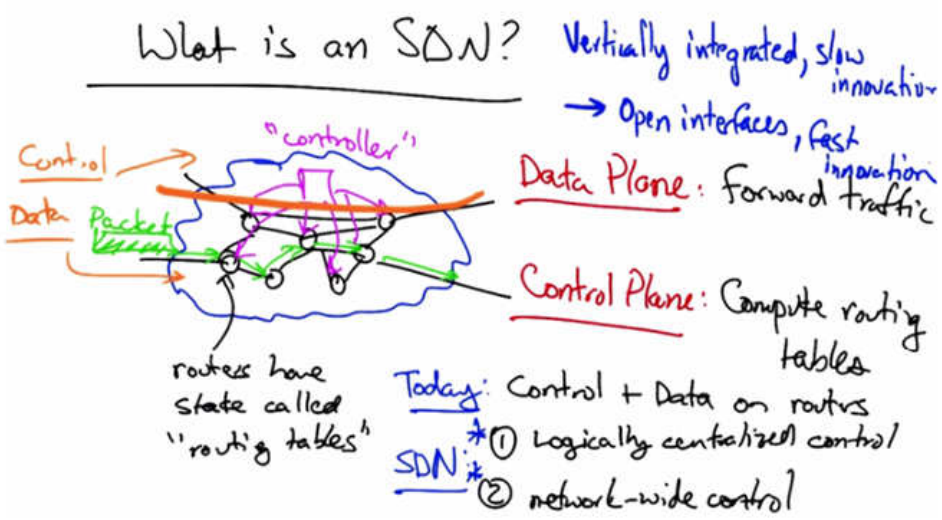


So to make network operations easier, routers should forward packets since router hardware is specialized to forward traffic at very high rates. They should collect measurements such as traffic statistics and topology information. But, on the other hand, there's no reason that a router should have to compute routes. Although conventionally routing has operated as a distributed computation of forwarding tables, the computation doesn't inherently need to run on the routers. Rather, the computation could be logically centralized and controlled from a centralized control program. This logical centralization is the fundamental tenant of SDN. So a simple way of summing up Software Defined Networking is simply to remove routing from the routers, and perform that routing computation at a logically centralized controller. Now of course, SDN has evolved to incorporate a much broader range of controls than simply routing decisions, and we'll talk about the range of control that SDN controllers enable in today's networks throughout this lesson.

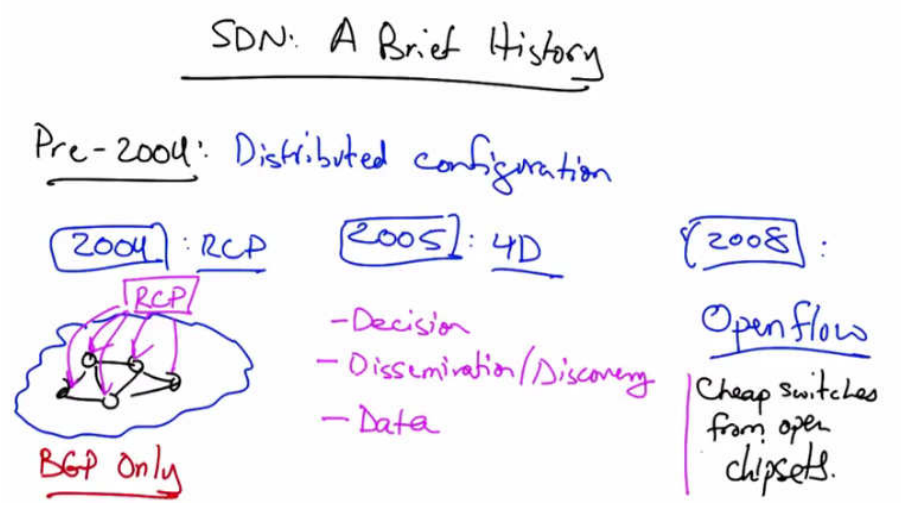
Software Defined Networking



Let's start with a brief overview of Software Defined Networking, or SDN. We'll first start by defining SDN, and in particular we'll talk about what is a Software Defined Network. Then we'll talk about what are the advantages of SDN over a conventional network architecture. We'll overview the history of SDN, the infrastructure that supports it (in particular how SDNs are designed and built), and the applications of SDN. Specifically, what they can be used for and how they can be used to simplify various network management tasks.

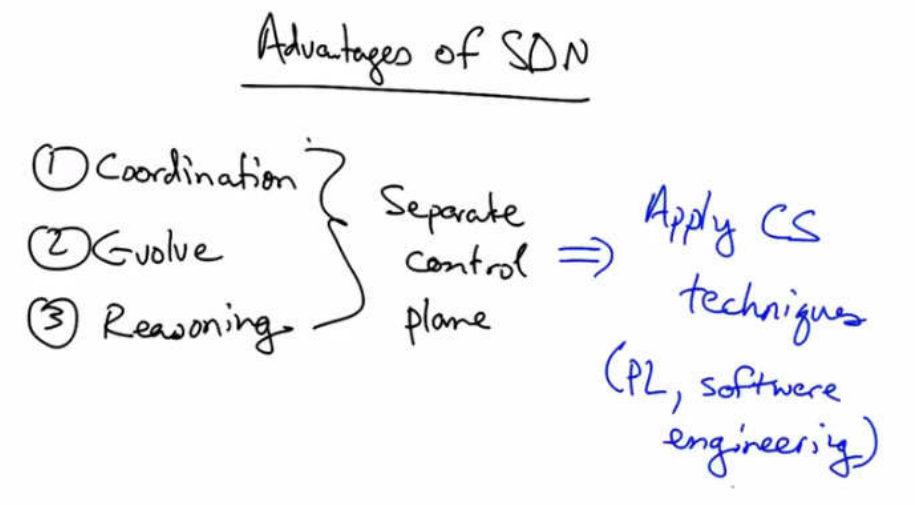


Perhaps the best way to understand what an SDN is, is to compare it to the behavior of today's networks. Today's networks have two functions. The first is the Data Plane, whose task it is to forward packets to their ultimate destination. But in order for the Data Plane to work, we also need a way of computing the state that each of these routers has that allows the routers to make the right decision in forwarding traffic to the destination. The state that lives in each of these routers that allows the routers to make these decisions about how to forward packets are called routing tables. It's the job of the network's Control Plane to compute these routing tables. In conventional networks, the Control and Data Plane both run on the routers that are distributed across the network. In an SDN, the Control Plane runs in a logically centralized controller. Additionally, the controller typically controls multiple routers across the network and often, the control program exerts control over all the routers in the network, thus facilitating network-wide control. These two characteristics are the defining features of a Software Defined Network. The separation of data and control allows a network operator to build a network with commodity devices, where the control resides in a separate control program. This re-factoring allows us to move from a network where devices are vertically integrated (making it very tough to innovate) to a network where the devices have open interfaces that can be controlled by software, thus allowing for much more rapid innovation.

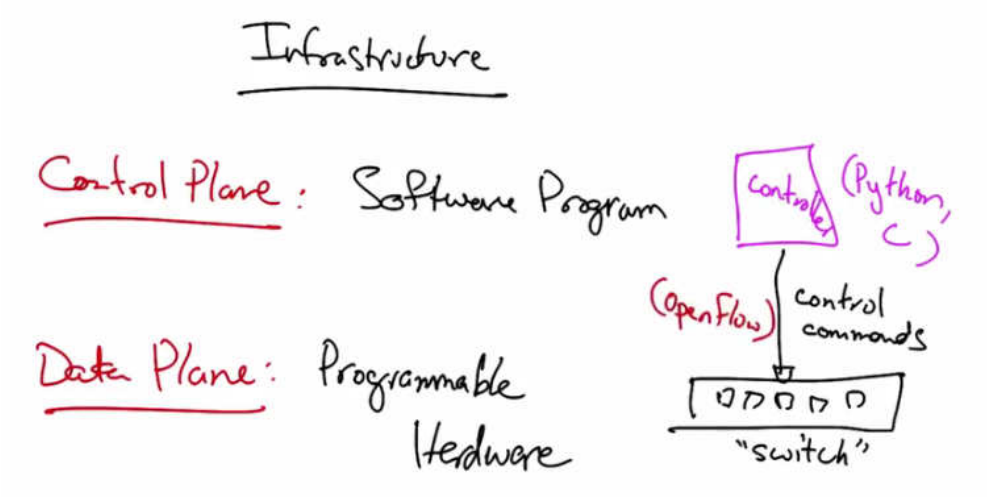


Let's survey a brief history of SDN. Previous to 2004, configuration was distributed, leading to buggy and unpredictable behavior. Around 2004, we had the idea to control the network from a logically centralized high level program. That logically centralized controller focused on the border gateway protocol and was called the routing control platform, or RCP. In 2005, researchers generalized the notion of the RCP for different planes. The decision plane, which computed the forwarding state for devices in the network; the data plane, which forwarded traffic based on decisions made by the decision plane; and the dissemination and discovery planes, which provide the decision plane the information that it needs to compute the forwarding state which ultimately gets pushed to the data plane. Around 2008, these concepts effectively hit the mainstream through a protocol called OpenFlow. OpenFlow's intellectual roots are with the RCP and 4D, but OpenFlow was made practical when merchant silicon vendors opened their APIs, so that switch chipsets could be controlled from software. So suddenly there was an emergence of cheap switches that were build based on open chip sets that could be controlled from software. This development effectively allowed us to decouple the control plane and the data plane in commodity switching hardware.

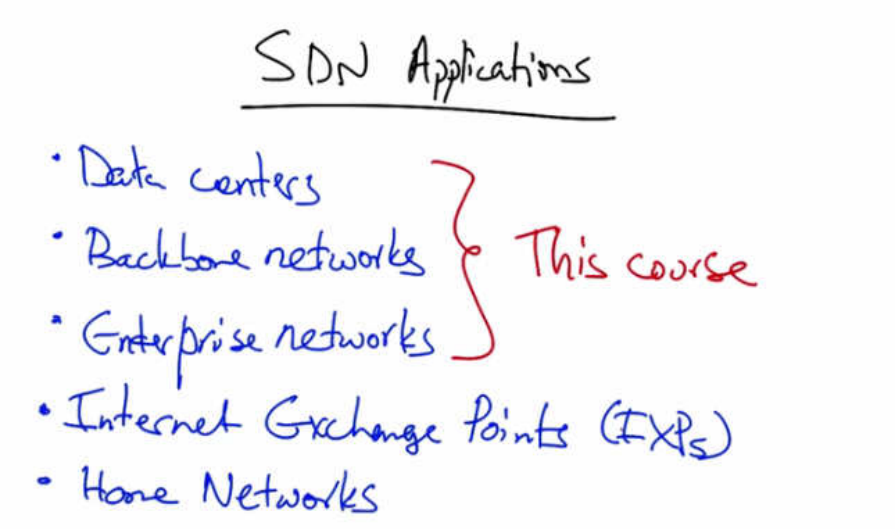
Advantages of SDN



SDN has many advantages over conventional networks. It's easier to coordinate behavior among a network of devices, the behavior of the network is easier to evolve, and it's also easier to reason about. These characteristics are all rooted in the fact that the control plan is separate from the data plane. Having a separate control plane or control program allows us to provide conventional cs techniques to old networking problems. So, whereas before it was incredibly difficult to reason about or debug a network's behavior, if the network behavior is now controlled by a logically centralized control program, we can use techniques from programming languages or software engineering to help us reason about the behavior of the network.

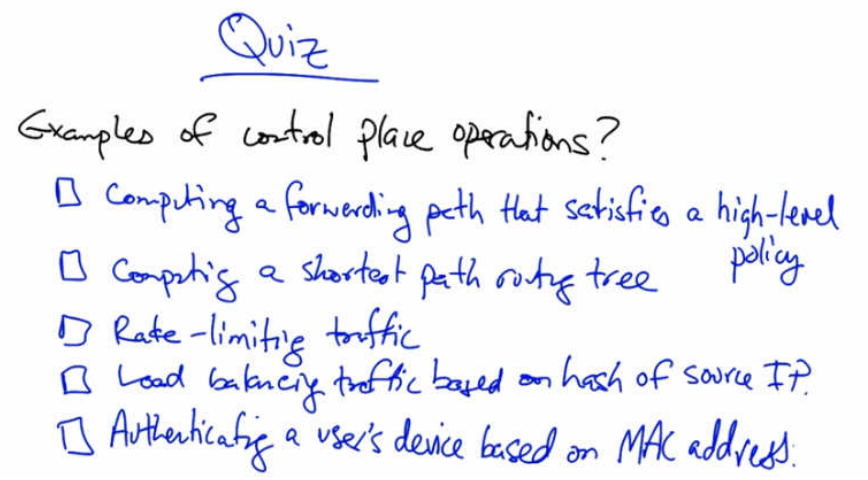


As far as SDN's infrastructure is concerned, the Control Plane is typically a software program written in a high level language, such as Python or C. On the other hand, the Data Plane is typically programmable hardware that's controlled by the control plane. The controller effects the forwarding state that's in the switch using control commands. Open flow is one standard that defines a set of control commands by which the controller can control the behavior of one or more switches.



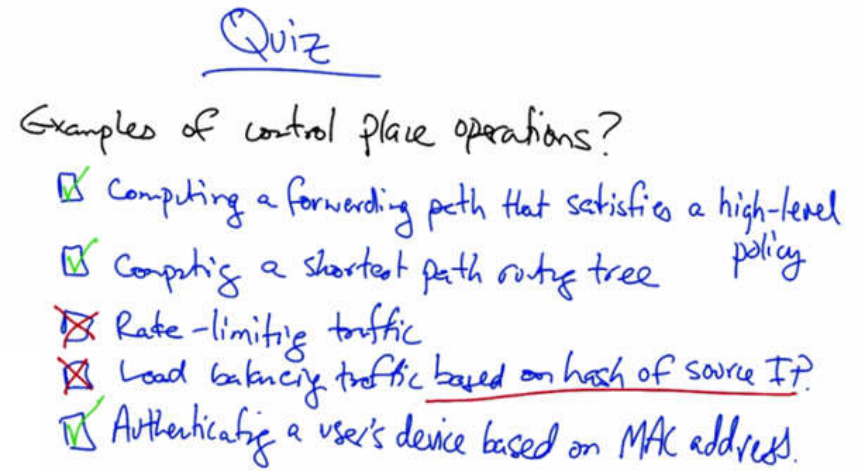
SDN has many applications including data centers, wide area backbone networks, enterprise networks, internet exchange points (or IXPs), and home networks. Later modules in this course will explore how software defined networks can solve network management problems in some of these areas. In this course we will focus in particular on the first three applications.

Control Plane Operations



So as a quick quiz, which of the following are examples of control plane operations? Computing a forwarding path that satisfies some high-level policy such as an access control policy? Computing a shortest path routing tree? Rate-limiting traffic so that the overall sending rate doesn't exceed a certain throughput? Load balancing traffic based on a hash of the packet source IP address? Or authenticating a user's device based on its MAC address? Please check all options that apply.

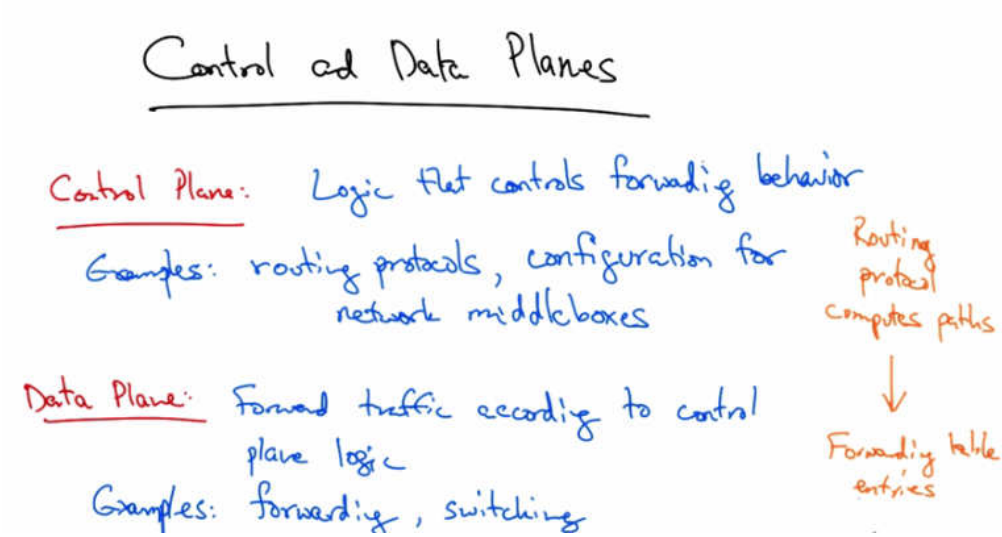
Control Plane Operations



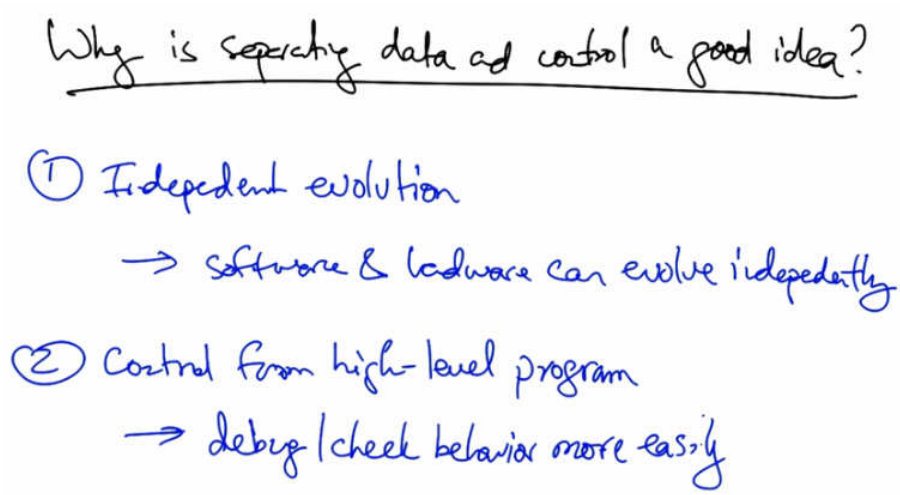
The job of the Control Plane is to compute the state that ultimately ends up in the data plane. So computing a forwarding path that satisfies a high-level policy is something that the Control Plane would do. The Control Plane can also compute shortest path routing trees. And it might make decisions about whether or not a user's device should be allowed to send traffic or not based on that device's MAC address. Rate-limiting is something that is typically done in the data plane, and the load-balancing example that we have listed here is such that a router or a switch would

make decisions in the data plane based on a hash of the source IP address. So all of the decisions are being made at forwarding time, not by a centralized high-level program.

Separating Data and Control



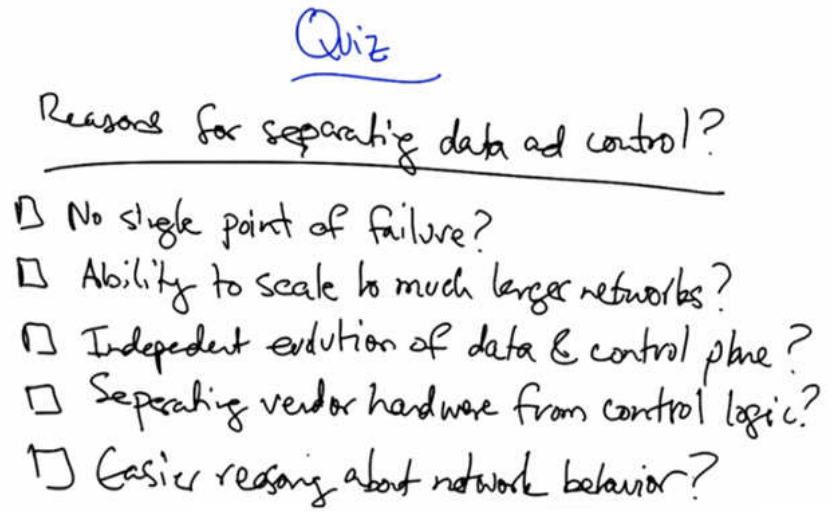
Let's quickly review the difference between the Control plane and the Data plane. The control plane is the logic that controls forwarding behavior. Examples of control plane functions include routing protocols as well as logic for configuring network middle boxes. Now, a routing protocol might compute shortest paths or a topology, but ultimately, the results of such computations must be installed in switches that actually do the forwarding. The forwarding table themselves and specifically the actions associated with forwarding traffic according to the Control plane logic is what constitutes the data plane. So examples of data plane functions include forwarding packets at the IP layer and doing things like switching at layer two. So, to reiterate, routing protocol functions that compute the paths are control plane functions, whereas the act of actually taking a packet on an input port and forwarding it to an output port, is a data plane function.



So why is separating the data and control planes a good idea? The first reason is independent evolution and development. Thus, software control of the network can evolve independently of the network hardware. The second reason that separating data and control plane is a good idea is the opportunity to control the network behavior from a high-level software program. Controlling the network from a high-level program in theory allows network operators to debug and check network behavior more easily than in the status quo where network behavior is determined by the distributed low level configuration across hundreds of switches and routers.

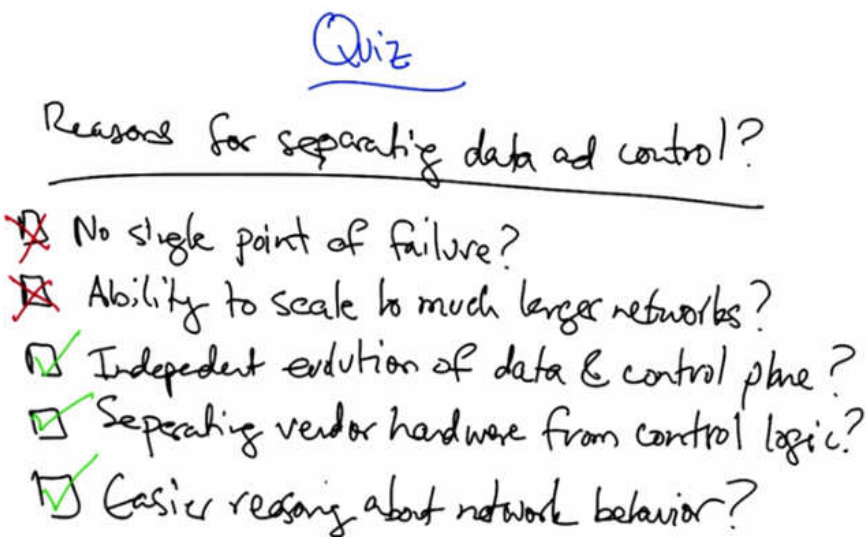
The separation of data and control provides opportunities for better network management in data centers by facilitating such network tasks as virtual machine migration to adapt to fluctuating network demands. In Routing, the separation of data and control provides more control over decision logic. In Enterprise networks, SDN provides the ability to write security applications such as applications that manage network access control. In Research networks, the separation of data and control effectively allows us to virtualize the network, so that research networks and experimental protocols can co-exist with production networks on the same underlying network hardware.

Reasons for Separating Data and Control



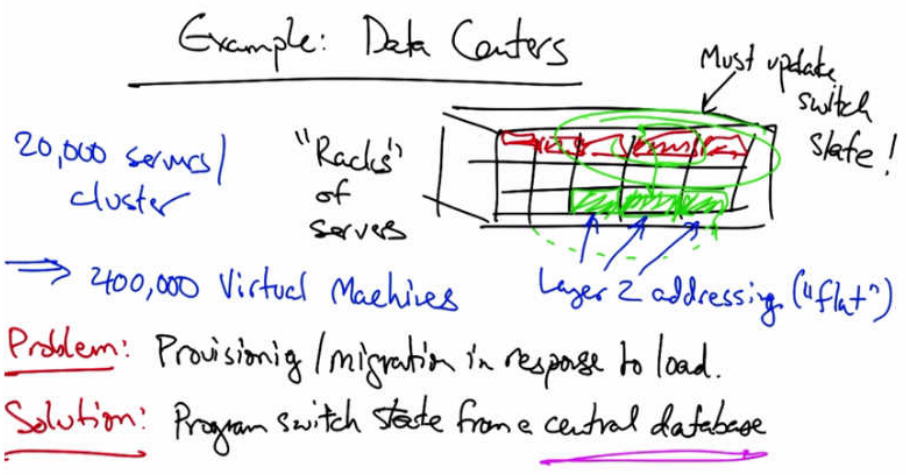
So as a quiz, what are some of the reasons for separating the control and data planes? Eliminating a single point of failure? Ability to scale to much larger networks? Independent evolution of the data and control plane? Separating vendor hardware from control logic? Or ease of reasoning about network behavior? Please check all options that apply.

Reasons for Separating Data and Control



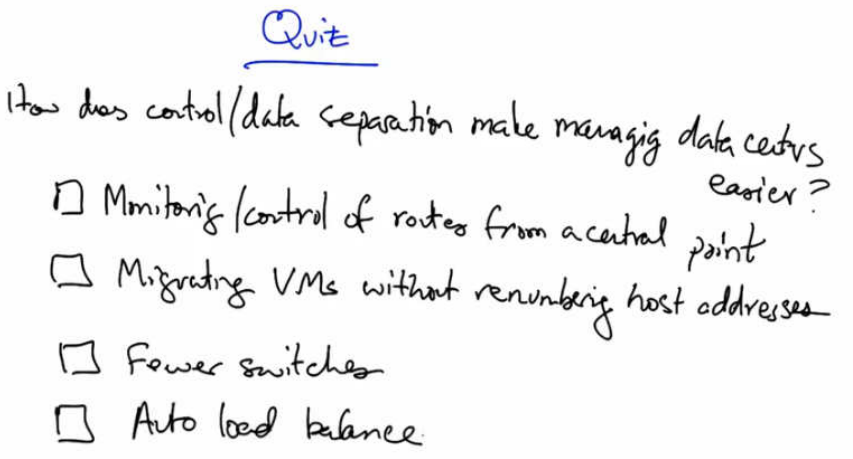
Separating the data and control plane can allow for independent evolution of the data and control plane, separating vendor hardware from the logic that controls the behavior of the network, and the potential to more easily reason about network behavior since the behavior is now controlled from a single, logically-centralized control program. While it's possible that separating the control plane from the data plane could result in architectures that are more fault tolerant or more scalable, the separation of data and control planes does not inherently make the network more fault tolerant or more scalable. Therefore, neither of the first two options apply.

Example Data Centers



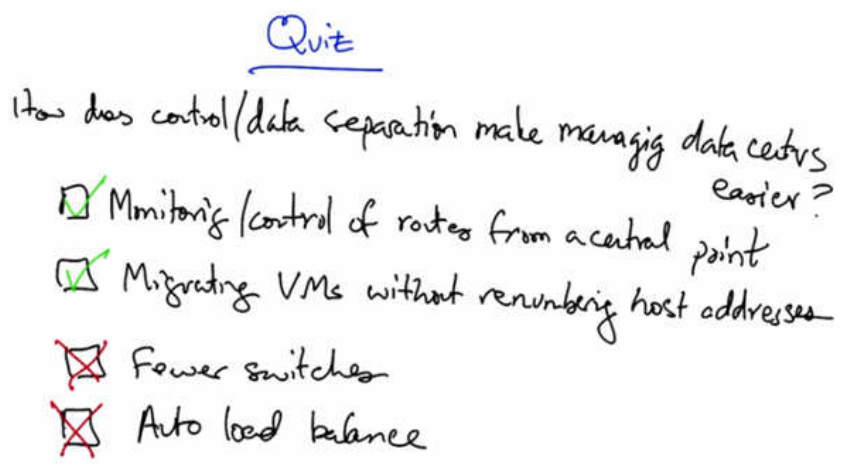
One example where SDN can provide huge wins, is in the data center. A data center typically consists of many racks of servers, and any particular cluster might have as many as 20,000 servers. Assuming that each one of these servers can run about 200 virtual machines, that's 400,000 virtual machines in a cluster. A significant problem is provisioning or migrating these virtual machines in response to varying traffic loads. SDN solves this problem by programming the switch state from a central database. So, supposing I have two virtual machines within the data center that need to communicate with one another, the forwarding state in the switches in the data center ensures that traffic is forwarded correctly. If we need to provision additional virtual machines or migrate a virtual machine from one server to another in the data center, the state in these switches must be updated. Updating the state in this fashion is much easier to do from a central controller or a central database. Facilitating this type of Virtual Machine Migration in the data center is one of the early killer apps of software-defined networking. This type of migration is also made easier by the fact that the servers are addressed with Layer two Addressing, and the entire data center looks like a flat, layer two topology. What this means is that a server can be migrated from one portion of the data center to another without requiring the virtual machine to obtain new addresses. All that needs to happen for forwarding to work is the state of these switches needs to be updated. The task of updating switch date in this fashion is very easy to do when the control and data planes are separate.

Managing Data Centers



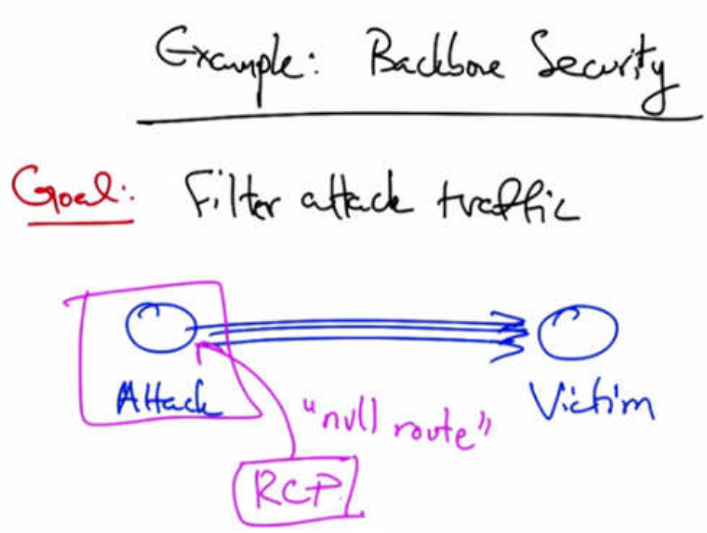
Let's have another quiz on data centers. So how does the control/data plane separation make managing data centers easier? The ability to monitor and control routes from a central point of control? The ability to migrate virtual machines without renumbering host addresses? A requirement for fewer switches? Or making load balance automatic? Please again check all that apply.

Managing Data Centers

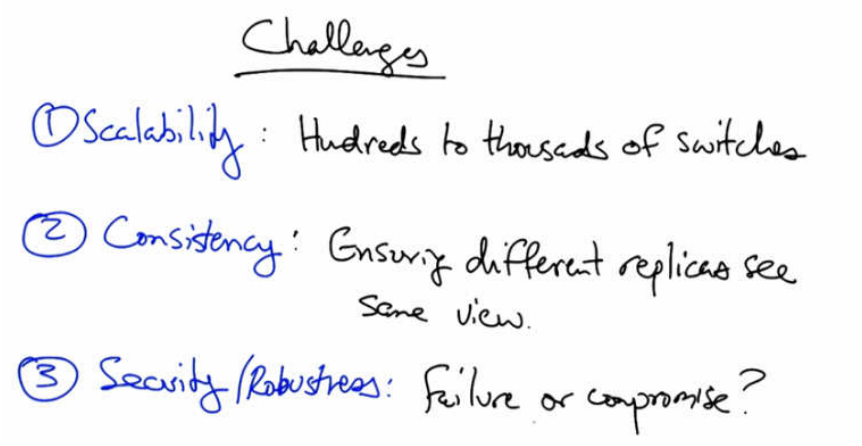


So as we discussed, control/data plane separation can make it easier to manage the data center by monitoring and controlling routes from a central point and allowing virtual machines to be migrated without renumbering host addresses. The control/data plane separation does not inherently make it possible to build a data center with few switches nor does it automatically balance load.

Challenges

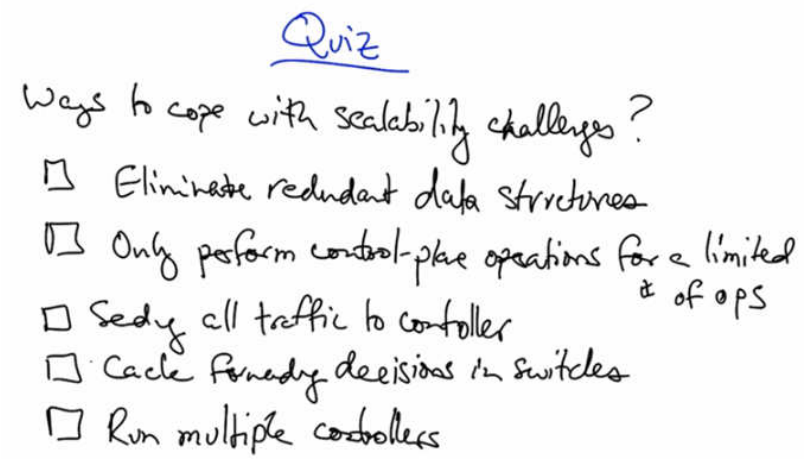


As another example where control and data plane separation comes in handy, let's look at the security of internet backbones, where filtering attack traffic is a regular network management task. Suppose that an attacker is sending lots of traffic towards a victim. In this case a measurement system might detect the attack, identify the entry point, and a controller such as the RCP might install what is called a null route to ensure that no more traffic reaches the victim from the attacker.



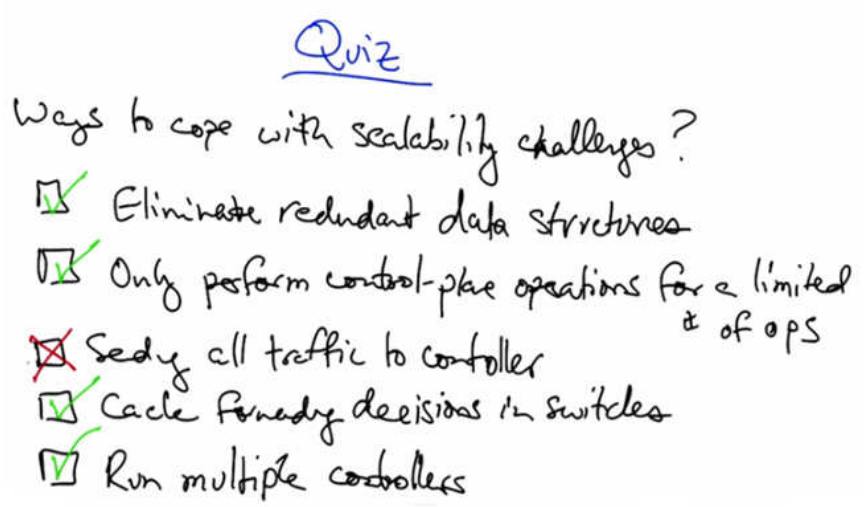
Two fundamental challenges with SDN are scalability and consistency. In an SDN, a single control element might be responsible for many forwarding elements. So control elements might be responsible for hundreds to thousands of switches. Of course, for redundancy and reliability, typically we want to replicate the controller. So while the controller is logically centralized, physically there may be many replicas. And in such a deployment scenario, we need to ensure that different controller replicas see the same view of the network so that they make consistent decisions when they're installing state in the data plane. A final challenge that's also worth mentioning is security, or robustness. In particular, we want to make sure that the network continues to function correctly in the event that a controller replica fails or is compromised.

Coping With Scalability



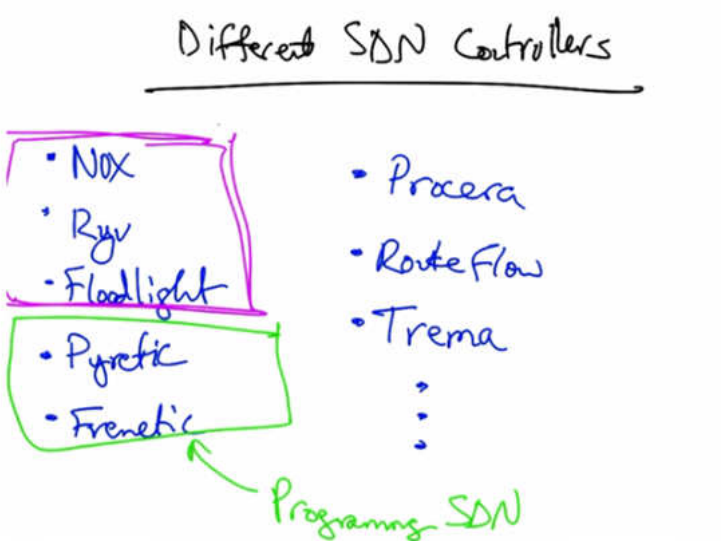
So as a brief quiz or thought question, let's think about some approaches for coping with the scalability associated with control and data plane separation. One could, for example, eliminate redundant data structures in the controller. Or only perform control operations for a limited number of network operations, such as only performing control operations for routing decisions. One might send all traffic through the controller to minimize forwarding decisions that routers and switches need to make. One could cache forwarding decisions in the switches, or run multiple controllers.

Coping With Scalabilty



Eliminating redundant data structures can help save memory in the control program running at the controller. Only performing a fixed number of network management operations, such as routing, can insure that the controller doesn't have to do too much, thereby improving scalability. Caching forwarding decisions that the control plane has already made in the switches can ensure that not too much traffic is redirected to the controller. And running multiple controllers can distribute the load of the control plane across multiple replicas. Sending all traffic to the controller only increases the controller load, and would not help with scale ability.

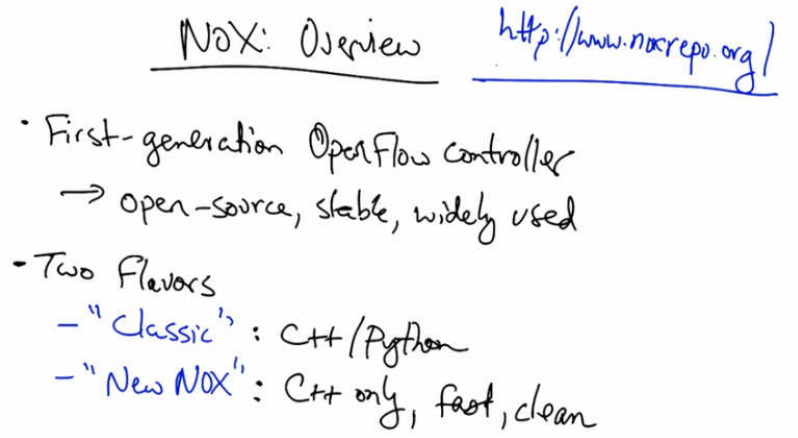
Different SDN Controllers



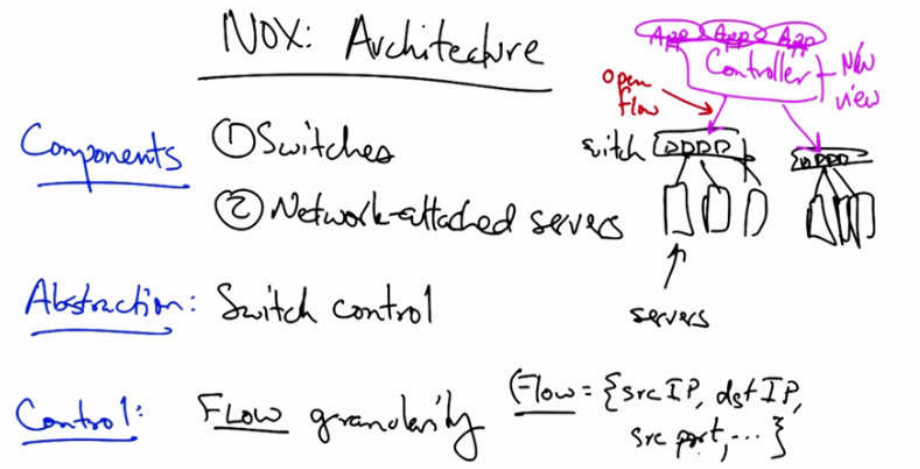
Now that we have a better understanding of the benefits of separating the data and control plane, let's now have a look at the many different options for SDN controllers. There are a number of different SDN controllers that exist, including NOX, Ryu, Floodlight, Pyretic, Frenetic, Procera,

RouteFlow, Trema, and the list goes on. In this lesson, we will explore the merits of these three controllers. And when we get to the lesson on Programming SDN, we will take a close look at Frenetic and Pyretic. Let's now jump in and take a look at these three controllers.

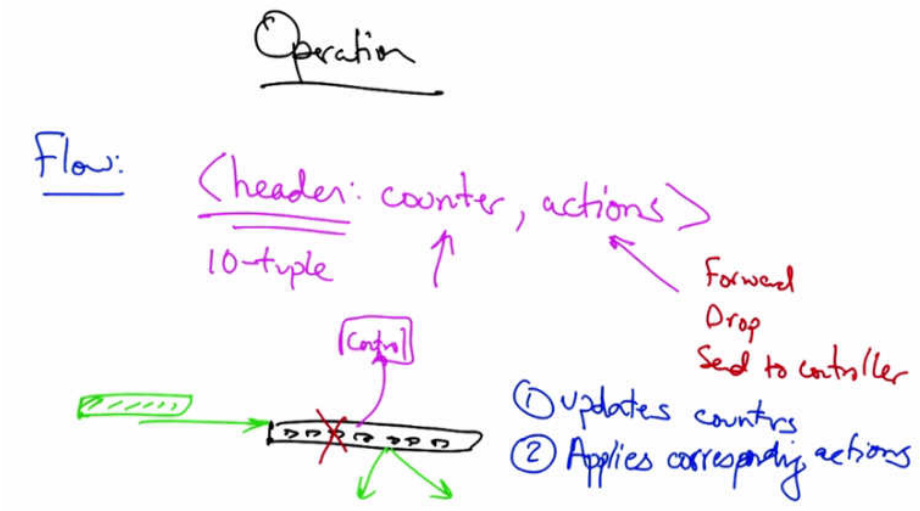
NOX Overview



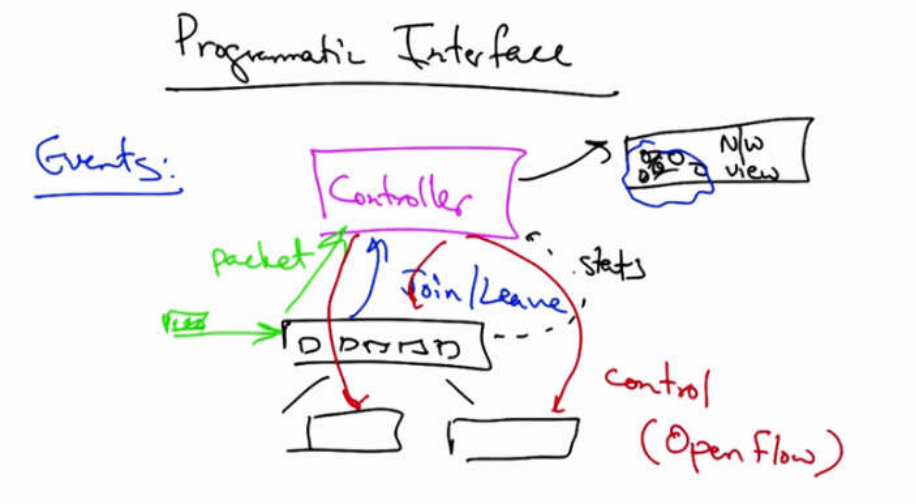
Nox was a first generation open flow controller. It is open source, stable, and widely used. There are two flavors of Nox, Classic Nox and the New Nox. Classic Nox was written in C++ and Python and is no longer supported. The new Nox is C++ only. The Code base is fast, clean, and well supported. More information about Nox is available at noxrepo.org.



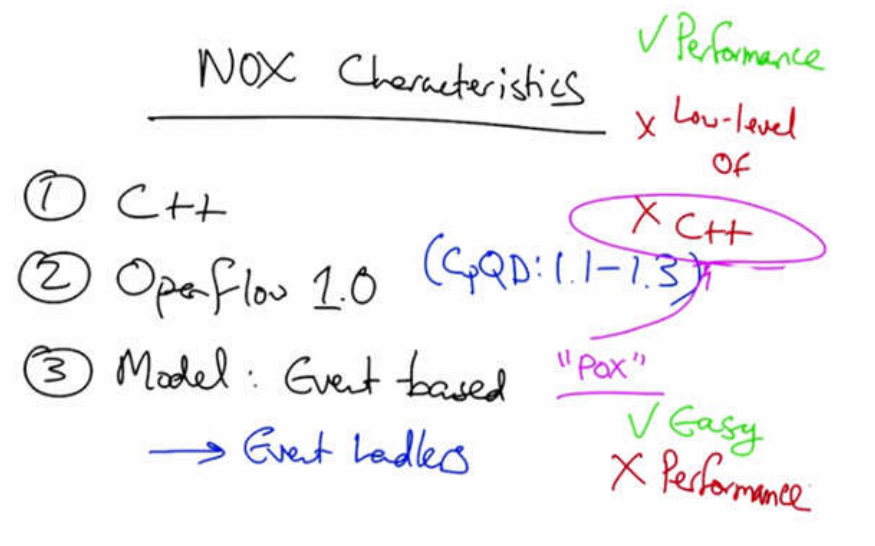
In a Nox network, there may be a set of switches and various network-attached servers. The controller maintains a network view and the controller may also run several applications that operate on that network view. The basic abstraction that NOX supports is a switch control abstraction where OpenFlow is the prevailing protocol. Control is defined at the granularity of flows which are defined by a ten-tuple in the original OpenFlow specification. So depending on whether a particular packet matches a subset of values specified as a flow rule, the controller may make different decisions for packets that belong to different parts of flow space, orr packets that match different subsets of the fields defined by a flow.



A flow is defined by the header or the 10-tuple which I just alluded to, a counter which maintains statistics, and actions that should be performed on packets that match this particular flow definition. Actions might include forwarding the packet, dropping it, or sending it to the controller. When a switch receives a packet, it updates its counters for counting packets that belong to that flow and applies the correspondence actions for that flow, which might include forwarding, dropping or sending to a controller.

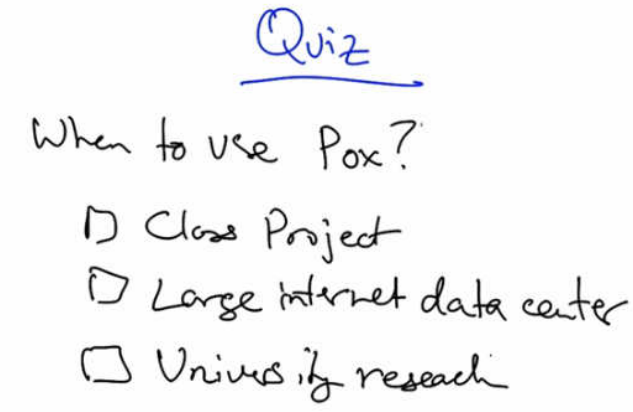


The basic programmatic interface for the Nox controller is based on events. A controller knows how to process different types of events, such as a switch, join, or leave; a packet in or packet received event should the switch redirect packet to controller; as well as various statistics. The controller also keeps tracks of a network view, which includes a view of the underlying network topology, and it also speaks a control protocol to the switches in the network. That control protocol effectively allows the controller to update the state in the network switches. The Nox controller implements the OpenFlow protocol.



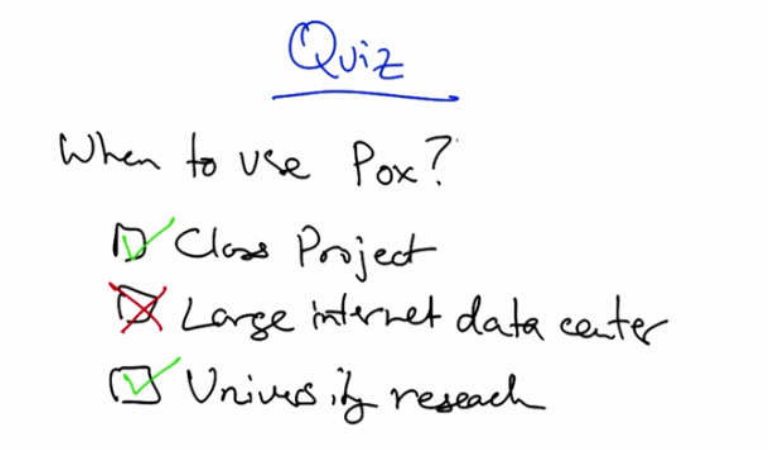
Nox is implemented in C++, and it supports OpenFlow 1.0. A fork of Nox called CPQD supports versions 1.1, 1.2, and 1.3. The programming model is event based and a programmer can write an application by writing even handlers for the Nox controller. NOX provides good performance but requires you to understand and be comfortable with the facilities and semantics of low level OpenFlow commands. Later in this module, we will explore controllers based on pyretic and frenetic that do not have this characteristic. NOX also requires the programmer to write the control application in C++, which can be slow for development and debugging. To address the shortcomings that are associated with development in C++, Pox was developed. Pox is widely used, maintained, and supported. It's also easy to use, and easy to read and write the control programs. Of course, as might come with implementing a controller in python, the performance of Pox is not as good as the performance of Nox.

When to Use Pox



So as a quick quiz, when might you use Pox? In a class project? In a large internet data center? Or in a university research project? Please check all that apply.

When to Use Pox



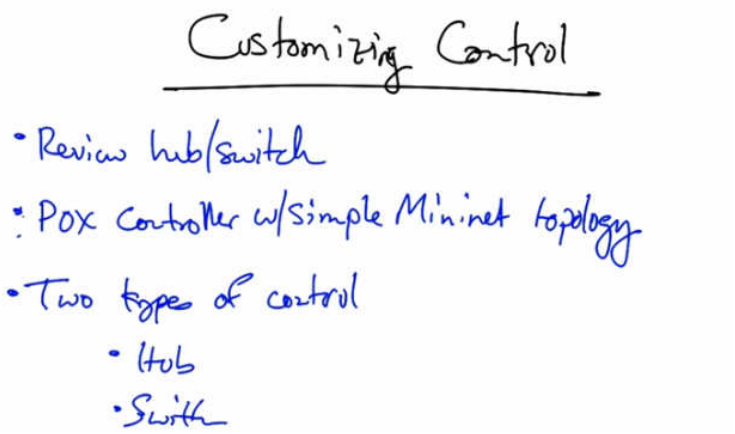
You might use Pox in a class project or in a university research project where there's a need to quickly prototype and evaluate a brand new control application. Pox is less applicable in a large internet data center because it does not perform as well as other controllers.

Ryu, Floodlight, Nox, and Pox

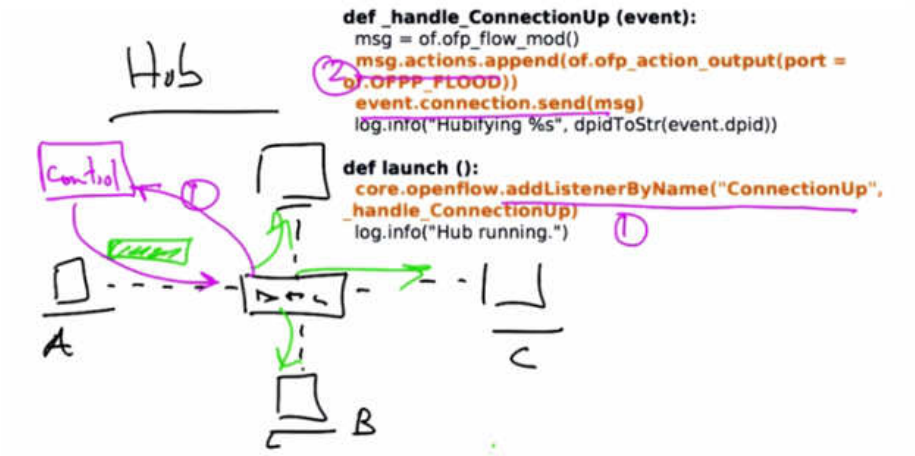
Other controllers include Ryu, which is an open source Python controller. Ryu supports OpenFlow 1.0, 1.2, and 1.3, as well as the Nicira extensions. It also works with Open Stack. The support for the later versions of OpenFlow and the integration with the Open Stack, are advantages over other SDN controllers. Because Ryu is implemented in Python, it still does not perform as well other SDN controllers, such as Nox. Another popular SDN controller is Floodlight. Floodlight is written in Java, it supports Overflow 1.0, and is a fork from the early Beacon controller. Floodlight is maintained by big switch networks. Advantages include good documentation, integration with the REST API, and good performance. Unfortunately, it also has a fairly steep learning curve. So you should use Floodlight if you already know Java, if you need

production level performance and support and you will use the REST API to interact with the controller. So we can compare these two controllers with the two controllers that we already discussed, Nox and Pox. We have controllers in three different languages: Python, Java, and C++. We have controllers that support later versions of OpenFlow, and support Open Stack. And we have controllers that provide better performance, as well as controllers that are easier to use for rapid prototyping. All of these controllers are still relatively hard to use because they entail interacting directly with OpenFlow flow table modifications, which operate at a very low level of matching, on flows and performing specific actions. As we'll see, it's possible to develop programming languages on top of these controllers that make it much easier for a network operator to reason about network behavior. Before we jump into higher level programming languages, however, let's first see how we can use these existing control frameworks to customize network control.

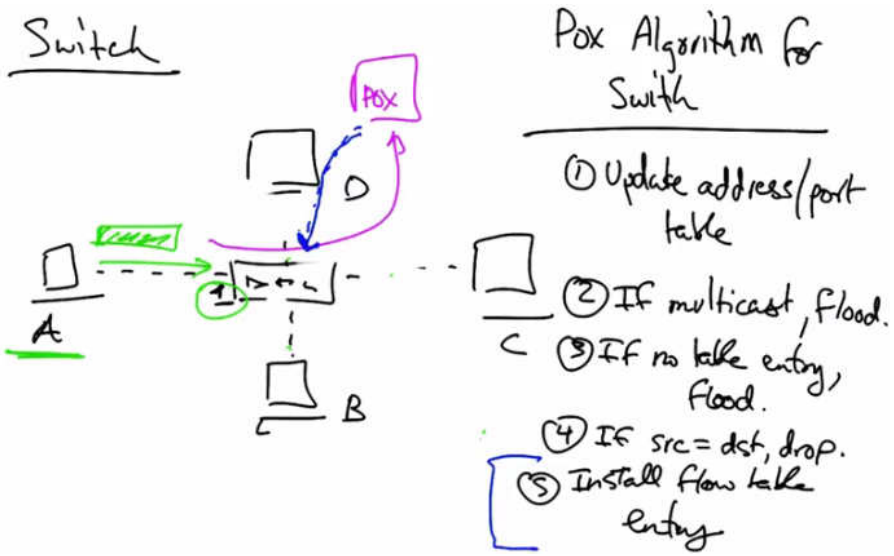
Customizing Control



In this lesson, we will learn how to write control programs to customize network control, we will review the operation of a hub and a learning switch, then we will explore how to use the POX controller to create a simple MiniNet topology, and then we will explore how to customize the Pox controller to perform two types of network control.

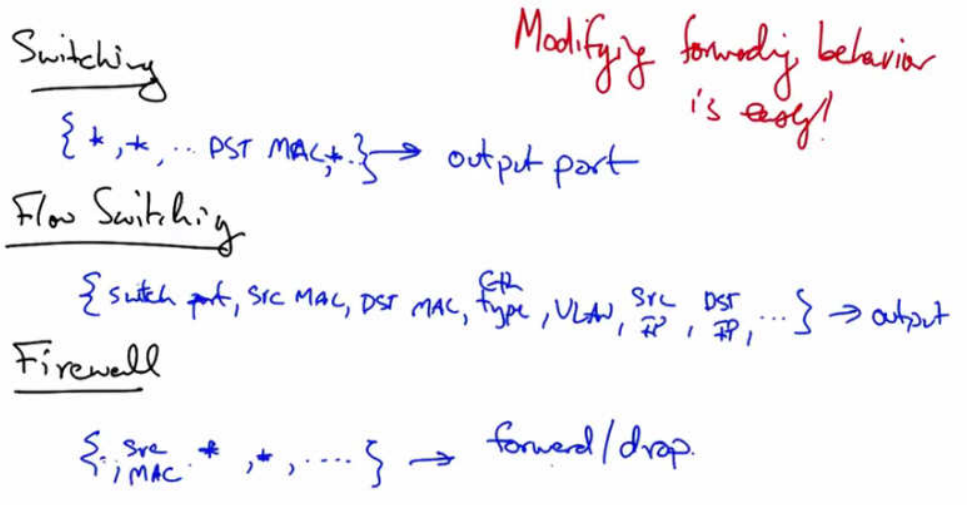


As a review, when a host sends a packet to a hub, the hub maintains no state about which output port a package should be forwarded to reach a particular destination. Therefore, the hub simply forwards the input packet on every output port. In Pox, this code is fairly simple. When the controller starts, it adds a listener that listens for a connection up, which is a connection from a switch. When the switch connects, it simply sends an OpenFlow flow modification back to the switch which says flood all packets out every output port. The first function here involved creates the open-flow massage and the second sends that message back to the switch.

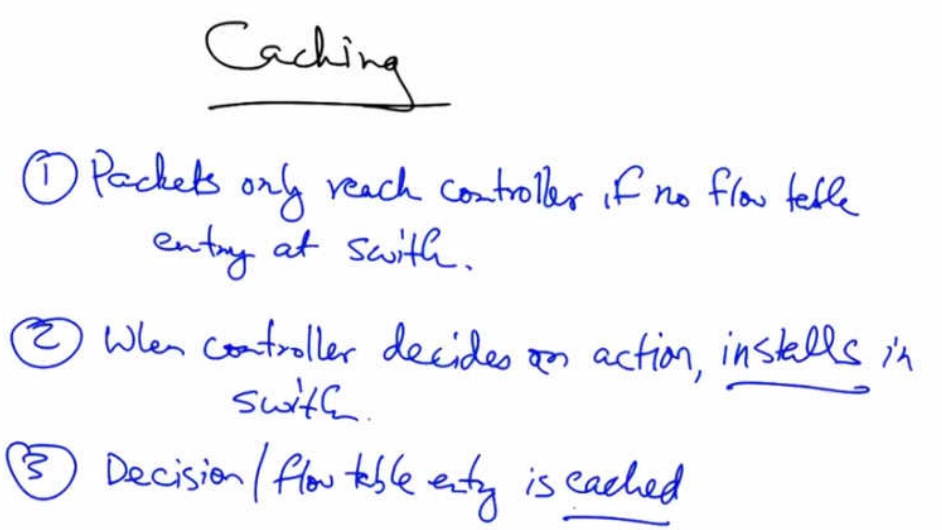


In contrast, a learning switch maintains a switch table that's initially empty. But when a packet arrives on input port one, the switch creates a table that associates host A with output port one such that whenever a subsequent packet is destined for destination A, the switch knows to forward the packet via output port one. I won't show you the full Python code here, but it's fairly simple, and you can go look at the Pox distribution to see the learning switch example. As before, when the first packet arrives at the switch, it is diverted to the controller, at this point, the controller maintains a hash table that maps the address to the output port. When it sees that first packet from Host A, it updates the address and port table. If the packet's a multicast packet, the controller makes a decision to flood that packet on all output ports. Likewise, if there's no table entry for the destination for that packet, the controller also instructs the switch to forward the packet on all output ports. If the source and destination address are the same, the controller instructs the switch to drop the packet. Otherwise, the controller installs the flow table entry corresponding to that destination address and output port. Installing that flow table entry in the switch prevents future packets for that flow from being redirected to the controller. Rather, all subsequent packets on that flow can be handled directly by the switch, since it now knows which output port to send a packet for that particular destination.

Summary

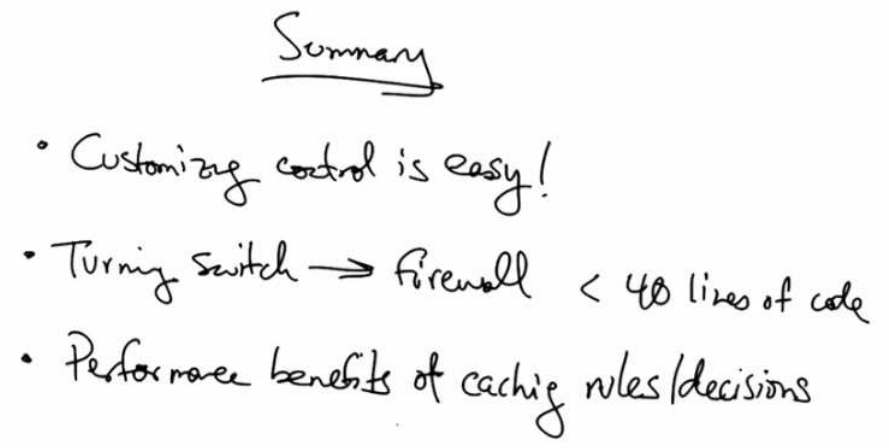


OpenFlow makes modifying forwarding behavior easy because forwarding decisions are based on matches on the OpenFlow 10-tuple. Layer two switching is simply a match on the destination Mac address which has a corresponding action of forwarding out a particular output port. If all of the fields are specified for forwarding out a particular output port, then we have flow switching behavior. If all of the flow specifications are wild carded except for, say, the source MAC address to make a forwarding or drop decision, then we have a firewall. Constructing a firewall is as simple as building a hash table that stores key value pairs, where the table maps a switch and source MAC address to a true or false value depending on whether traffic should be forwarded or dropped. The controller might then only decide to forward traffic if the firewall entry maps to true.



It is important to emphasize the performance implications of caching the decisions at the switch. So, packets only reach the controller if there's no flow table entry at the switch. If on the other hand, there is a float table entry at the switch, then the switch can simply forward the packets

rather than sending them to the controller. So when a controller decides to take an action on a packet, it installs that action as a flow table entry in the switch, and that decision or flow table entry is cached until that flow table entry expires.



In summary, customizing control is easy. We've explored how to use the POX controller to develop alternate control programs. And it's possible to turn a switch into a firewall in less than 40 lines of python code. We also explored the performance benefits of caching rules and decisions to avoid sending too much traffic to the controller. As we know, forwarding performance in a switch is as fast, but whenever we have to send traffic to the controller, it slows things down. So whatever decisions we can cache in the switch will only serve to improve the performance of the network.

CS 6250: Computer Networking

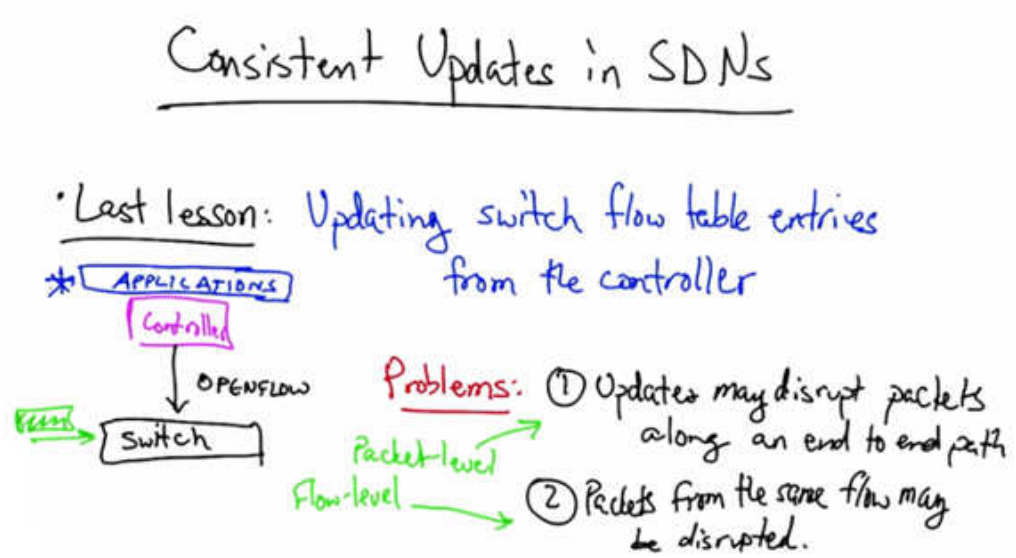
Lecture 9.1: Programming Software Defined Networks

SDN Intro

Now that you know that network management is a tough, complicated process, we're going to take a look at one of the most recent advancements in networking, software defined networking.

Along the way, you're going to complete two exciting projects in Mininet. In the first, you'll write your own virtual switch. In the second, you'll use a programming language designed for software-defined networking to create a firewall.

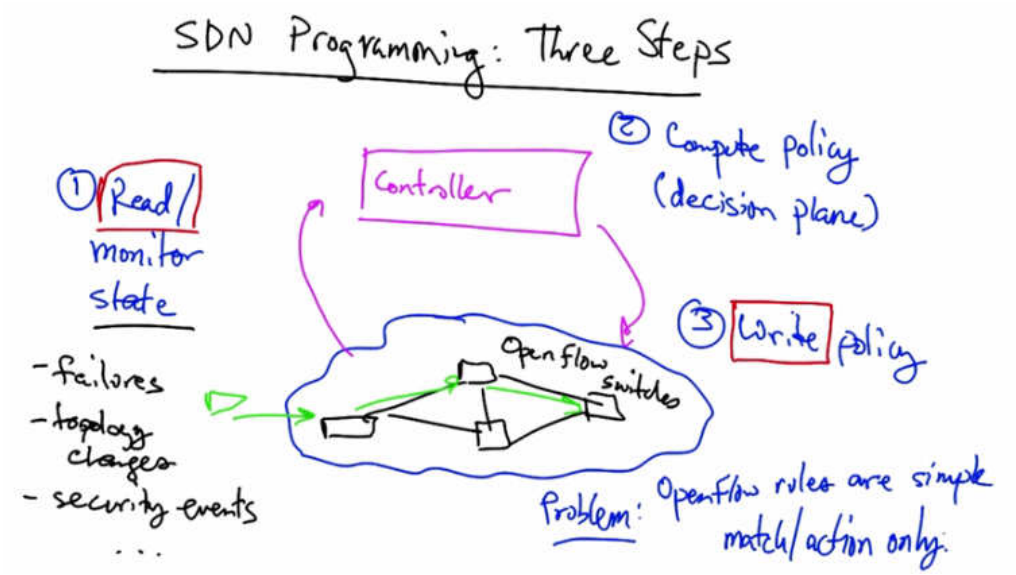
Updates in Software Defined Networks



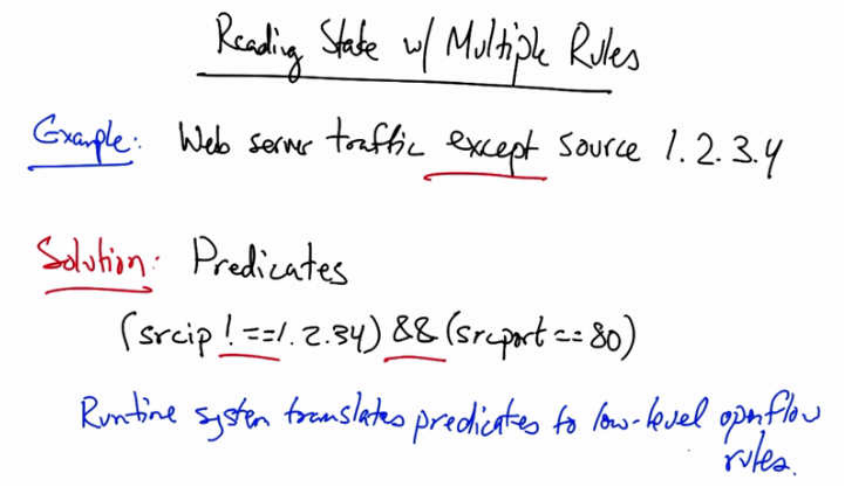
In this lesson we'll be exploring consistent updates in SDN's. As a reminder from the last lesson, we looked at how to update switch flow table entries using OpenFlow control commands from the control. The OpenFlow API, however, does not provide specific guarantees about the level of consistency that packets along an end-to-end path can experience. So for example, updates to multiple switches along a path in a network that occur at different times may result in problems such as forwarding loops. Additionally, if updates to the switches along an end-to-end path occur in the middle of a flow, packets from the same flow may be subjected to different network states. These two problems are known as consistency problems. The first problem is known as a packet level consistency problem, and the second problem is known as a flow level consistency

problem. In this lesson, we will explore these problems in more detail and look at various approaches to guaranteeing consistent updates in SDNs. To think about consistency properly, we first need a notion of a high level programming model that sits on top of what we would call the southbound interface. We'll talk about how to write applications that use the controller interface that we learned about in the last lesson that can rely on a better notion of consistency than existing controller platforms currently provide. Let's first think about how we want to program these applications and what type of abstraction the applications would require from the underlying control interface.

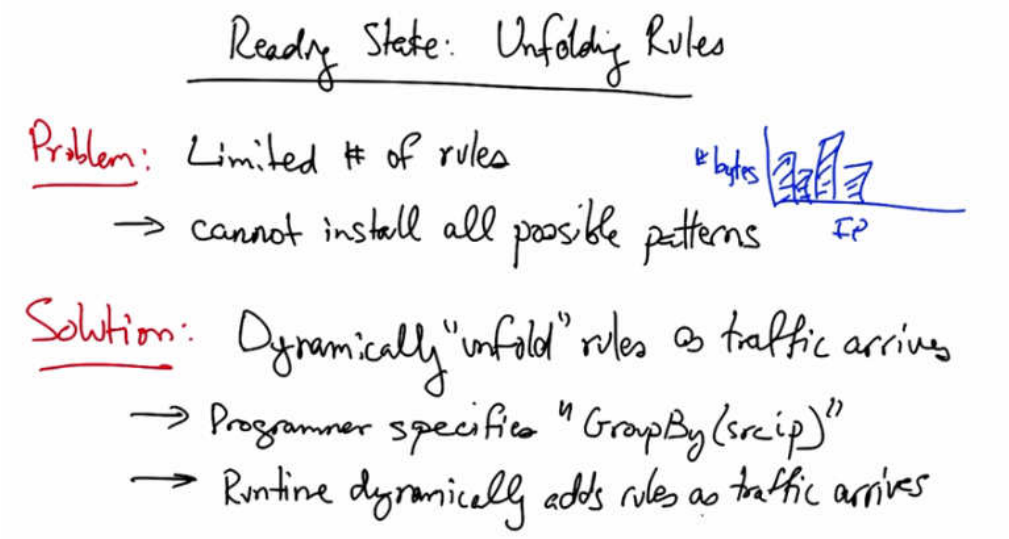
SDN Programming Introduction



Let's consider a network of SDN switches, such as OpenFlow switches, and a controller that is controlling those switches, and let's assume that we would like to write a program using this interface. We can think about this programming as proceeding in three steps. The first is that the controller needs to read or monitor network state, as well as various events that may be occurring in the network. These events may include failures, topology changes, security events, and so forth. The second step is to compute the policy based on the state that the controller sees from the network. This is effectively what we talked about last time, is the role of the decision plane, in deciding what the forwarding behavior of the network should be, in response to various states that it reads from the network switches. The third step is to write policy back to the switches by installing the appropriate flow table state into the switches. Consistency problems can arise in two steps. First, the controller may read state from the network switches at different times, resulting in an inconsistent view of the network-wide state, and second, the controller may be writing policy as traffic is actively flowing through the network, which can disrupt packets along an end-to-end path or packets that should be treated consistently because they're part of the same flow. Both reading and writing networks state can be challenging because OpenFlow rules are simple match action predicates, so it can be very difficult to express complex logic with these rules. If we want to read state that requires multiple rules, expressing a policy that allows us to read such a state can be complicated without more sophisticated predicates.

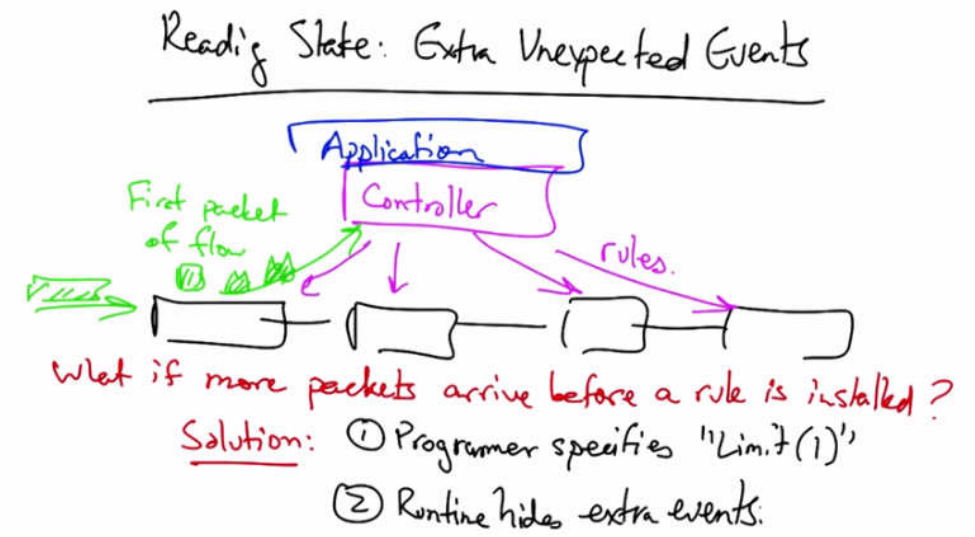


For example, let's suppose that when we are reading state, we'd like to see all web serving traffic except for source 1.2.3.4. Simple match action rules do not allow us to express such exceptions. As a solution to this problem, we need a language primitive that allows us to express predicates. Here is a simple statement that has several predicates, such as AND and NOT. A runtime system can then translate these predicates into low-level OpenFlow rules, ensuring that they are installed atomically and in the right order. Another problem that arises is that switches only have limited space for rules. It's simply not possible to install all possible rule patterns for every set of flows that we'd like to monitor.

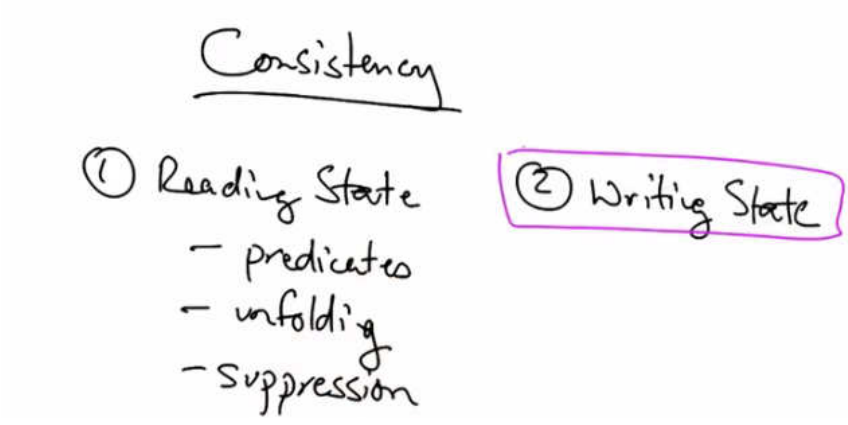


For example, if we'd like to count the number of bytes for every source IP address and generate a histogram with the resulting traffic, we would potentially need a flow table entry for every possible source IP address. It's simply not possible to install all of these possible rules. The solution is to have the run time system dynamically unfold rules as traffic arrives. A programmer would specify something like a group by source IP address, and the run time system would dynamically add open flow rules to the switch as traffic arrives, thereby guaranteeing that there are only rules in the switch that correspond to active traffic.

Reading Network State

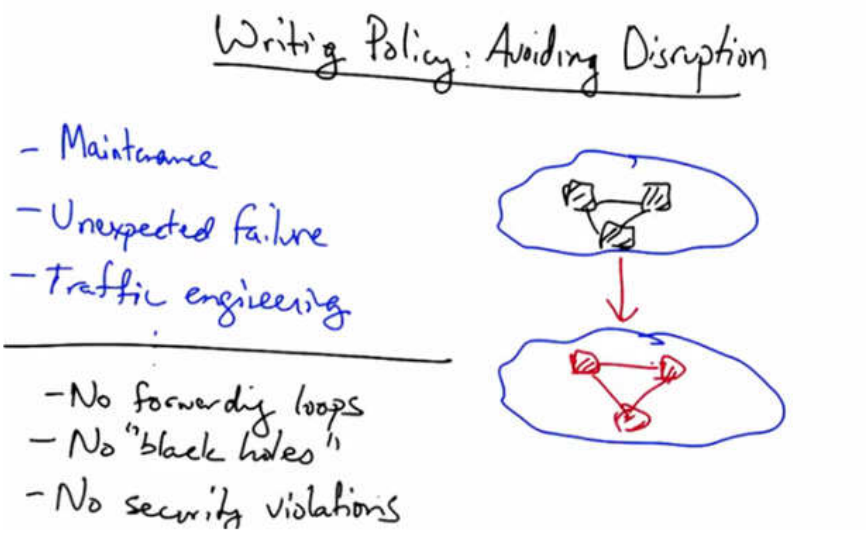


Another problem that arises when reading state is that extra, unexpected events may introduce inconsistencies. A common programming idiom is that the first packet goes to the controller and once the controller figures out what policy to apply for that flow, the controller then installs rules in the switches, in the network, corresponding to that flow. What if more packets should arrive at the switch before the controller has a chance to install rules for that flow? At this point, multiple packets may reach the controller, but the application it is running on top of the controller may not need or want to see these additional packets. So, the solution is to have the programmer specify by a high level language a limit of one, indicating that the application should only see the first packet of the flow and that the subsequent packet should be suppressed. The runtime system then hides the extra events.

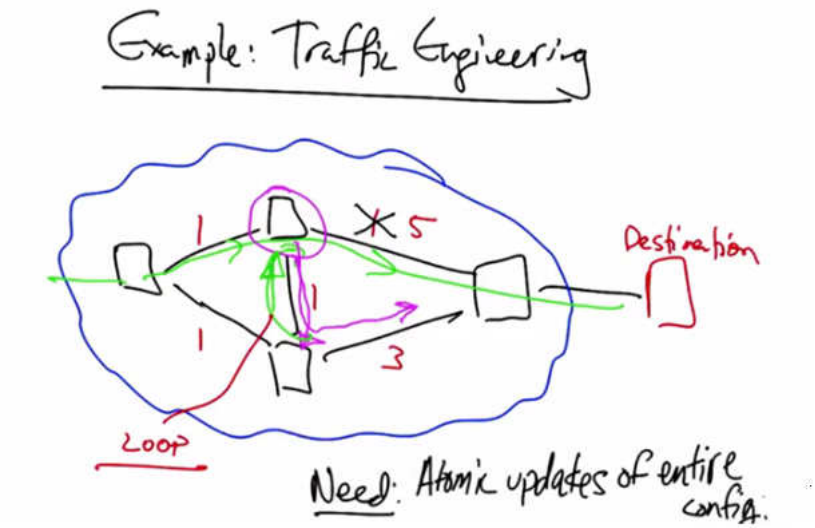


So to remind you where we are, we talked about problems with consistency when reading state from the network, and we talked about three approaches to helping guarantee consistency when reading state: predicates, rule unfolding, and suppression. And let's now talk about primitives that can help maintain consistency when writing state.

Writing Network Policy

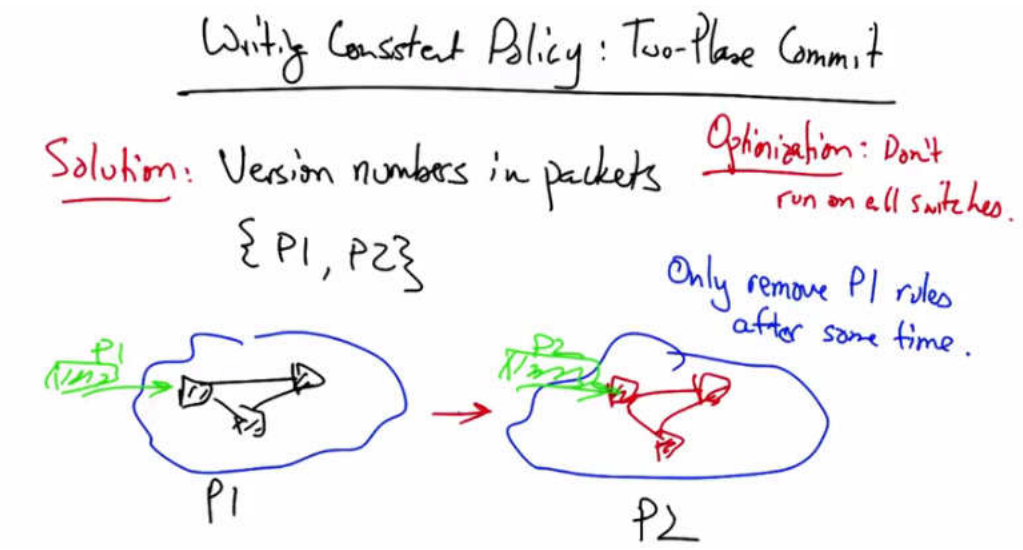


There are many reasons that a controller might want to write policy to change the state and the network switches, including maintenance, unexpected failure, and traffic engineering. Any of these network tasks involve or require updating state in the network switches, and when that state transition happens, we want to make sure that forwarding remains correct and consistent. In particular, we would like to maintain the following invariance. There shouldn't be any forwarding loops and there shouldn't be any black holes whereby a router or switch receives a packet and doesn't know what to do with it. There also shouldn't be cases where traffic is going where it shouldn't be allowed to go because of the network being in an inconsistent state.



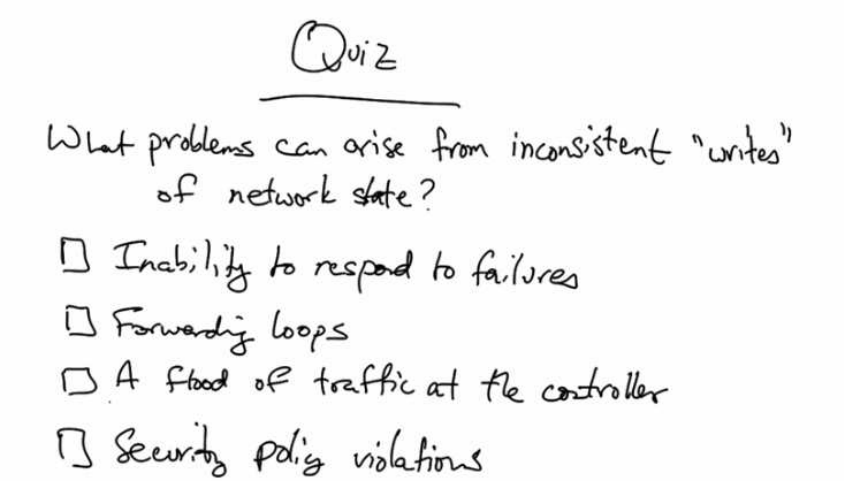
Let's now consider an example of what might happen when policies are written to the network if they're written in an inconsistent fashion. Let's consider a case where we have a network that is performing shortest routing to some destination, and the link weights are as shown here in the figure. Traffic in the network would flow along the path shown in green. Let's suppose now that

an operator wants to change the network state to shift traffic off of this link. He could do so by updating the link weight. In doing so, the new shortest path from this top router would be as follows. But, what if the state in the top switch occurred before the state in the bottom switch could be updated? In this case, we would have a potential forwarding loop. Traffic would proceed to the bottom switch. But the bottom switch would still have the old network state and would continue to forward traffic to the top switch, resulting in a forwarding loop. If rules are installed along a path out of order, packets may reach a switch before the new rules do. So, in this type of model we would have to think about all possible packet and event orderings to ensure that consistent behavior resulted. So we need atomic updates of the entire configuration.



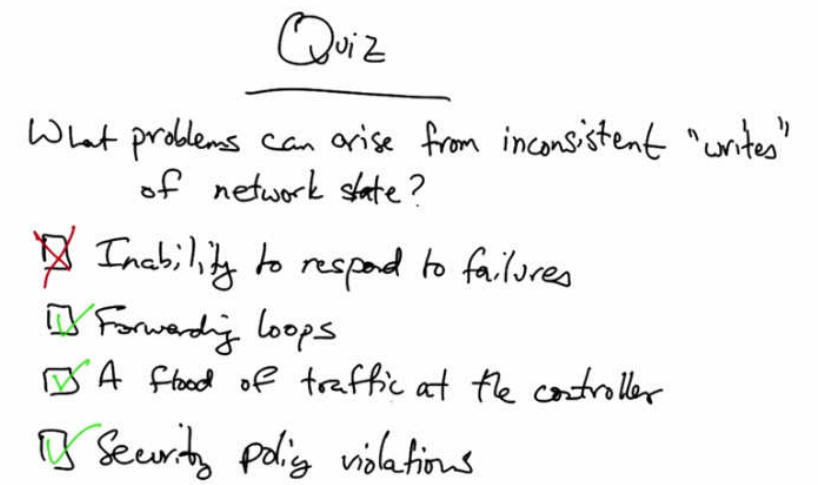
The solution to this problem is to use a two phase commit so that packets are either subjected to the old configuration on all switches, or to the new configuration on all switches. But packets aren't subjected to the new policy on some switches and the old policy on others. The idea is to tag the packet on ingress so that the switches maintain copies of both P1 and P2 for some time. When all switches have received rules corresponding to the new policy, then incoming packets can be tagged with P2. After some time, when we're sure that no more packets with P1 are being forwarded through the network, we can only then remove the rules corresponding to policy P1. Now, the naive version of two-phase commit, requires doing this on all switches at once, which essentially doubles the rule space requirements since we have to store the rules for both P1 and P2. We can limit the scope of the two phase commit by only applying this mechanism on switches that involve the affected portions of the traffic or the affected portions of the topology.

Inconsistent Policy Write Quiz



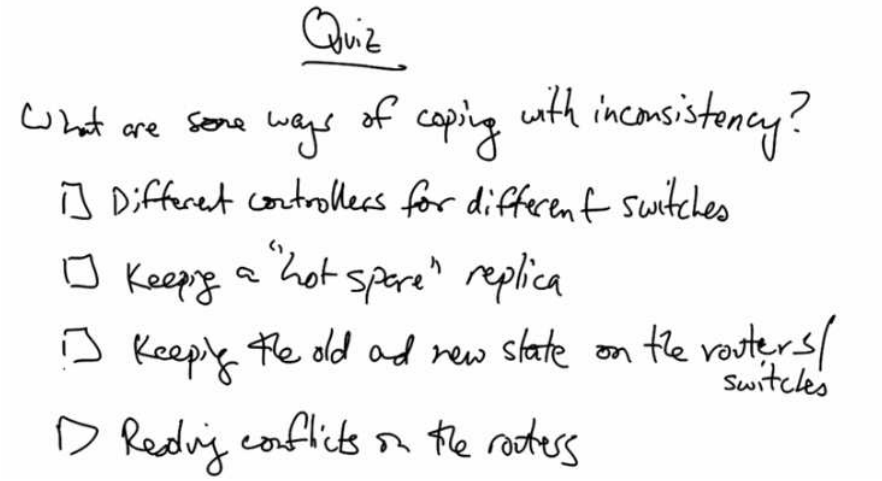
So here's a quick quiz. What types of problems can arise from inconsistent applications of writing policy? Inability to respond to failures, forwarding loops, a flood of traffic at the controller, or security policy violations?

Inconsistent Policy Write Quiz



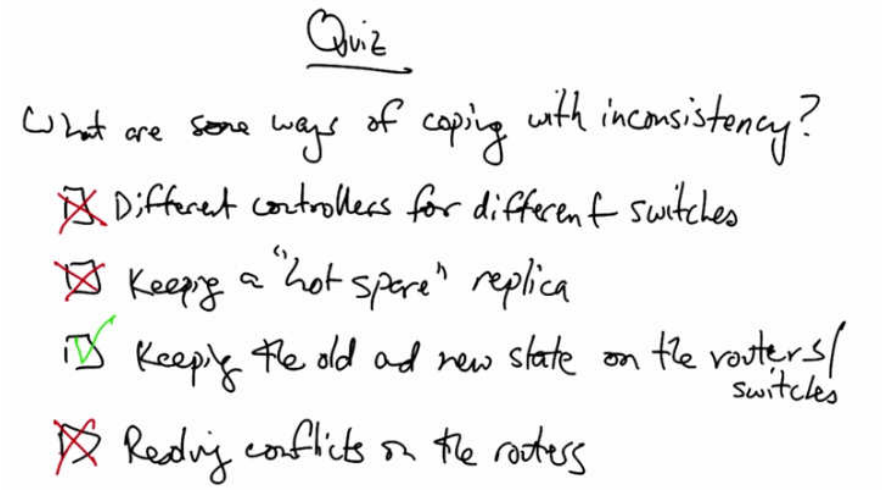
Inconsistent writes can result in forwarding loops or security policy violations where traffic ends up going to parts of the network where it shouldn't go as a result of inconsistent switch state. The ability to respond to failures is orthogonal to consistency. A flood of traffic at the controller technically involves problems with reading state in a consistent fashion. But since there also involves a step where the controller writes state to the switches while packets are still arriving at the controller, I would consider that answer to be correct as well.

Coping With Inconsistency Quiz



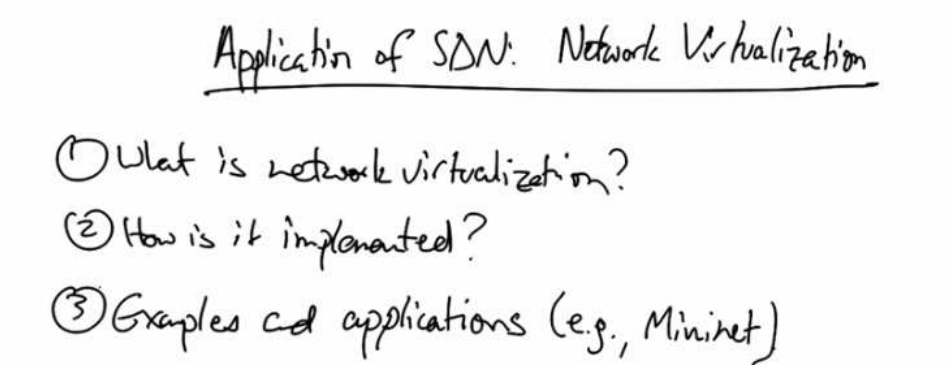
What are some approaches to coping with inconsistency? Running different controllers for different switches? Keeping a "hot spare" replica that has a complete view of the network state? Keeping the old and new state on the routers and switches and switching over only when all of the switches have received the new state? Or relying on the routers themselves to resolve the conflict?

Coping With Inconsistency Quiz

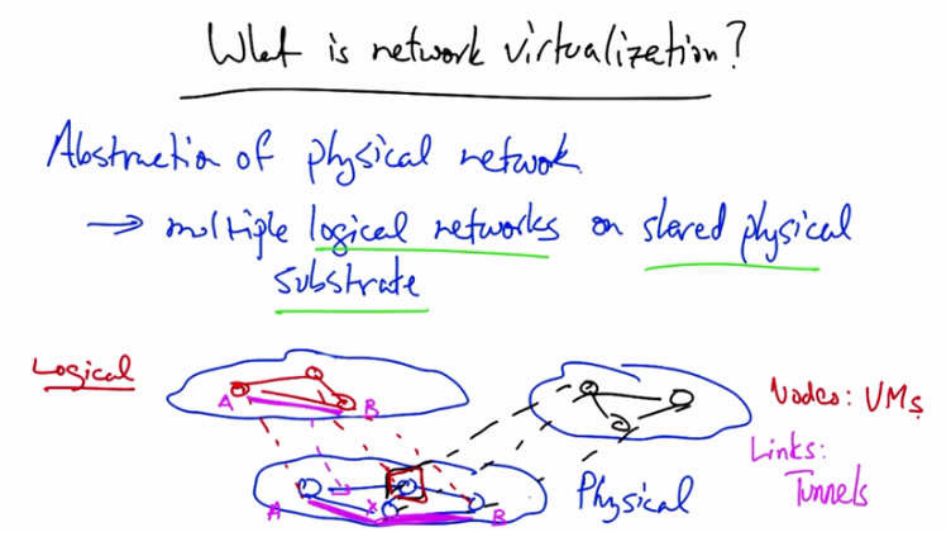


In this case, there is only one correct answer, which is keeping the old and new state on the routers and switches. This is the two-phase commit approach that we talked about. Running different controllers for different switches could obviously result in an inconsistent state, since each of those controllers may be making independent decisions. Keeping a "hot spare" replica does no good if the replica also writes state inconsistently to the network. And resolving conflicts on the routers also doesn't work because no router has a complete view of the network state.

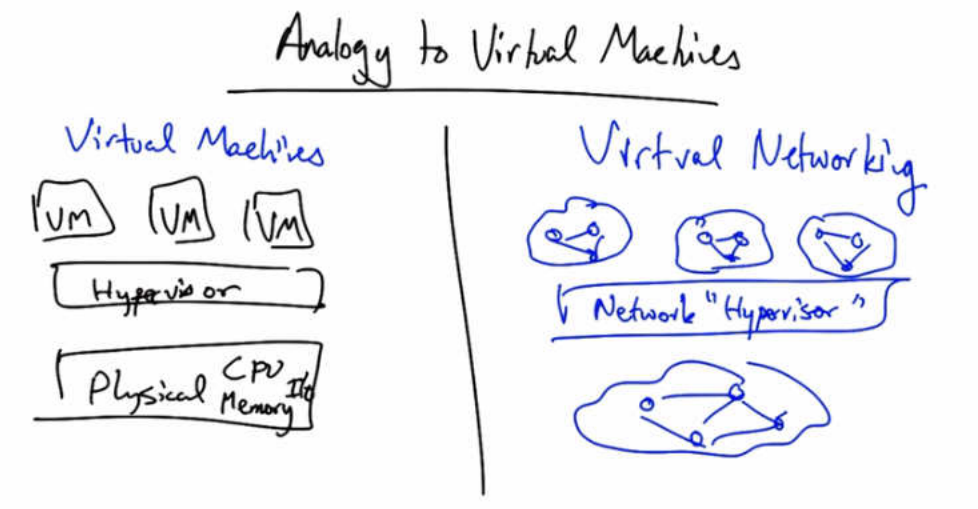
Network Virtualization



Let's now talk about an application of software defined networking, which is network virtualization. So we'll talk first about what network virtualization is, then we'll talk about how it's implemented, and then we'll talk about some examples and applications, such as Mininet.

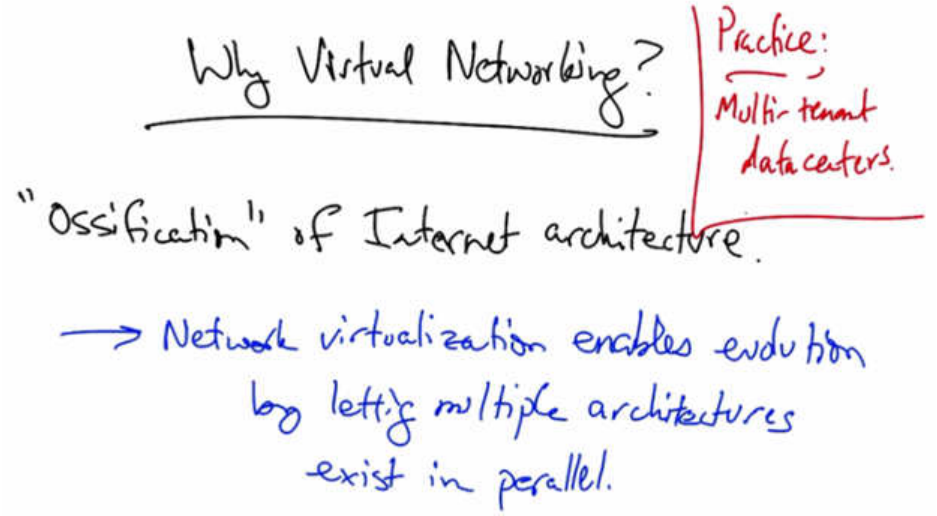


So network virtualization is simply an abstraction of the physical network, where multiple logical networks can be run on the same underlying shared physical substrate. For example, a logical network might map a particular network topology onto the underlying physical topology. And there might be multiple logical networks that map onto the same physical topology. And these logical networks might actually share nodes and links in the underlying physical typology, but each logical network has its own view as if it were running its own private version of the network. Now you can see from this picture that the nodes in the physical network need to be shared or sliced. So the nodes in the physical topology might be virtual machines. Similarly, a single link in the logical topology might map to multiple links in the physical topology. The mechanism to achieve these virtual links is typically through tunneling. So a packet that's destined from A to B in the logical topology, might be encapsulated in a packet that's destined for node X first, before the packet is decapsulated and ultimately sent to B.



It may also be easy to understand virtual networking as an analogy to virtual machines, which you may be familiar with already. So in a virtual machine environment, we have virtual machines where a hypervisor arbitrates access to the underlying physical resources, providing to each virtual machine the illusion that it's operating on its own dedicated version of the hardware. Similarly, with virtual networking, a network hypervisor of sorts arbitrates access to the underlying physical network to multiple virtual networks, providing the illusion that each virtual network actually has its own dedicated physical network.

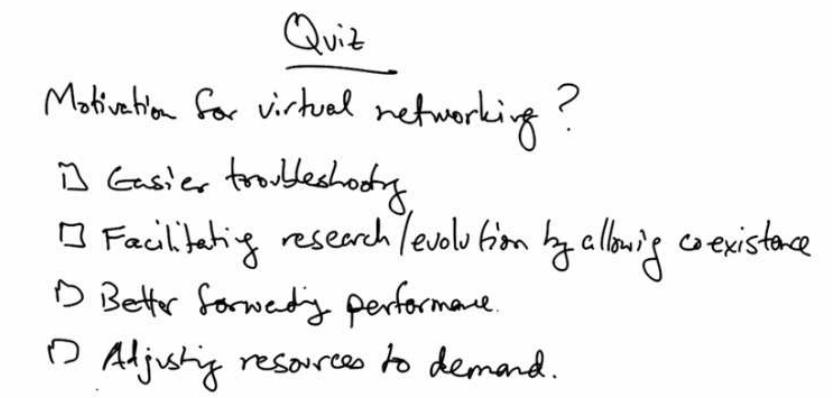
Why Use Network Virtualization



One of the main motivations for the rise of virtual networking was the "ossification" of the internet architecture. In particular because the internet protocol was so pervasive, it made it very difficult to make fundamental changes to the way the underlying internet architecture operated. There was a lot of work on overlay networks in the 2000's, but one size fits all network architectures were very difficult to deploy. So rather than try to replace existing network architectures, network virtualization was intended to allow for easier evolution. In other words,

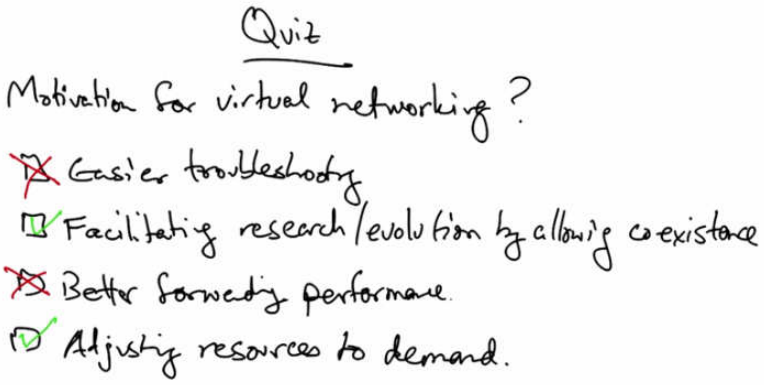
network virtualization enables evolution because we didn't have to pick a winner for a replacement for IP. We could instead let multiple architectures exist in parallel. Now, this was sort of a green field view of why virtual networking was potentially a good idea. In practice, network virtualization has really taken off in multi-tenant data centers where there may be multiple tenants or applications running on a shared cluster of servers. Well known examples of this include Amazon's EC2, Rack Space, and things like Google App Engine. Large service providers such as Google, Yahoo, and so forth, also use network virtualization to adjust the resources that are devoted to any particular service at a given time.

Network Virtualization Quiz



So what are the motivations for network virtualization or virtual networks that we've discussed? Easier troubleshooting? Facilitating research and evolution by allowing coexistence of production networks with experimental ones? Better forwarding performance? And adjusting the resources of the network as demands change? Please check all that apply.

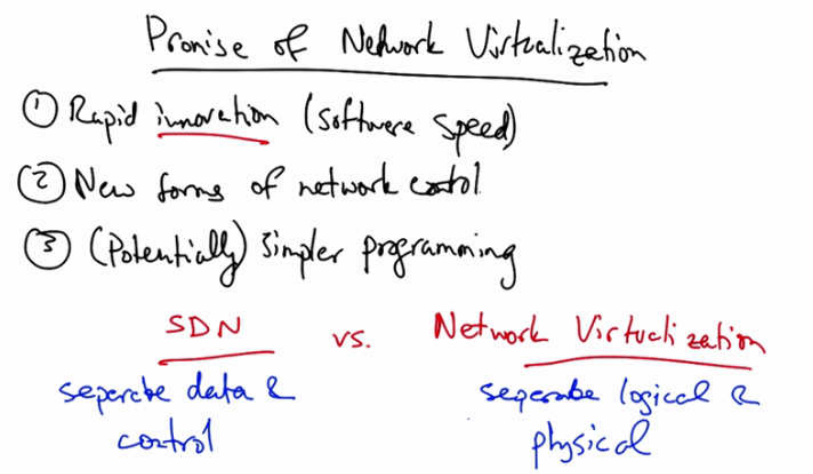
Network Virtualization Quiz



As we discussed, virtual networks can facilitate research in evolution by allowing experimental networks to coexist with production networks. Because the networks are virtual, they can be scaled up and down, adjusting the resources that are devoted to any one particular service as demands change. We discuss this in the context of production networks, such as Google and

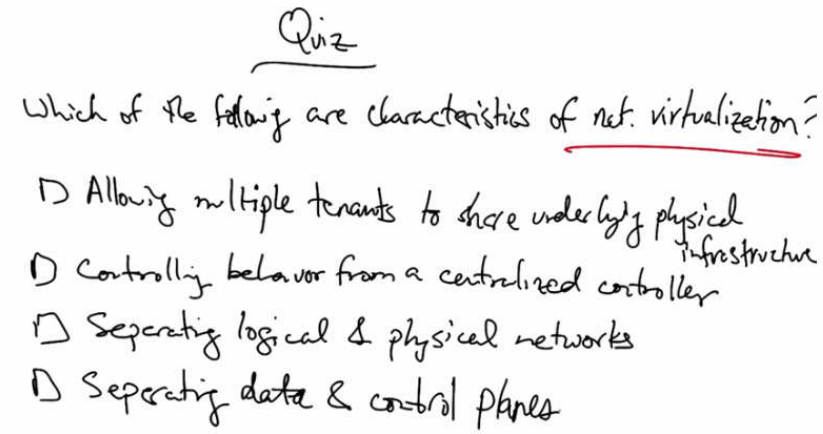
Yahoo. Virtual networks are not inherently easier to troubleshoot, nor do they necessarily provide better forwarding performance. In fact, forwarding performance may be worse, due to the additional level of indirection that has been added.

Network Virtualization Uses SDN



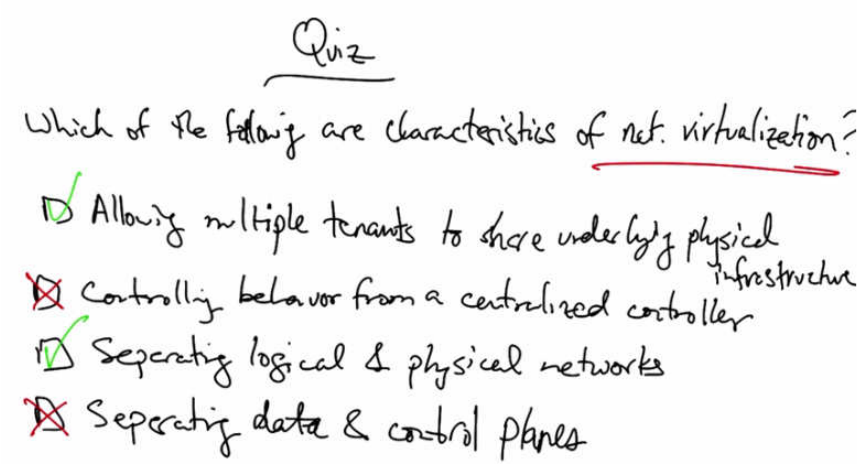
Some of the promised benefits of Network Virtualization are more rapid innovation since innovation can proceed at the rate at which software evolves rather than on hardware development cycles, allowing for new forms of network control and potentially simplifying programming. It is important to make a distinction between Network Virtualization and Software-Defined Networking. Network Virtualization is arguably one of the first killer applications for SDN. And in some sense, SDN is a tool for implementing Network virtualization. But the two are not one and the same. Remember the defining tenant of SDN is the separation of the data and control plane, whereas the defining tenant of Network virtualization is to separate the underlying physical network from the logical networks that lie on top of it. So SDN can be used to simplify many aspects of Network virtualization. But it does not inherently obstruct the details of the underlying physical network.

Characteristics of Network Virtualization Quiz



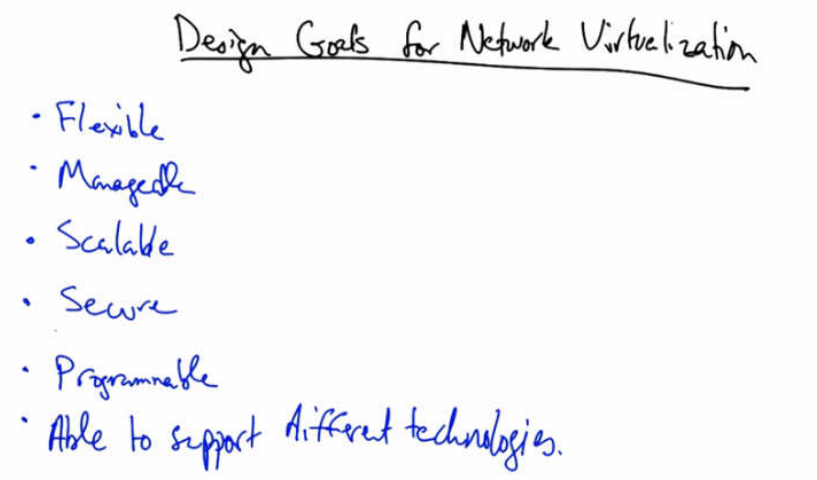
So which of the following are characteristics of network virtualization, but not necessarily characteristics of SDN? Be careful, the distinction between SDN and virtual networks was as we discussed in the previous part of this lesson. Allowing multiple tenants to share underlying physical infrastructure? Controlling behavior from a logically centralized controller? Separating logical and physical networks? Or separating data and control planes? Again, please feel free to check all that apply.

Characteristics of Network Virtualization Quiz



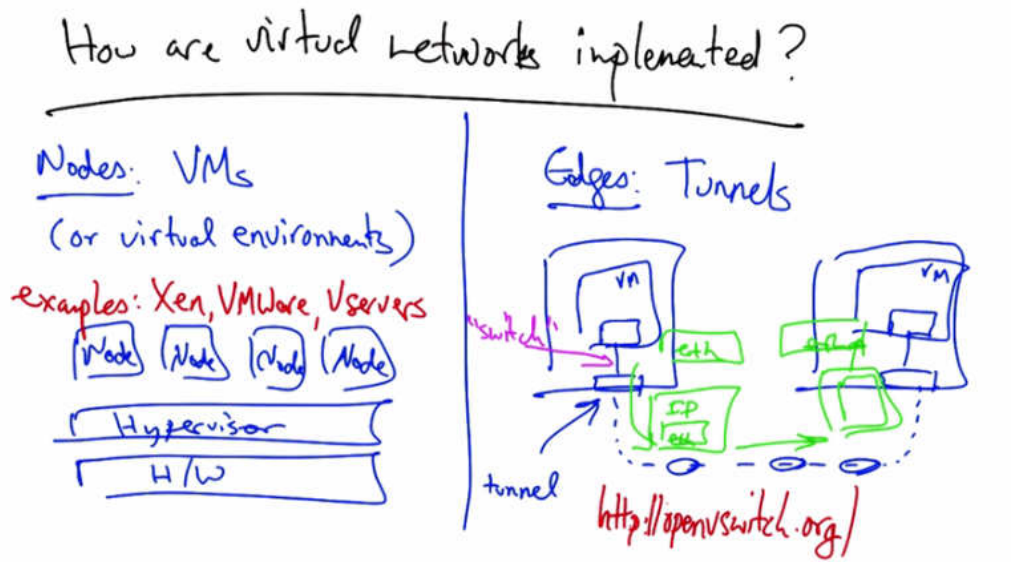
Network virtualization can allow multiple tenants to share the underlying physical infrastructure. And it also separates logical and physical networks. The other two options are defining characteristics of software defined networking, but not of network virtualization.

Design Goals for Network Virtualization



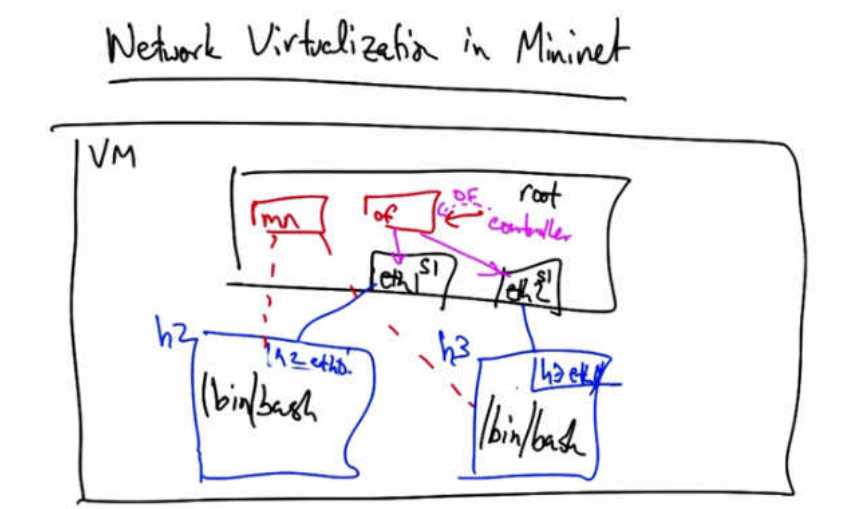
So virtual networks have various design goals. It should be flexible, able to support different topologies, routing, and forwarding architectures, and independent configurations. They should be manageable. In other words, they should separate the policy that a network operator is trying

to specify from the mechanisms of how those policies are implemented. They should be scalable, maximizing the number or coexisting virtual networks. They should be secure by isolating the different logical networks from one another. They should be programmable, and they should be heterogeneous in the sense that they should support different technologies.

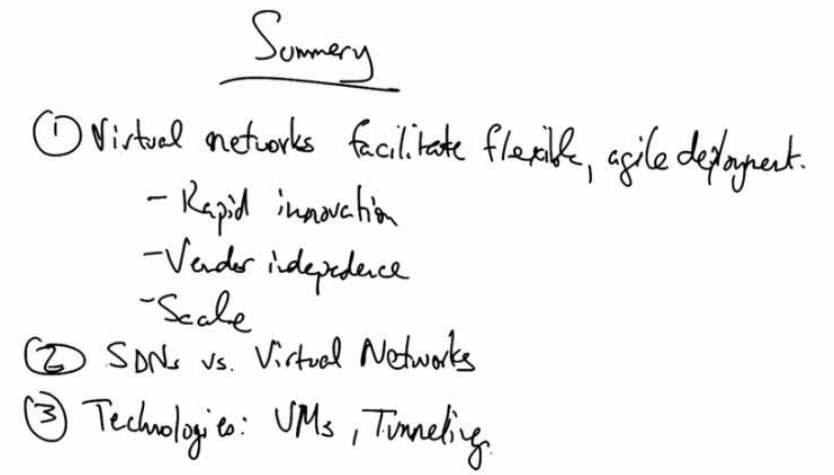


So virtual networks have two components, nodes and edges. The physical nodes themselves must be virtualized. One possible way of virtualizing a node is a virtual machine. A more lightweight way of virtualizing a node is using a virtual environment such as a VServer or a Jail. The hypervisor, or whatever technology is enabling the virtual environment, can effectively slice the underlying physical hardware to provide the illusion of multiple guest nodes or multiple virtual nodes. Examples of node virtualization include virtual machine environments such as Xen or VMware, or what's called OS level virtualization or virtual environments, such as Linux Vservers. Now, in a virtual network, we need to connect these virtual machines. Each virtual machine or virtual environment has its own view of the network stack. And we may want to provide the appearance that these nodes are connected to one another over a Layer two topology, even if they are in fact separated by multiple IP hops. One possible way of doing that is to encapsulate the Ethernet packet as it leaves the VM on the left in an IP packet. The IP packet can then be destined for the IP address of the machine on the right, and when the packet arrives at this machine, the host can decapsulate the packet and pass the original Ethernet packet to the VM or the virtual environment that's residing on that physical node. Each one of these physical hosts may, in fact, host multiple virtual machines or virtual environments, which creates the need for a virtual switch that resides on a physical host. This virtual switch provides the function of networking virtual machines together over a virtual layer two topology. The Linux bridge is an example of a software switch that can perform this type of function. Open Vswitch is another example of software that performs this type of glue function. You can see more information about Open Vswitch on the URL provided here.

Virtualization in Mininet

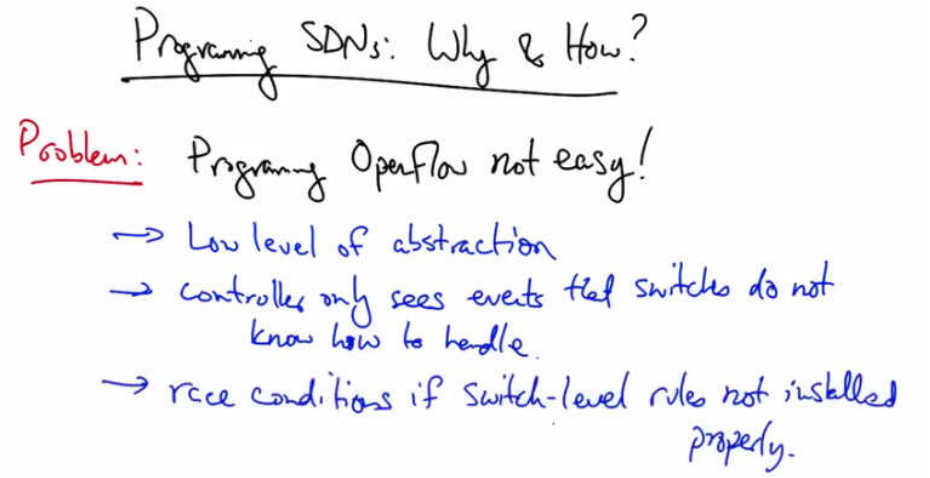


The Mininet tool we have been using in the course is actually an example of network virtualization. We are in fact running an entire virtual network on your laptop. When you start Mininet using the MN script, each host in the virtual network is a bash process with its own network name space. A network name space is kind of like a virtual machine except it's a lot more lightweight. It's in fact called OS Level Virtualization. So, each one of these virtual nodes has its own view of the network stack as shown here with these interfaces. But it has a shared file system and it's not, in fact, running its own independent virtual machine. The root namespace manages the communication between these distinct virtual nodes, as well as the switch that connects these nodes in the topology that you set up. Virtual Ethernet pairs are assigned two name spaces. For example, S1 eth1 is assigned to an interface in H2's network name space. And S1 eth2 is assigned to a network name space in H3's virtual network name space. The OpenFlow switch effectively performs forwarding between the interfaces in the root name space. But because the interfaces are paired, we get the illusion of sending traffic between h2 and h3. When we make modifications to the OpenFlow switch via the controller, we're in fact doing that in the root name space.



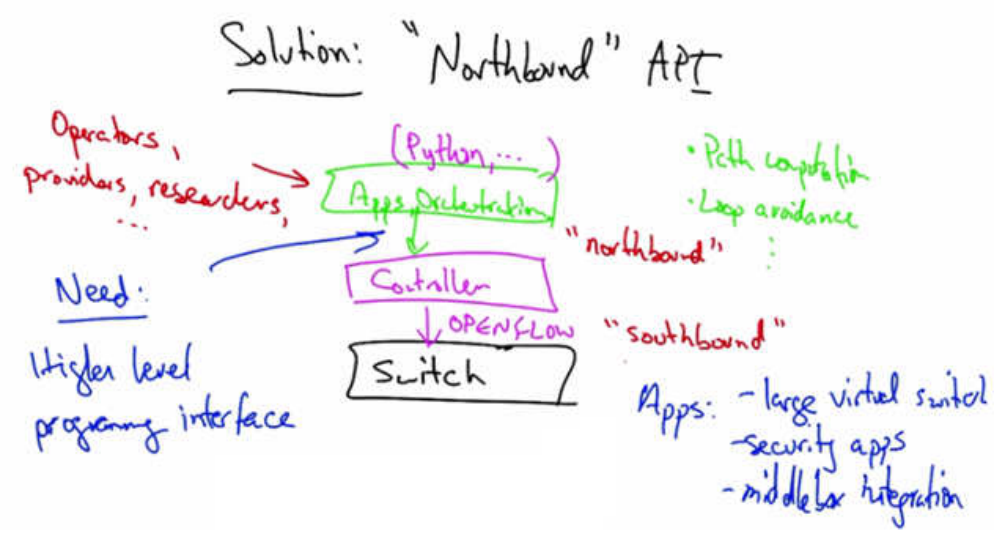
In summary, virtual networks facilitate flexible, agile deployment, by enabling rapid innovation at the pace of software, vender independence, and scale. We talked about the distinction between SDN's and virtual networks, as well as various technologies that enable virtual networks, such as virtual machines for creating virtual nodes and tunneling for creating virtual links.

SDN Programming Difficulty



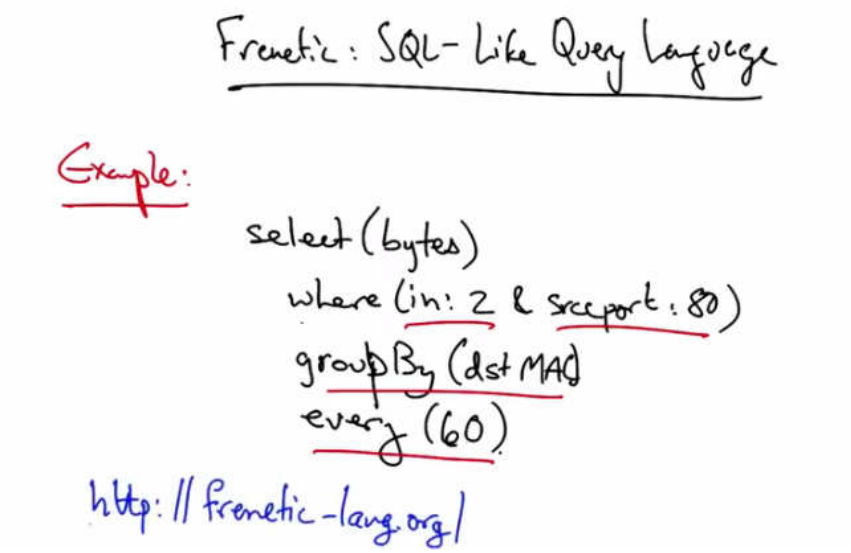
In this lesson we'll talk about the why and how of programming SDNs. Unfortunately, programming OpenFlow is not easy. It offers only a very low level of abstraction in the form of match action rules. The controller only sees events that switches don't know how to handle, and there can be race conditions if switch level rules are not installed properly, as we've already seen in the lesson on consistent updates

SDN Programming Interface



The solution to this is to provide some kind of northbound API, which is a programming interface that allows applications and other kinds of orchestration systems to program the network. So where we have at the low-level the controller updating state in the switch using OpenFlow flow modification rules, we may have applications or orchestration systems that need to perform more sophisticated tasks, such as path computation, loop avoidance, and so forth. But we need a higher-level programming interface that allows these applications to talk to the controller so the application isn't writing low-level OpenFlow rules, but rather is expressing what it wants to have happen in terms of higher-level behaviors without regard to such things as whether or not the rules are being installed in a consistent, and correct fashion. Various people may write these applications including network operators, service providers, researchers, and really anyone who wants to develop capabilities on top of OpenFlow. The benefits of such a northbound API are vendor independence, as well as the ability to quickly modify or customize control through various popular programming languages. The idea is that these applications might be written in high-level programming languages, such as Python, and wouldn't actually have to perform low-level switch modifications, but rather could express policies in terms of much higher-level abstractions. Examples of such applications include the implementation of a large virtual switch abstraction, security applications, and services that may need to integrate traffic processing with middle boxes. This programmatic interface is called the northbound API and currently there's no standard for the northbound API, as there is for the southbound API in OpenFlow. But we'll look at various APIs in programming languages that each compile to OpenFlow rules that are installed on switches across the network.

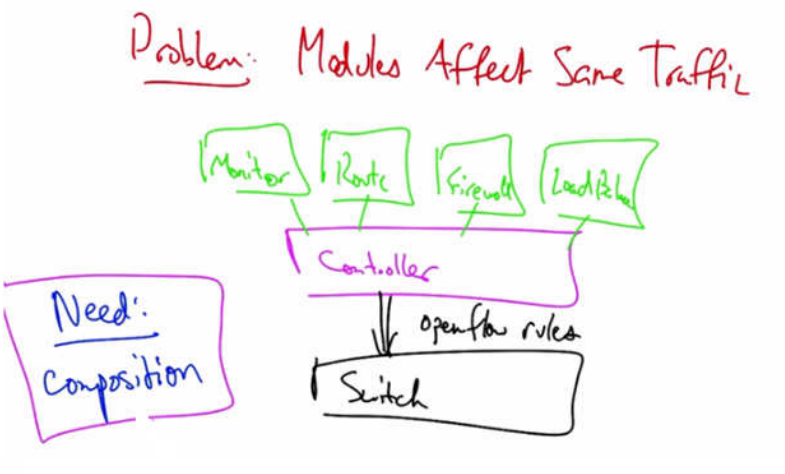
Frenetic Language



One example of a programming language that sits on top of such a north-bound API is Frenetic, which is an SQL-like query language. For example, Frenetic would allow a programmer to count the number of bytes, grouped by destination Mac address, and report the updates to these counters every 60 seconds. The "group by" statement allows a grouping of counts by the destination mac address. "Where" allows restrictions to only count traffic coming from a web server coming in on a particular port, and" every" specifies that the results of this query should

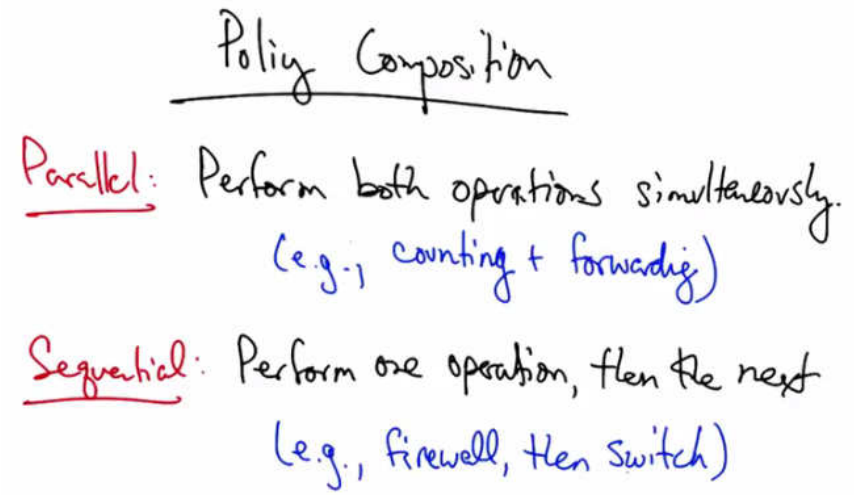
only be returned every 60 seconds. More information about Frenetic is available at frenetic-lang.org. And in the course, we'll actually going to use a language called Pyretic that is based on the same underlying theory as Frenetic, except that it's embedded in Python.

Overlapping Network Policies

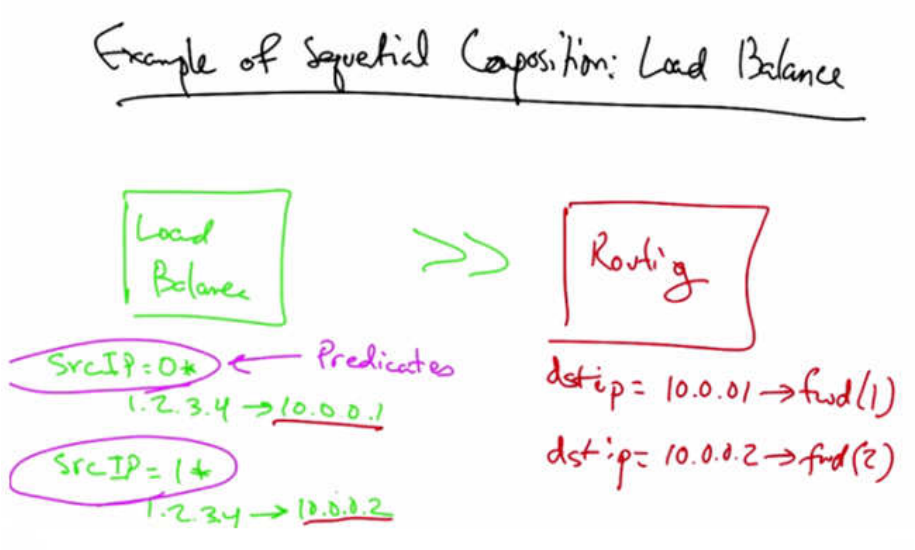


One issue with programming at this higher level of abstraction is that an operator might write multiple modules, each of which effects the same traffic. For example, an operator might write an application that monitors traffic. Another one that specifies how routing should take place, another that involves the specification of firewall rules And yet another that balances traffic load across the links in the network. Ultimately, all of these applications, or modules, must be combined into a single set of OpenFlow rules that together achieve the network operator's overall goal. For this, we need composition operators, or ways that specify how these individual modules should be combined or composed into a single coherent application. Let's now talk about two different ways to compose policies.

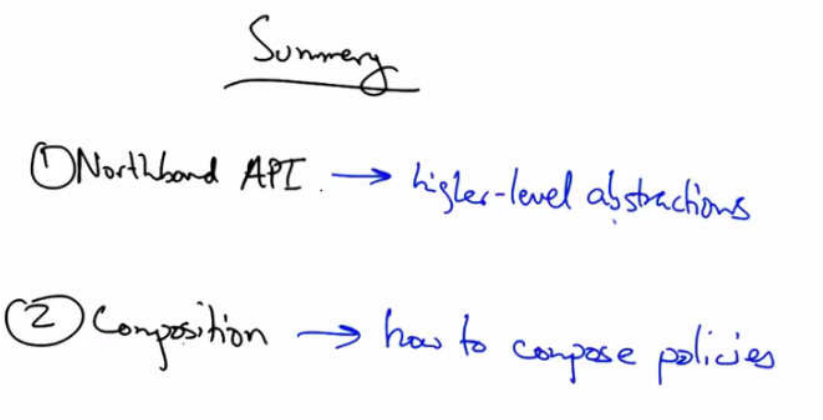
Composing Network Policies with Pyretic



One way of composing policies is to perform both operations simultaneously. For example, one might want to forward traffic but also count how much traffic is being forwarded. Both of those operations can be performed in parallel. Another way of composing policies is in sequence. Sequential composition performs one operation then the next. For example, we might want to implement a firewall, and whatever traffic makes it though the firewall might then be subjected to the switching policy.

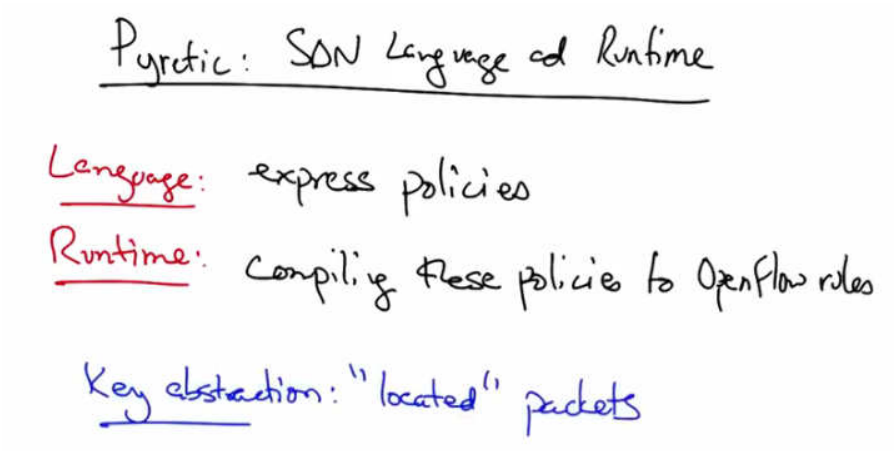


One example of sequential composition, might be a load balancer. In this example, a policy might take some traffic coming from half of the source IP addresses and rewrite that to one server replica, and take the other half of the traffic and rewrite it to the other replica. After the load balancer rewrites the destination IP address, we need a routing module to forward the traffic out the appropriate port on the switch. In this case, we've used sequential composition to first apply a load balance policy that rewrites the destination IP address based on the source IP address where the traffic is coming from and sequentially apply a routing policy that forwards the traffic out the appropriate port depending on the resulting destination IP address after that rewrite has taken place. Notice that we can use predicates to specify which traffic traverses which modules. Those predicates can apply specific actions based on things like the input port and the packet header fields. The ability to compose policies in this fashion allows each module to partially specify functionality without having to write the policy for the entire network. This leaves some flexibility so that one module can implement a small bit of the network function, leaving some functions for other modules. This also allows for module re-use, since a module need not be tied to a particular network setting. For example, in this particular example where we've applied that load balancer followed by routing, the load balancer spreads traffic across the replicas without regard to the underlying network paths that traffic takes once those destination IP addresses are rewritten.

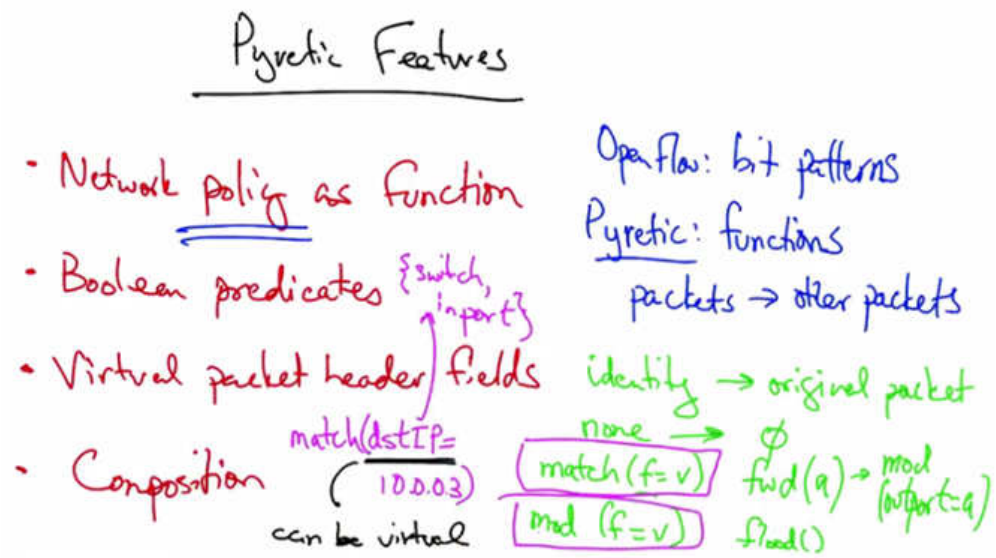


In summary, we've covered two concepts. One is the notion of a Northbound API, which sits on top of an SDN controller and provides and exposes higher level abstractions that allows the operator or programmer to write policies without regard to how OpenFlow rules eventually get installed. We've also talked about two different composition operators. Parallel composition and sequential composition, which specify how individual simpler policies can be composed to implement more complex network applications, thus allowing different SDN control programs to independently perform tasks on the same traffic.

Pyretic Language



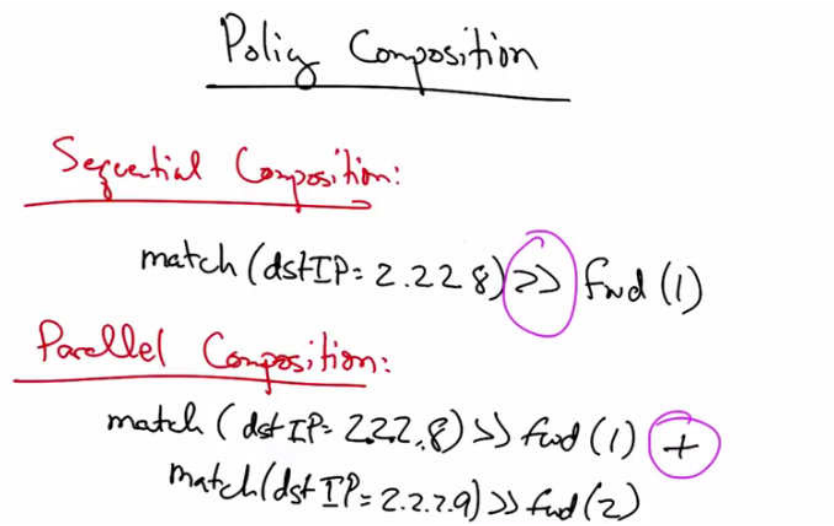
In this lesson we will look at Pyretic which is an SDN language and run time that implements some of the composition operators that we discussed in the last lesson. The language is a way of expressing these high level policies, and the run time provides the function of compiling these policies to the OpenFlow rules that eventually are installed on the switches. One key abstraction in Pyretic is the notion of located packets, the idea that we can apply a policy based on a packet and its location in the network, such as the switch at which that packet is located or the port on which that packet arrives.



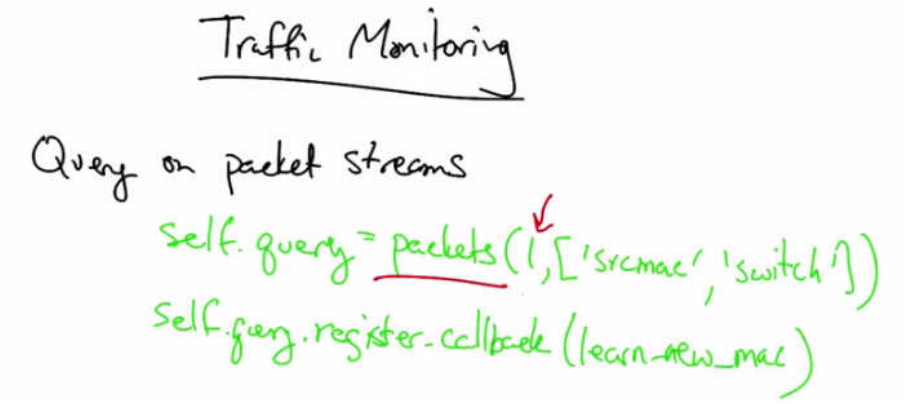
Pyretic offers several features. The first is the ability to take as input a packet and then return packets at different locations in the network. This effectively allows the implementation of network policy as a function that simply takes packets and returns other packets at different locations. The second feature of Pyretic is the notion of Boolean predicates. Unlike OpenFlow rules which do not permit the expression of simple conjunctions such as AND and OR, or negations like NOT, Pyretic allows the expressions of policies in terms of these predicates. Pyretic also provides the notion of virtual package header fields. Which allows the programmer to refer to packet locations and also to tag packets so that specific functions can be applied at different portions of the program. Pyretic also provides composition operators, such as parallel and sequential composition, which we discussed in the last lesson. The notion of network policy as a function contrasts with the OpenFlow style of programming. In OpenFlow, policies are simply bit patterns, in other words, match statements for which matching packets are subject to a particular action. These types of policies can be particularly difficult to reason about. In contrast, in Pyretic, policies are functions that map packets to other packets. Some example functions in Pyretic include the identify function, which returns the original packet; none or drop, which returns the empty set; match, which returns the identity if the field f matches the value v and returns none or drop otherwise; mod, which returns the same packet with the field f set to v; forward, which is simply syntactic sugar on mod to say that the output port field in the packet should be modified to the parameter specified; and flood, which returns one packet for each port on the network spanning tree. In OpenFlow, packets either match on a rule or they simply fall through to the next rule. So, OR, NOT, etc, can be tough to reason about. In contrast, Pyretic's match function outputs either the packet or nothing, depending on whether the predicate is satisfied. For example, we could apply a match statement that says match destination IP equals 10.0.0.3. and this function would take packets as input and only return packets that satisfy this particular predicate. In addition to the standard packet header fields, Pyretic offers the notion of virtual packet header fields, which is a unified way of representing packet metadata. In Pyretic, the packet is nothing more than a dictionary that maps a field name such as the destination IP address to a value. Now, these field names could correspond to fields in an actual packet header. But they can also be virtual. For example, we could provide a match statement based on a switch,

indicating that we only want to return packets that are located at a particular switch or on the input port, indicating that we only want to see packets whose attributes match a particular input port. The match function matches on this packet meta-data and the mod function can modify this meta-data.

Composing Network Policies with Pyretic



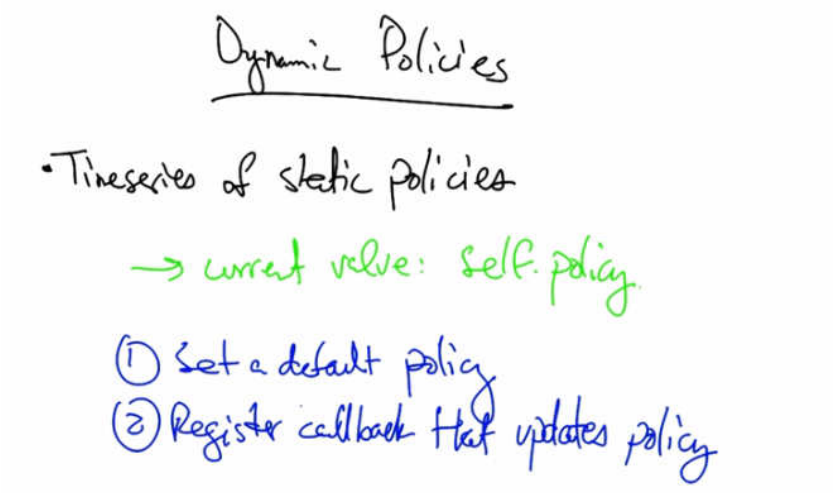
Pyretic enables the notion of both sequential and parallel composition as we've discussed in previous lessons. For example, we could match all packets for a particular destination IP address and send them or forward them out a particular output port. The double greater than sign shown here is the way of expressing sequential composition in Pyretic. Parallel composition allows two policies to be applied in parallel. In this example, we match on a particular destination IP address and subsequently forward the traffic out Output Port one. In Parallel, we apply a different set of policies that match on a different source IP address and output the packets on a different output port. In Pyretic, the plus operator performs parallel composition of policies.



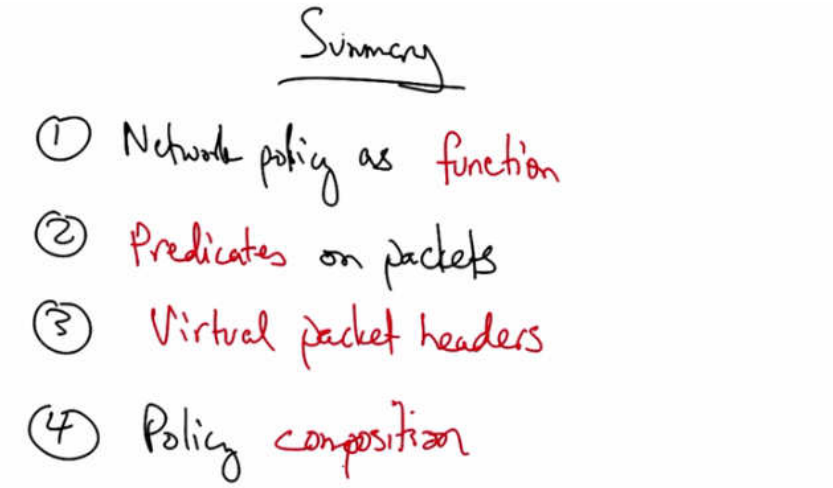
Pyretic allows an operator to construct queries which allow the program to see packet streams. For example, the packets query allows the operator to see packets arriving at a particular switch with a particular source MAC address. The one parameter here indicates that we only want to see

the first packet that arrives with a unique source MAC address and switch. We can then register callbacks for these packet streams that are invoked to handle each new packet that arrives for that query.

Dynamic Policies in Pyretic

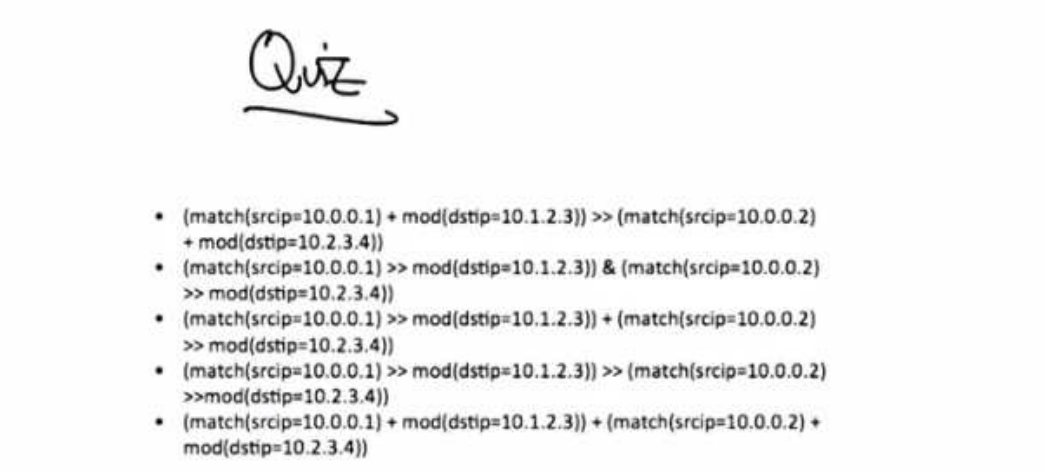


Dynamic policies are policies whose forwarding behavior can change. They are represented as a time series of static policies. The current value of the policy at any time is self dot policy. A common programming idiom in Pyretic is to set a default policy and then register a call back that updates that policy. In the assignment, you will create a similar topology that you created in the pox assignment, but we will now use pyretic to implement a simple switch and firewall. In pyretic every first packet with a new source MAC address at the switch is read by a query. The policy is updated with a new predicate every time a new mapping between a MAC address and an output port is learnt. In the assignment, you also create a dynamic firewall policy, register a callback to check the rules, and sequentially compose your firewall policy with a learning switch, thus provided as part of the pyritic distribution.



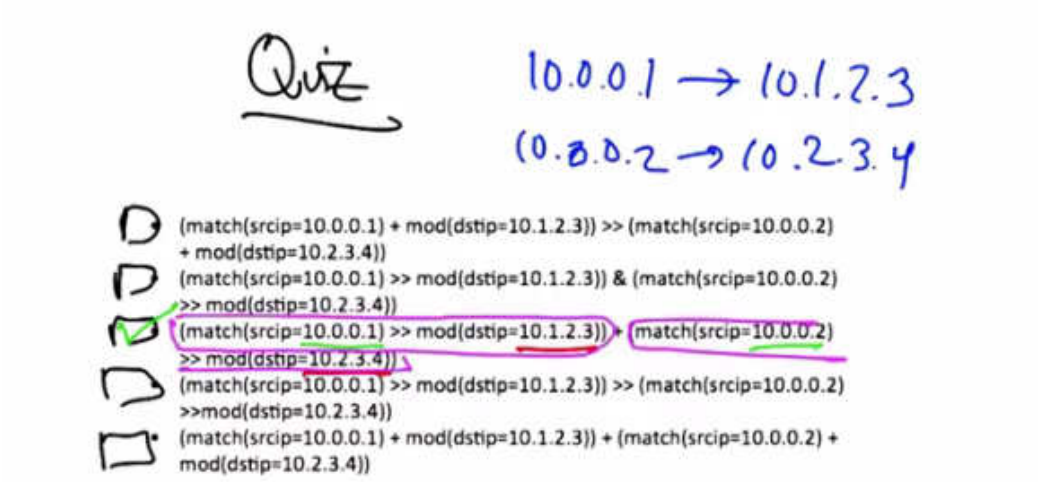
In summary, pyretic allows operators to write complex network policies as functions. It allows an operator to express predicates on packets including things such as AND, OR, and NOT. It provides the capability to specify and modify virtual packet headers as packet metadata, and it provides ways to compose complex network policies from simpler independent ones.

Pyretic Policy Quiz



As a quiz, which of the following is the appropriate pyretic rule for sending traffic from source IP address 10.0.0.1 to destination at IP address 10.1.2.3, and traffic from source IP address 10.0.0.2 to destination IP address 10.2.3.4

Pyretic Policy Quiz



The following policy matches the appropriate source IP addresses and forwards to the corresponding output destination IP address. Each of these matching and forwarding operations can happen in parallel.

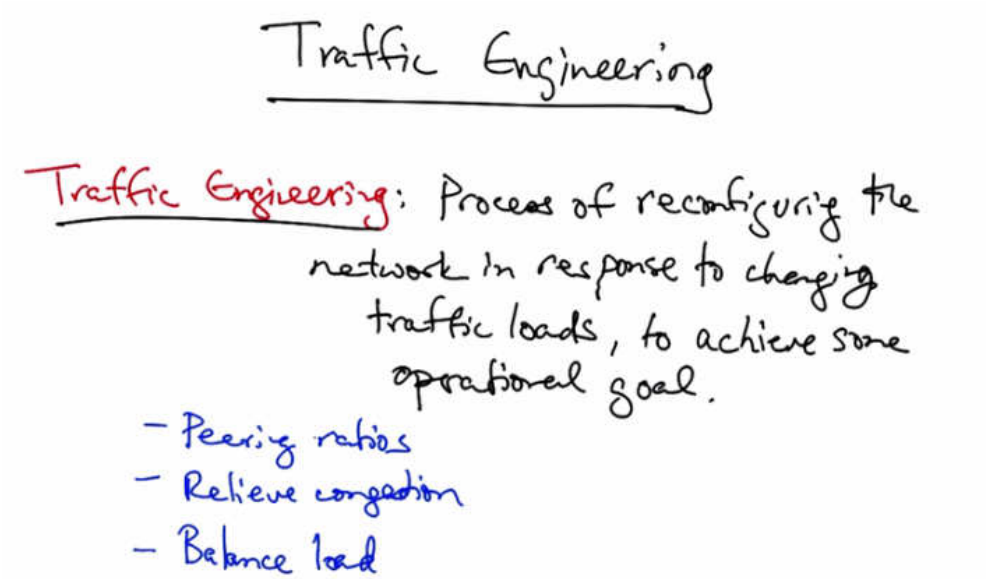
CS 6250: Computer Networking

Lecture 10: Traffic Engineering

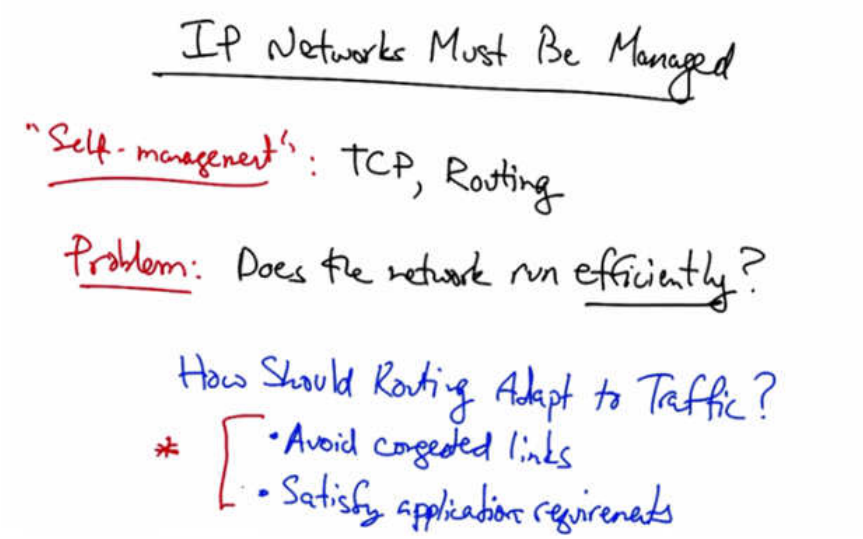
Traffic Engineering Intro

Whew. We just completed our exploration of software defined networking, and now we're jumping into traffic engineering. Traffic engineering is a fancy way of describing how network operators deal with large amounts of data flowing through their networks. Companies like Google are doing exciting work in this area, so be sure to pay attention to the instructor note section, where we'll highlight exciting research papers.

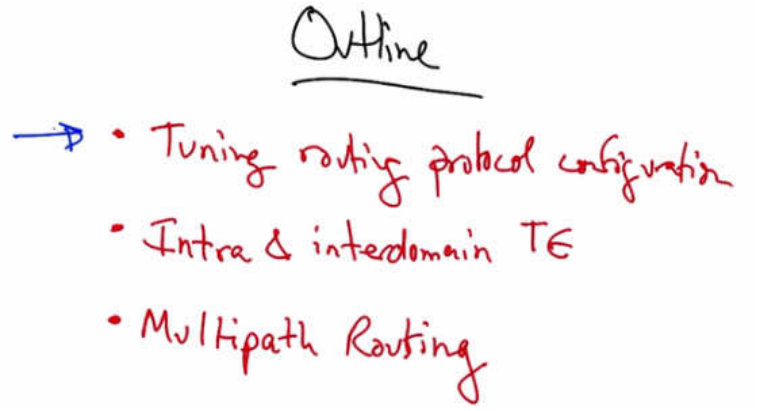
Traffic Engineering Overview



This lesson covers traffic engineering. Traffic engineering is the process of reconfiguring the network in response to changing traffic loads to achieve some operational goal. A network operator might want to reconfigure the network in response to changing loads to, for example, maintain traffic ratios in a peering relationship, or to relieve congestion on certain links in the network, or to balance load more evenly across the available links in the network. In this lesson we will explore different ways of performing traffic engineering. We will start by looking at conventional approaches to traffic engineering and we'll explore how software-defined networking is used to make the process of traffic engineering easier in both data center networks and transit networks.

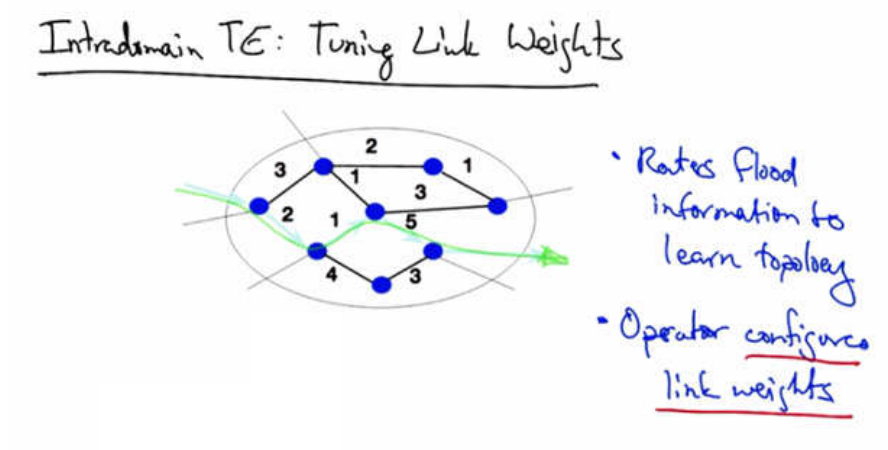


IP networks must be managed. Now, in some sense, IP networks manage themselves. There are several examples of protocols on the internet that in some sense manage themselves. TCP senders send less traffic during congestion and routing protocols will adapt to topology changes. The problem is that even though these protocols are designed to adapt to various changes, the network may not run efficiently. There may be congested links when idle paths exist or there might be a high-delay path that some traffic is taking when a low-delay path, in fact, exists. So a key question that traffic engineering tries to address is how routing should adapt to traffic. In particular, traffic engineering attempts to avoid congested links and satisfy certain application requirements such as delay. These are, in some sense, the essential questions that traffic engineering attempts to answer.

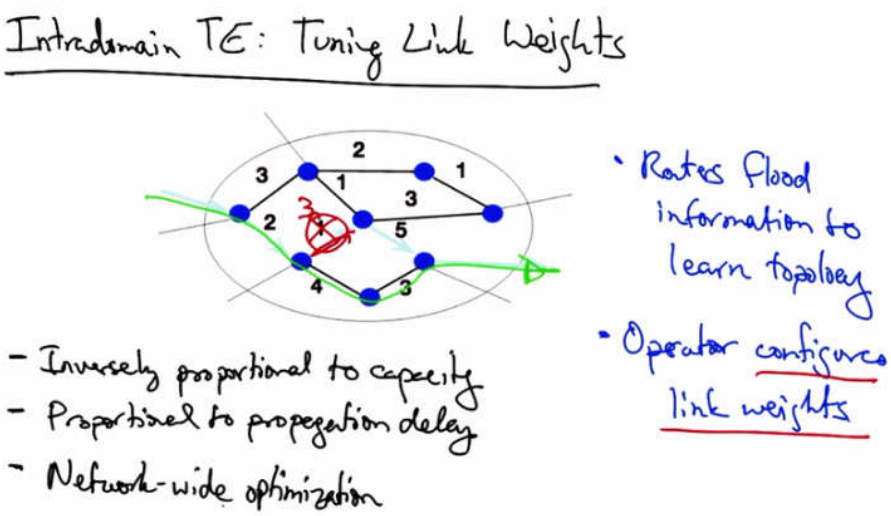


In the rest of this lesson we will look at how network operators can tune a routing protocol configuration to affect how network traffic traverses the links in the network. And in particular we will look at both intradomain traffic engineering, that is how to reconfigure the protocols within a single autonomous system to adjust traffic flows, as well as interdomain traffic engineering, or how to adjust how traffic flows between autonomous systems. We will also look at multi-path routing and how it is used to achieve various traffic engineering goals. Let's start by looking at how network operators can tune static link weights in a routing protocol like OSPF or ISIS, to affect how traffic flows through the network.

Intradomain Traffic Engineering

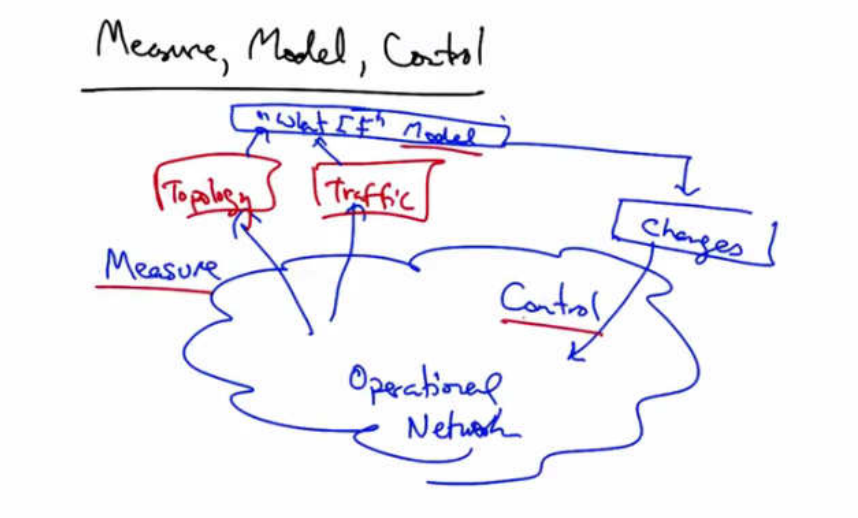


Let's assume that we have a single autonomous system with static link weights as shown. In such a setup, routers will flood information to one another to learn the network topology, including the link weights on links connecting individual routers. An operator can affect how traffic flows through the network by configuring the link weights. By changing the link weight configuration, an operator can affect the shortest path between two points in this graph, thus affecting the way that traffic flows. In the link weight settings shown here, traffic would flow along the green path.

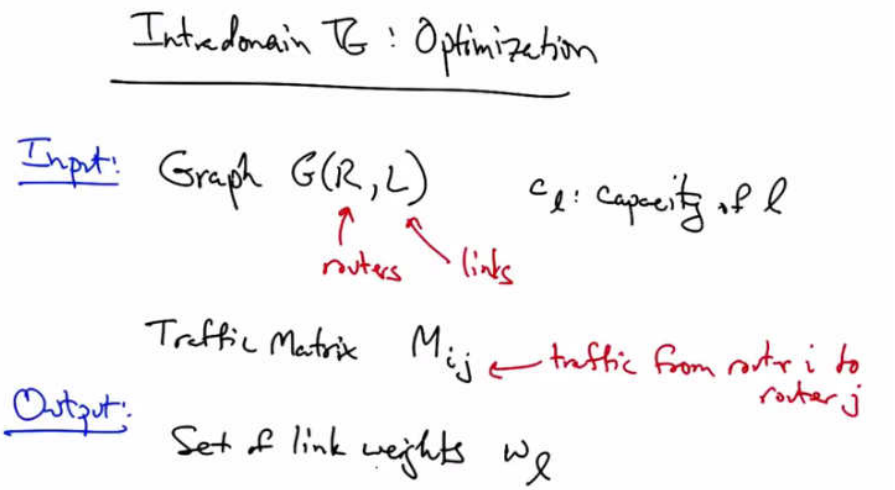


Suppose that the operator would like to shift traffic off of a congested link in the middle of the network such as this one by changing the link weight from one to three. The shortest path between this node and this node, now takes an alternate route. So we can see from this simple example, that by adjusting the link weights in an intra-domain topology, the operator can affect how traffic flows between different points in the network, thus affecting the load on the network links. In practice, network operators set these link weights in a variety of ways. One could set the link weights inversely proportional to capacity, proportional to propagation delay, or the operator might perform some network wide optimization based on traffic.

Measuring, Modeling and Controlling Traffic



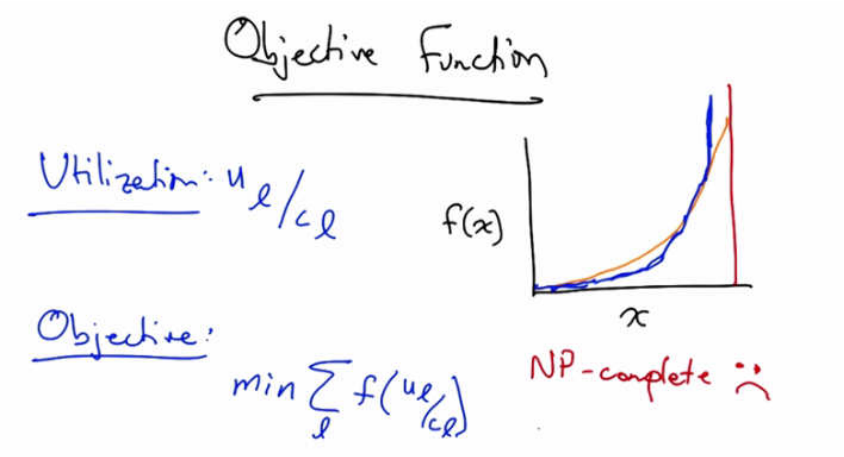
Traffic engineering has three steps: measuring the network to figure out the current traffic loads, forming a model of how configuration affects the underlying paths in the network, and then ultimately reconfiguring the network to exert control over how the traffic flows through the network. An operator might measure the topology and traffic, feed the topology and traffic to a what-if model that predicts what will happen under various types of configuration changes, decide which changes to affect on the network, and then, ultimately, control the network behavior by readjusting link weights. So, in summary, we have measurement, modeling and control. Each of these three components requires a significant amount of heavy lifting to make both practical and accurate in practice.



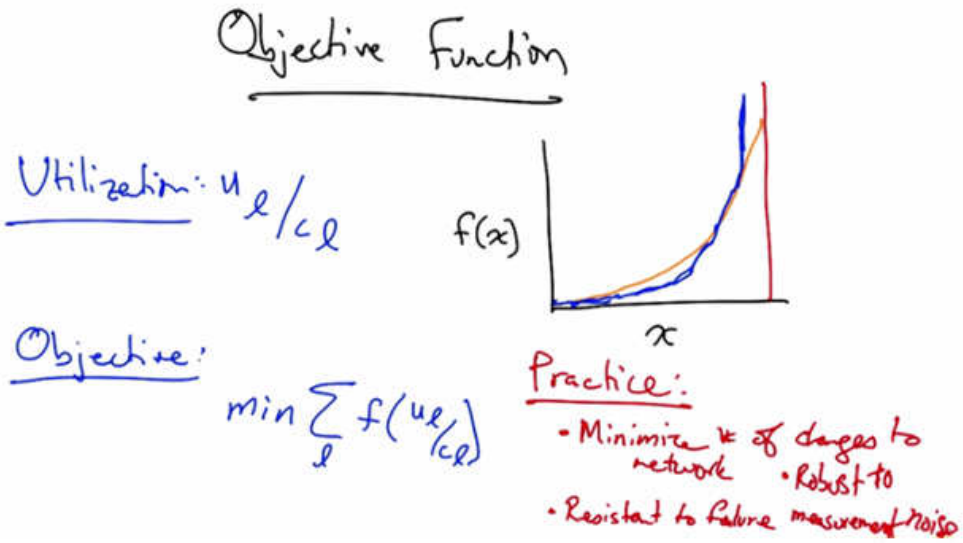
Intradomain traffic engineering attempts to solve an optimization problem where the input is the graph G, where R is the set of routers, and L is the set of unidirectional links. Each link L also has a fixed capacity. Another input is the traffic matrix ,or offered traffic load, where M_ij represents the traffic load from router i to router j. The output of the optimization problem is a

set of link weights, where w_l is the weight on any unidirectional link l in the network topology. Ultimately, the setting of these link weights should result in a fraction of the traffic from i to j, traversing each link l, such that those fractions satisfy the network-wide objective function. Defining an objective function is tricky. We could talk about, for example, minimizing the maximum congested link in the network, evenly splitting traffic loads across links, and so forth.

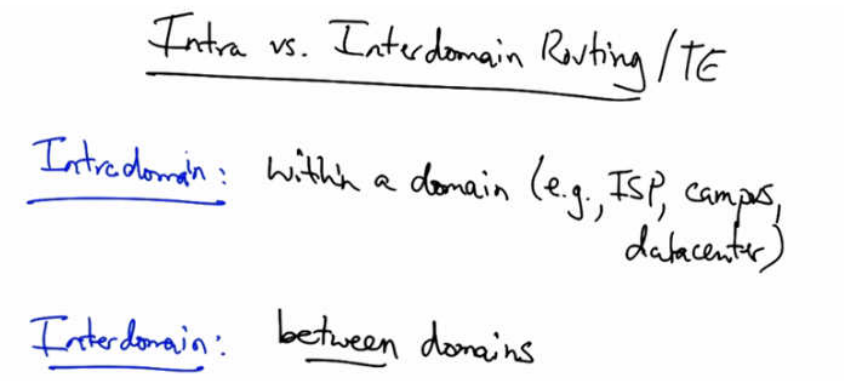
Link Utilization Function



What we'd like to represent is that the cost of congestion increases in a quadratic manner as the loads on the links continue to increase, ultimately becoming increasingly expensive as link utilization approaches one. Solving the optimization problem, however, is much easier if we use a piecewise linear cost function. We can define utilization as the amount of traffic on the link divided by the capacity and our objective might be to minimize the sum of this piecewise linear cost function over all the links in the network. Unfortunately, solving this optimization is still NP-complete, which means that there is no efficient algorithm to find the optimal setting of link weights, even for simple objective functions.

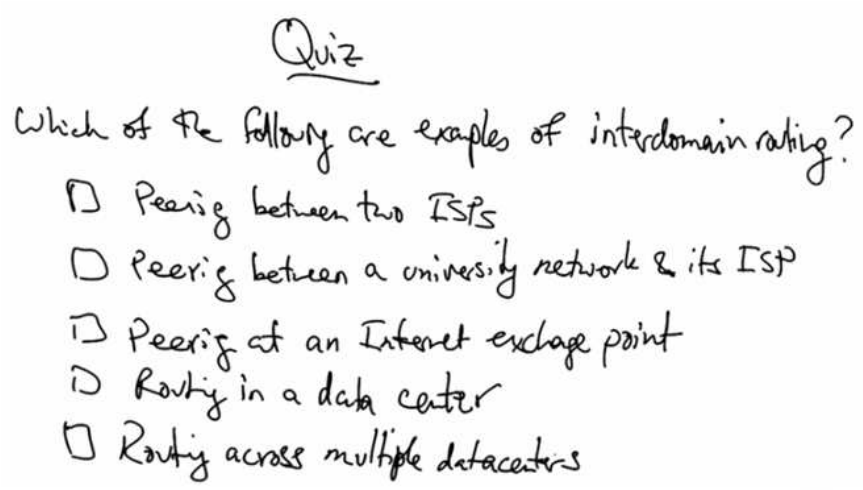


The implications of this are that we have to resort to searching through a large set of combinations of link weight settings to ultimately find a good setting. So clearly searching through all the link weights is suboptimal, but the graphs turn out to be small enough in practice such that this approach is still reasonably effective. In practice, we also have other operational realities to worry about. For example, minimizing the number of changes to the network. Often just changing one or two link weights is enough to achieve a good traffic load balance solution. Whatever solution we come up with must be resistant to failure and it should be robust to measurement noise. We also want to limit the frequency of changes that we make to the network. This completes our overview of intradomain routing.



And now we will take a look at interdomain routing. Intradomain routing and traffic engineering concerns traffic flow within a single domain, such as an ISP, a campus network, or a data center. In contrast, interdomain routing and interdomain traffic engineering concerns routing that occurs between domains, something that we looked at before in the context of the Border Gateway Protocol.

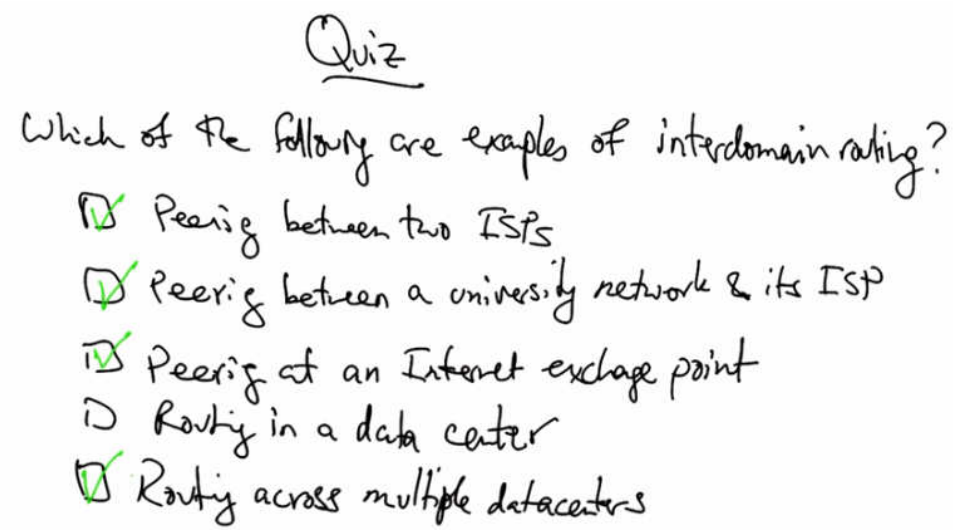
Interdomain Routing Quiz



Before we proceed with our discussion of interdomain traffic engineering, let's take a brief quiz reminding ourselves about the differences between intradomain routing and interdomain routing.

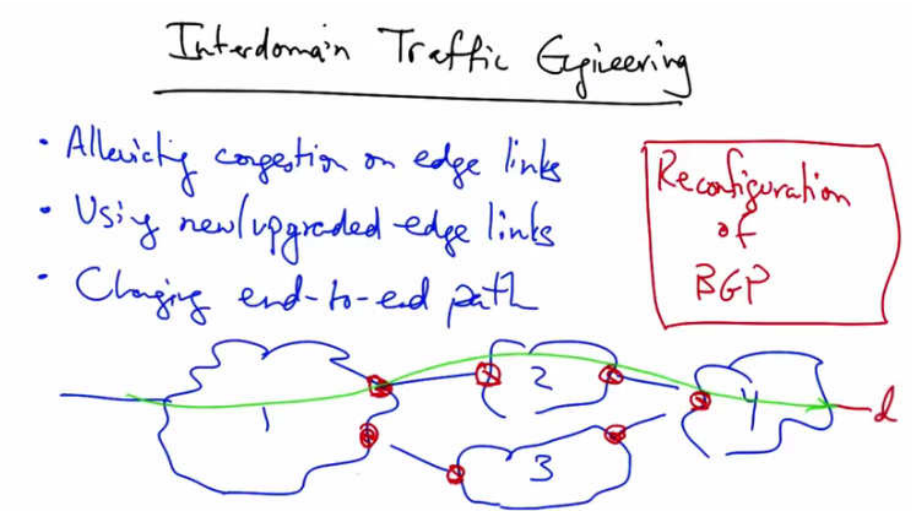
Which of the following are examples of interdomain routing? Peering between two internet service providers? Peering between a university network, and its ISP? Peering at an internet exchange point? Routing inside a data center? Or routing across multiple data centers? Please, again, check all that apply.

Interdomain Routing Quiz Answer

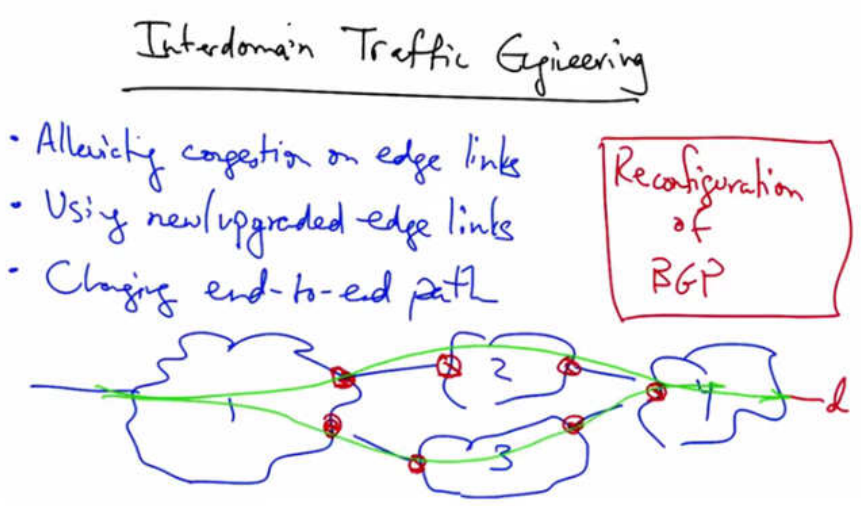


Peering between two ISPs involves interdomain routing, as does peering between a university network and its ISP, or any peering at an Internet exchange point, for that matter. Routing between multiple data centers typically involves routing in a wide area, and hence may involve interdomain routing. Routing within a single data center concerns routing in a single domain and hence does not concern interdomain routing.

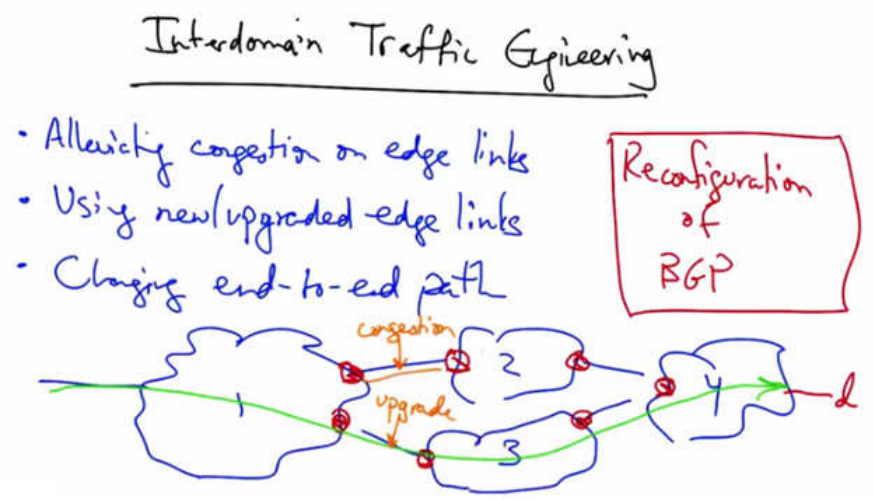
BGP in Interdomain Traffic Engineering



Inter-domain routing concerns routing between domains or autonomous systems. It involves the reconfiguration of the border gateway protocol, policies or configurations that are running on individual routers in the network. Changing BGP policies at these edge routers can cause routers inside an autonomous system to direct traffic to or away from certain edge links. We can also change the set of egress links for a particular destination. For example, an operator of autonomous system one might observe traffic to destination D traversing the green path.

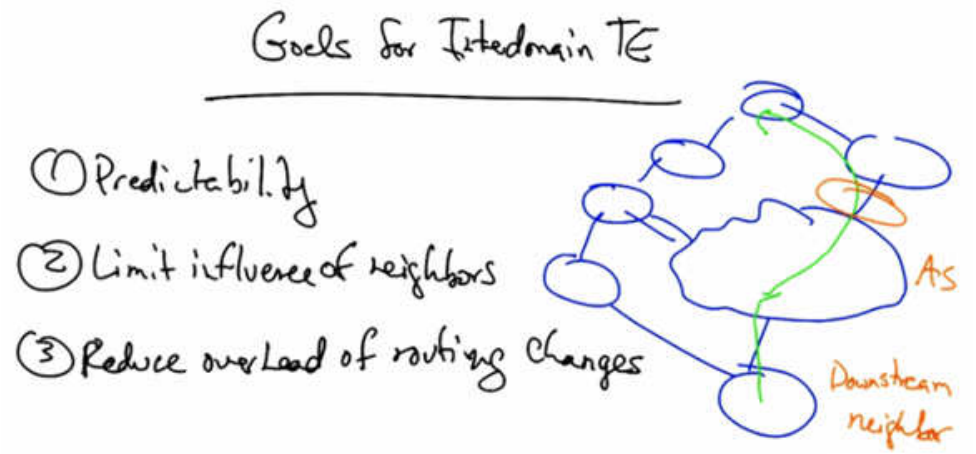


But by adjusting BGP policies, the operator might balance load across these two edge links, or shift all of the traffic for that destination to the lower path.

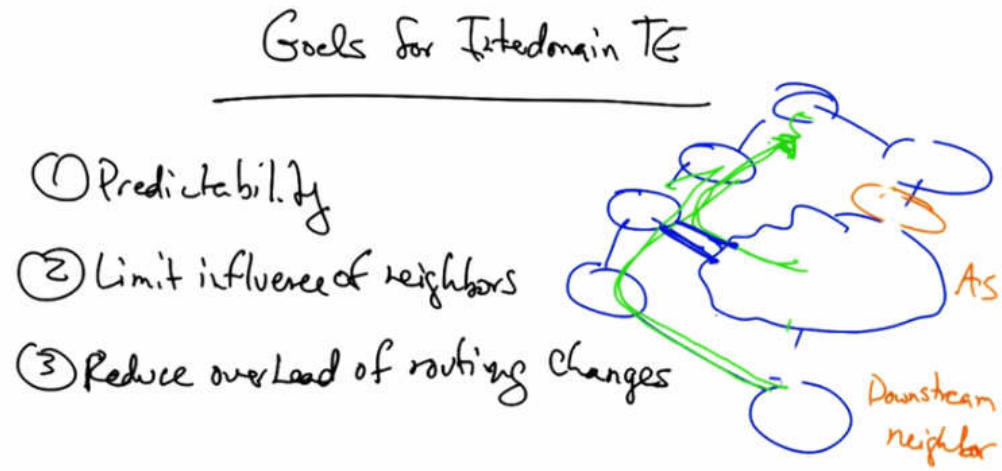


An operator might wish to use inter domain traffic engineering if an edge link is congested, if a link is upgraded, or if there's some violation of a peering agreement. For example, AS1 and AS2 have an agreement that they only send a certain amount of traffic load over that link in a particular time window. If the load exceeds that amount, an operator would need to use BGP to shift traffic from one egress link to another.

Interdomain Traffic Engineering Goals

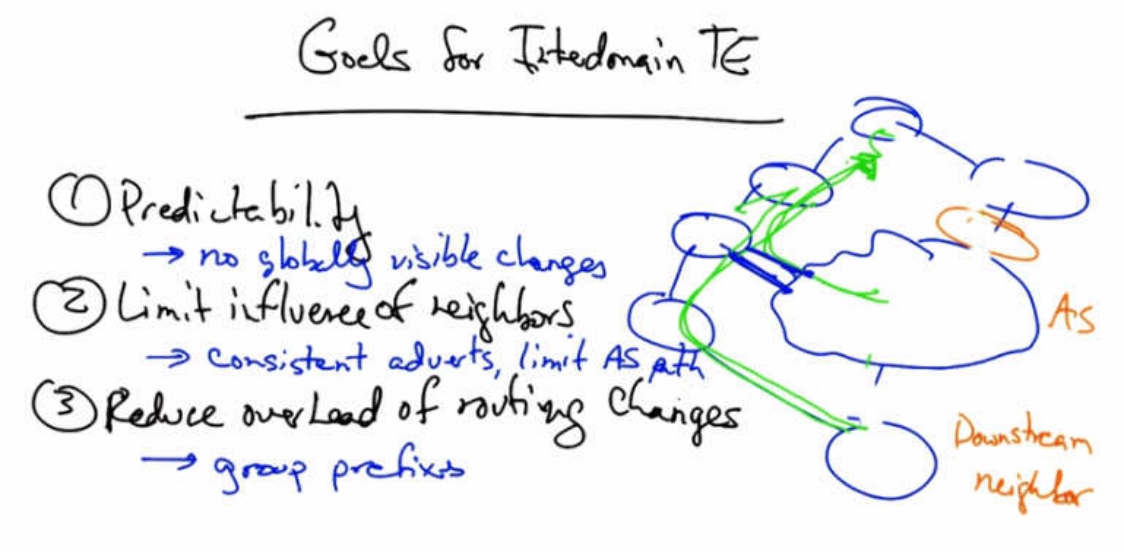


Effective Inter-domain Traffic Engineering has three goals. One is predictability. In other words, it should be possible to predict how traffic flows will change in response to changes in the network configuration. Another goal is to, limit the influence of neighboring domains. In particular, we'd like to use BGP policies and changes to those policies that limit how neighboring ASes might change their behavior in response to changes to the PGP configuration that we make in our own network. And finally, we'd like to reduce the overhead of routing changes by achieving our traffic engineering goals with changes to as few IP prefixes as possible. To understand the factors that confound predictability Let's look at how the inter-domain routing choices of a particular autonomous system can wreak havoc on predictability.



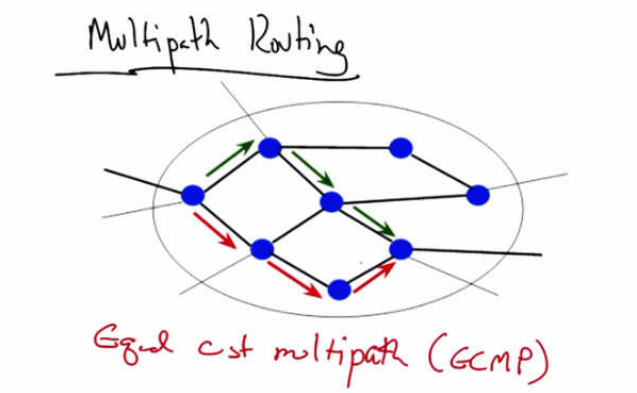
Let's suppose that a downstream neighbor is trying to reach the autonomous system at the top of this figure. The AS here might wish to relieve congestion on a particular peering link. To do so, this AS might now send traffic to that destination out a different set of autonomous systems. But once this AS makes that change, note that it's choosing a longer AS path. Now taking a path of three hops rather than two. In response, the downstream neighbor might decide not to send its traffic for that destination through this autonomous system at all, thus affecting the traffic matrix

that this AS sees. So, all the work that went into optimizing the traffic load balance for this AS is for naught, because the change that it made effectively changed the offered traffic loads and hence the traffic matrix. One way to avoid this type of problem and achieve predictable traffic flow changes is to avoid making changes like this that are globally visible. In particular, note that this change caused a change in the AS path link of the advertisement to this particular destination from two to three. Thus, other neighbors, such as the downstream neighbor here, might decide to use an alternate path as a result of that globally visible routing change. By avoiding these types of globally visible changes, we can achieve predictability. Another way to achieve effective interdomain traffic engineering is to limit the influence of neighbors. For example, an autonomous system might try to make a path look longer with AS path prepending. If we consider treating paths that have almost the same AS path length as a common group, we can achieve additional flexibility. Additionally, if we enforce a constraint that our neighbors should advertise consistent BGP route advertisements over multiple peering links (should multiple appearing links exist), ghat gives us additional flexibility to send traffic over different egress points to the same autonomous system, enforcing egress points to the same autonomous system. Enforcing consistent advertisements turns out to be difficult in practice, but it is doable. To reduce the overhead of routing changes, we can group related prefixes. Rather than exploring all combinations of prefixes to move a particular volume of traffic, we can identify routing choices that group routes that have the same AS paths, and we can move groups of prefixes according to these groups of prefixes that share an AS path. This allows us to move groups of prefixes by making tweaks to local preference on regular expressions on AS path. We can also focus on the small fraction of prefixes that carry the majority of traffic. Ten percent of origin AS is responsible for about 82 percent of outbound traffic. Therefore we can achieve significant gains in rebalancing traffic in the network by focusing on the heavy hitters.

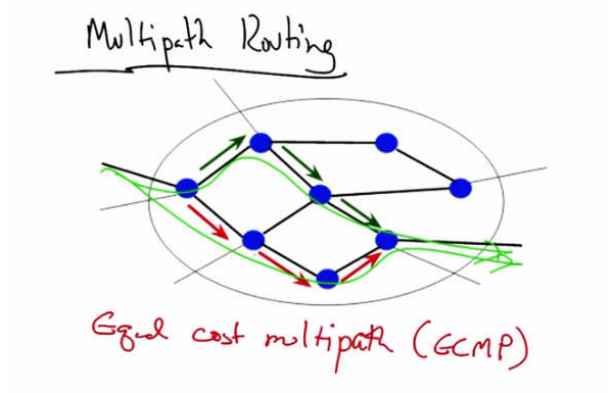


In summary, to achieve predictability, we effect changes that are not globally visible. To limit the influence of neighbors, we enforce consistent advertisements and limit the influence of AS pass length. And to reduce the overhead of routing changes, we group pre fixed according to those that have common AS paths and move traffic in terms of groups of prefixes.

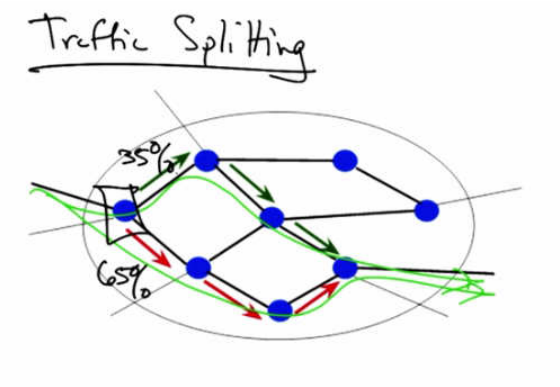
Multipath Routing



Another way to perform traffic engineering is with Multipath Routing, where an operator can establish multiple paths in advance. This approach applies both to intradomain routing and interdomain routing. The way this is done in intradomain routing is to set link weights such that multiple paths of equal cost exist between two nodes in the graph. This approach is called equal cost multipath.

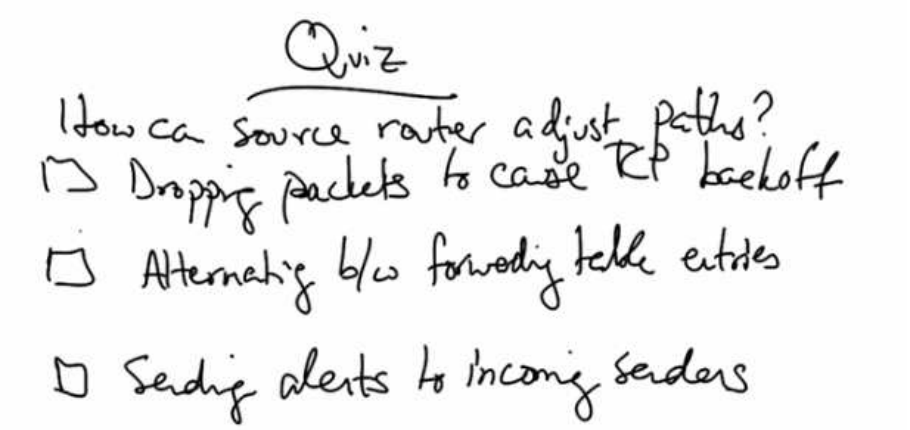


Thus traffic will be split across paths that have equal costs through the network. A source router might also be able to change the fraction of traffic that's sent along each one of these paths. Sending, for example, 35% along the top path and 65% along the bottom path.



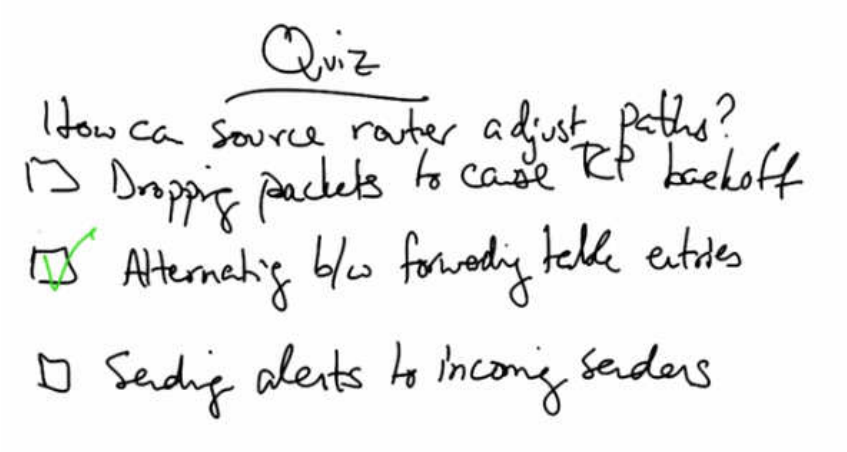
It might even be able to do this based on the level of congestion that's observed along these paths. The way that the router would do this is by having multiple forwarding table entries with different next hops for outgoing packets to the same destination.

Source Router Path Quiz



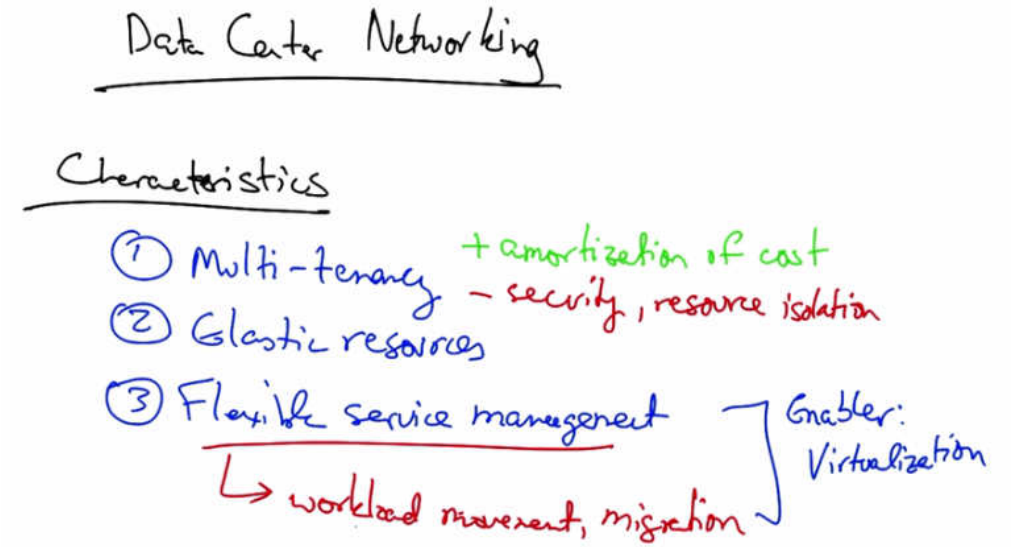
So quickly, how can a source router adjust paths to a destination when there are multiple paths to the destination? Dropping packets to cause TCB backoff, alternating between multiple forwarding table entries to the same destination, or sending alerts to incoming senders whenever a route changes?

Source Router Path Quiz Answer



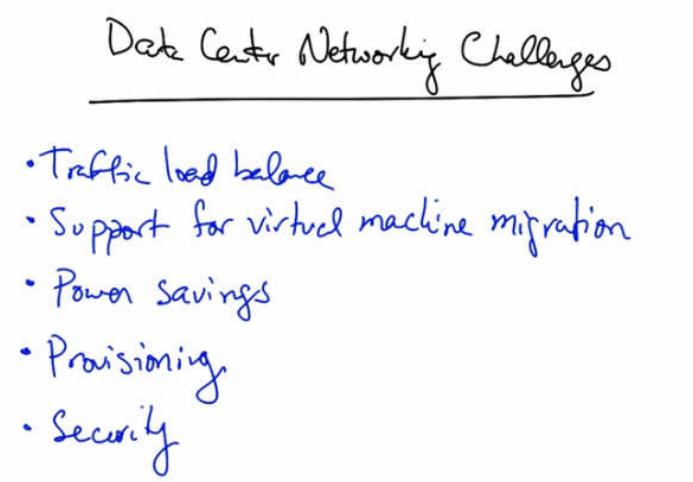
A source router can adjust traffic over multiple paths by having multiple forwarding table entries for the same destination and splitting traffic flows across the multiple next hops depending on, for example, the hash of the IP packet header.

Data Center Networking

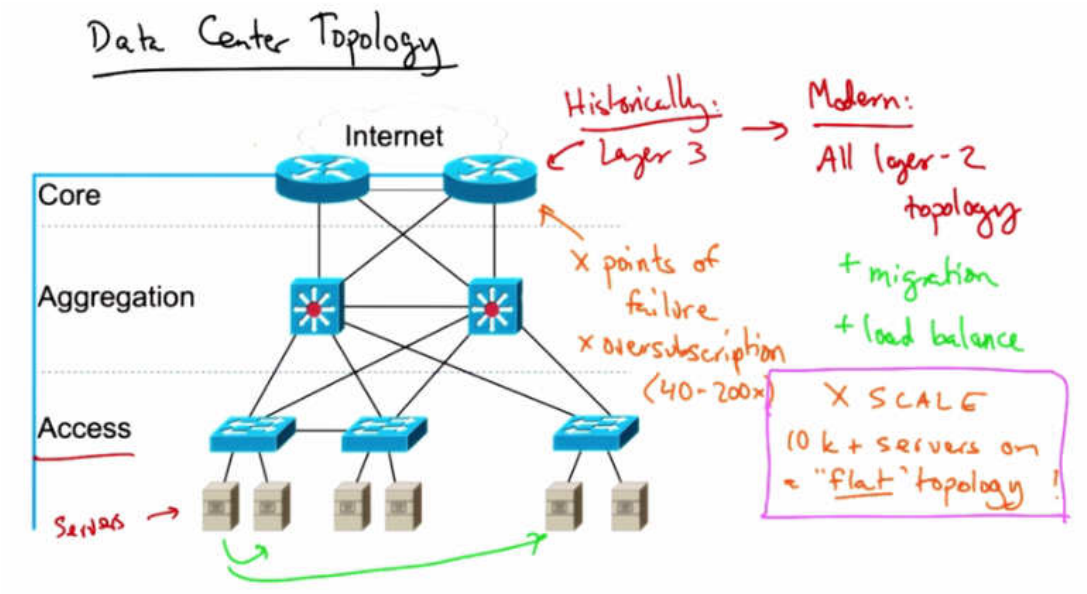


Let's now talk about data center networking and how network operators perform traffic engineering inside a data center. First of all, what characterizes a data center? Data center network has 3 important characteristics. One is multi-tenancy. Multi-tenancy allows a data center provider to advertise the cost of shared infrastructure, but because there are multiple independent users, the infrastructure must also provide some level of security and resource isolation. Data Center network resources are also elastic, meaning that as demand for service fluctuates, the operator can expand and contract these resources. It also can be pay per use, meaning that as the need to use more resources arises or disappears, a service provider can adjust how much resources are devoted to the particular service running in the data center. Another characteristic of data center networking is flexible service management, or the ability to move work, or workloads, to other locations inside the data center. For example, as load changes for a particular service, an operator may need to provision additional virtual machines, or servers to handle the load for that service or potentially move it to a completely different set of servers inside the infrastructure. This workload movement and virtual machine migration essentially creates the need for traffic engineering solutions inside a data center. A key enabling technology in data center networking is the ability to virtualize servers. This makes it possible to quickly provision, move, and migrate servers and services in response to fluctuations in workload. But while provisioning servers and moving them is relatively easy, we must also develop traffic engineering solutions that allow the network to reconfigure in response to changing workloads and migrating services.

Data Center Networking Challenges



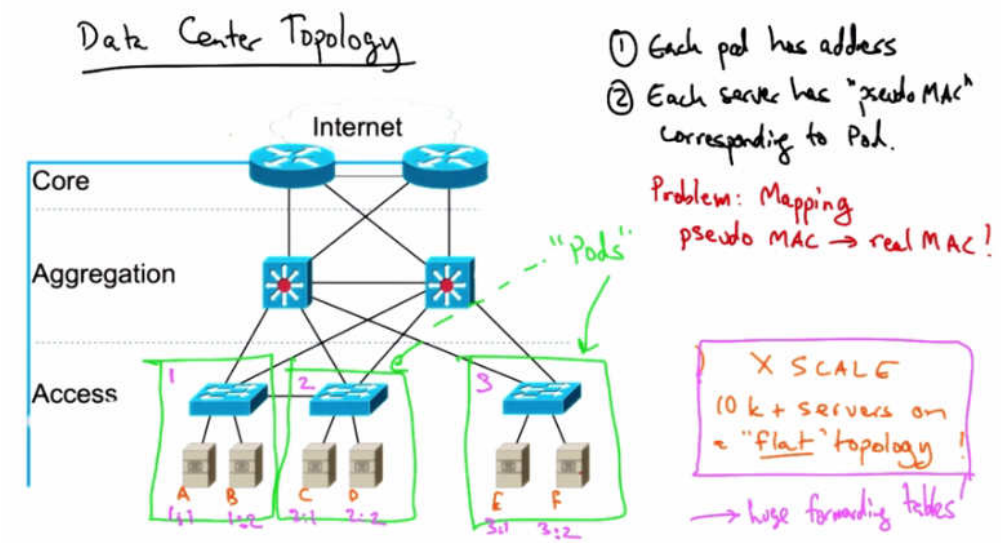
Some of the challenges for data center networking include traffic load balance, support for migrating virtual machines in response to changing demands, adjusting server and traffic placement to save power, provisioning the network when demands fluctuate, and providing various security guarantees, particularly in scenarios that involve multiple tenants. To understand these challenges in a bit more detail, let's take a look at a typical data center topology.



A topology typically has three layers: an access layer, which connects the servers themselves. An aggregation layer which connects the access layer, and then the core. Historically, the core of the network has been connected with layer three, but increasingly, modern data centers are connected with an entire layer-two topology. A layer-two topology makes it easier to perform migration of services from one part of the topology to another, since these services can stay on the same layer-two network and hence would not need new IP addresses when they moved. It also becomes easier to load balance traffic. On the other hand, a monolithic layer-two topology

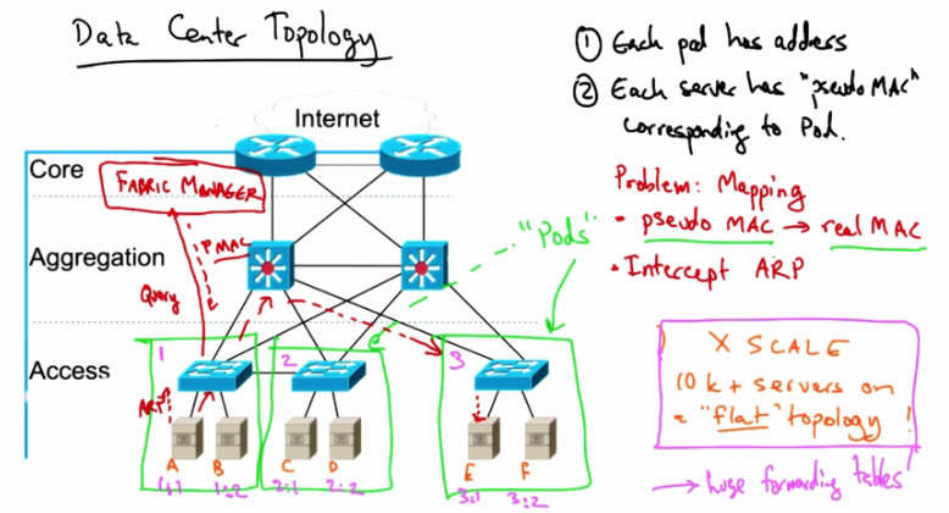
makes scaling difficult, since now we have tens of thousands of servers on a single flat topology. In other words, layer-two addresses are not topological. So the forwarding tables in these switches can't scale as easily, because they can't take advantage of the natural hierarchy that exists in the topology. Other problems that exist in this type of topology is that the hierarchy can potentially create single points of failure, and links at the top of the topology, in the core, can become oversubscribed. Modern data center operators have observed that as you move from the bottom of the hierarchy up towards the core, that the links at the top can carry as much as 200 times as much traffic as the links at the bottom of the hierarchy. So there's a serious capacity mismatch in that the top part of the topology has to carry a whole lot more traffic than the bottom. We'll explore how modern data center network architectures address these various challenges, but let's first take a quick look at one way of solving the scale problem.

Data Center Topologies



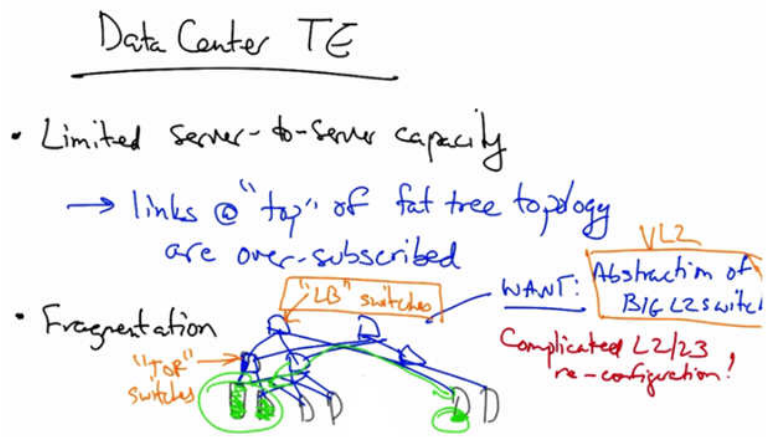
Recall that the Scale problem arises because we have tens of thousands of servers on a flat layer two topology, where all of the servers have a topology independent MAC or hardware address and thus, in the default case every switch in the topology has to store a forwarding table entry for every single MAC address. One solution is to introduce what are called pods and assign pseudo MAC addresses to each server corresponding to the pod in which they're location is in the Topology. So in addition to having a real MAC address, each server has what's known as a pseudo-MAC address, as shown in pink. Thus, switches in the Data Centre Topology no longer need to maintain forwarding table entries for every host. They only need to maintain entries for reaching other pods in the Topology. Once a frame answers a pod, the switch then of course has entries for all of the servers inside that pod but they don't need to maintain entries for the MAC addresses for servers outside of each pod. For example, the switch in pod one only needs to maintain entries for these two servers with MAC addresses A and B, but it doesn't need to maintain independent entries with servers with MAC addresses C and D. It only needs to maintain an entry for how to reach pod 2. Likewise for pods 2 and pods 3. Now, in such a Data Centre Topology, of course, these hosts are unmodified so they're still going to respond to things

like ARP queries with their real MAC addresses. So we need a way of dealing with that, as well as a way of Mapping pseudo MAC addresses to real MAC addresses.



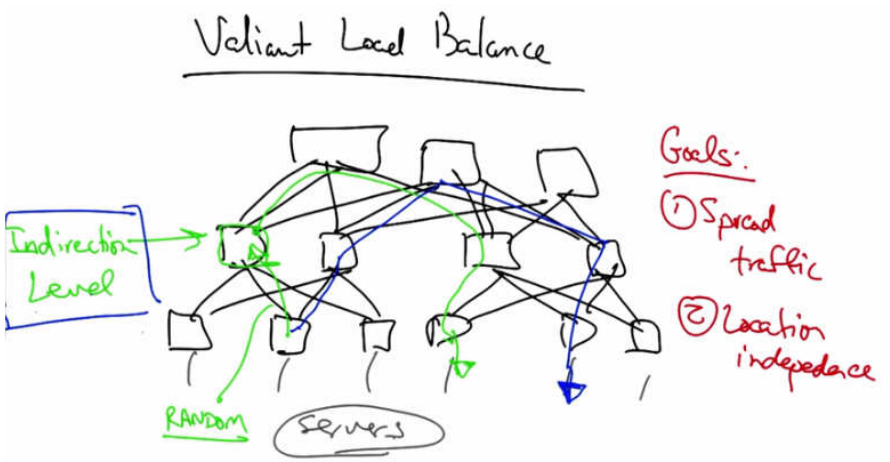
The solution is as follows. When a host such as server A issues an ARP query, that query is intercepted by the switch,. But instead of flooding that query, the switch intercepts the query and forwards it to an entity called the Fabric Manager. The Fabric Manager then responds with the pseudo-MAC corresponding to that IP address. Host A then sends the frame with the destination pseudo-MAC address, and switches in the Topology can forward that frame to the appropriate pod sorresponding to the pseudo MAC address of the destination server. Once the frame reaches the destination pod, let's say in this case pod 3, the switch at the top of that pod can then map the pseudo MAC address back to the real MAC address. And the server that receives the frame receives an Ethernet frame with its real destination MAC address, so it knows that the Ethernet frame was intended for it. By intercepting our queries in this way and providing a Mapping between topological pseudo MAC addresses and real, physical MAC addresses, we can achieve hierarchical forwarding in a large Layer 2 Topology without having to modify any host software.

Data Center (Intradomain) Traffic Engineering



In this lesson, we will look at how data center traffic engineering, through customized topologies and special mechanisms for load balance, can help reduce link utilization, reduce the number of hops to reach the edge of the data center, and make the data center network easier to maintain. We saw in the last lesson how existing data center topologies provide extremely limited server to server capacity because of the over subscription of the links at the top of the hierarchy. Additionally, as services continue to be migrated to different parts of the data center, resources can be fragmented, significantly lowering utilization. For example, if the service denoted by green is running mostly in one part of the data center, but there's a little bit running on a virtual machine in another part of the data center, this might cause traffic to traverse links of the data center topology hierarchy, thus significantly lowering utilization and cost efficiency. Reducing this type of fragmentation can result in complicated layer two or layer three routing reconfiguration. What we'd like to have is just the abstraction of one large layer to switch. This is the abstraction that VL2 provides. So, VL2 has two main objectives. One is to achieve layer-two semantics across the entire data center topology. This is done with a name-location separation and a resolution service that resembles the fabric manager which we talked about in the last lesson and which is described in more detail in the paper. To achieve uniform high capacity between the servers and balance load across links in topology, VL2 relies on flow-based random traffic interaction using valiant load balancing. Let's take a closer look at how that load balancing works.

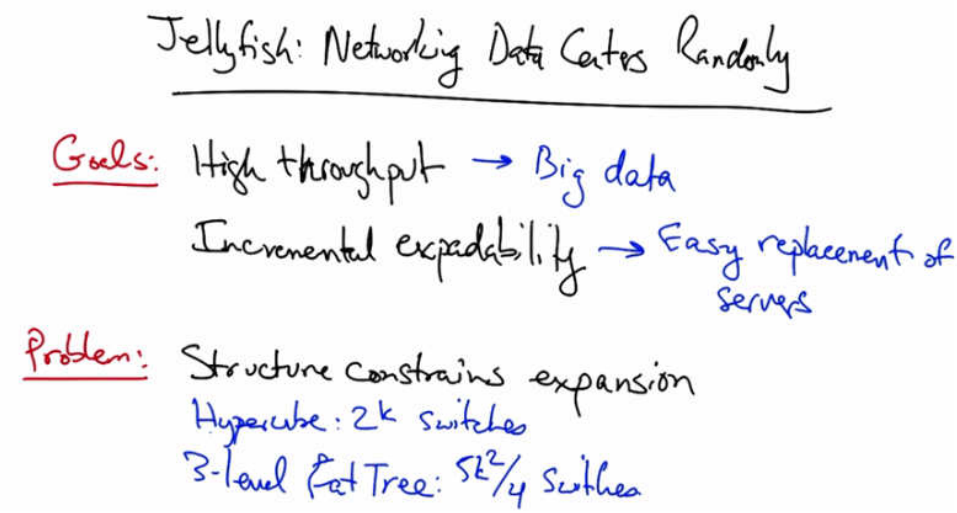
Valiant Load Balancing



The goals of Valiant load balancing in the VL2 network are to spread traffic evenly across the servers, and to ensure that traffic load is balanced independently of the destinations of the traffic flows. Field two achieves this by inserting an indirection level into the switching hierarchy. When a switch at the access layer wants to send traffic to a destination, it first selects a switch at the indirection level to send the traffic at random. This intermediate switch then forwards the traffic to the ultimate destination depending on the destination MAC address of the traffic. Subsequent flows might pick different indirection points for the traffic, at random. This notion of picking a random indirection point to balance traffic more evenly across a topology, actually comes from multi-processor architectures and has been rediscovered in the context of data centers. So, in this lesson we have explored how valiant load balancing can be used on a slightly

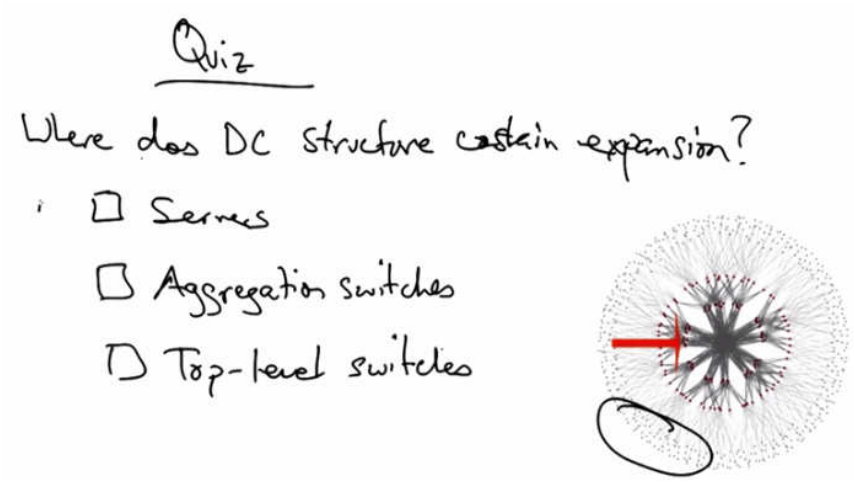
modified topology to achieve better load balance than in traditional fat tree networks, without an indirection layer and valiant load balancing. In the next lesson we'll look at how a custom random topology can make some of these traffic engineering problems even easier.

Jellyfish Data Center Topology



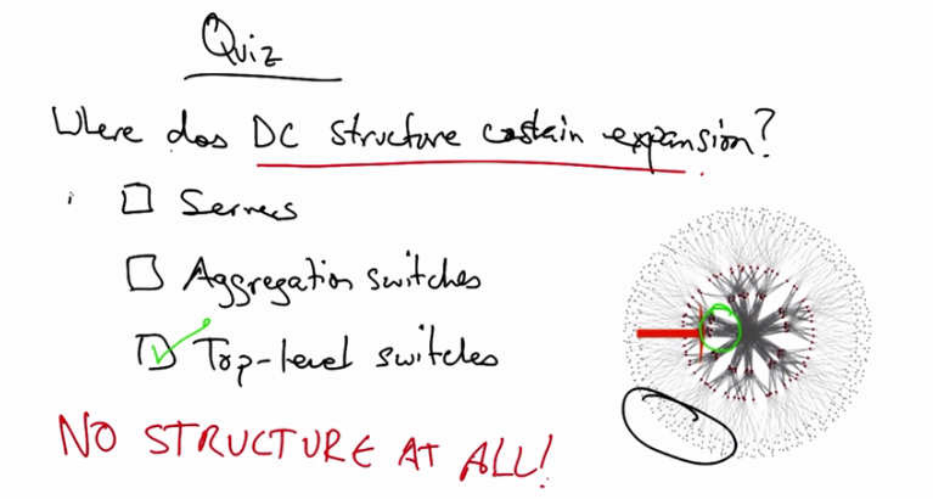
In this lesson we'll look at Jellyfish, a technique to network data centers randomly. The goals of Jellyfish are to achieve high throughput to support, for example, big data analytics or agile placement of virtual machines and incremental expandability, so that network operators can easily add or replace servers and switches. For example, large companies like Facebook are adding capacity on a daily basis. Commercial products make it easy to expand or provision servers in response to changing traffic load but not the network. Unfortunately, the structure of the data center networks constrains expansion. Structures such as a hypercube require two to the K switches, where K is the number of servers. Even more efficient topologies, like a FAT tree, are still quadratic in the number of servers.

Data Center Topology Quiz



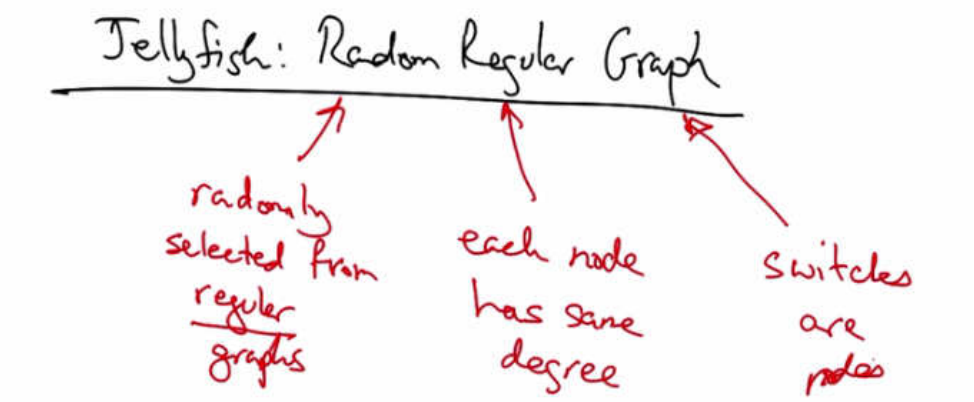
By looking at this figure showing a data center topology where the servers are at the edge of the graph, can you try to figure out where data center topology primarily constrains expansion? Is it at individual servers, aggregation switches or top-level switches?

Data Center Topology Quiz Answer

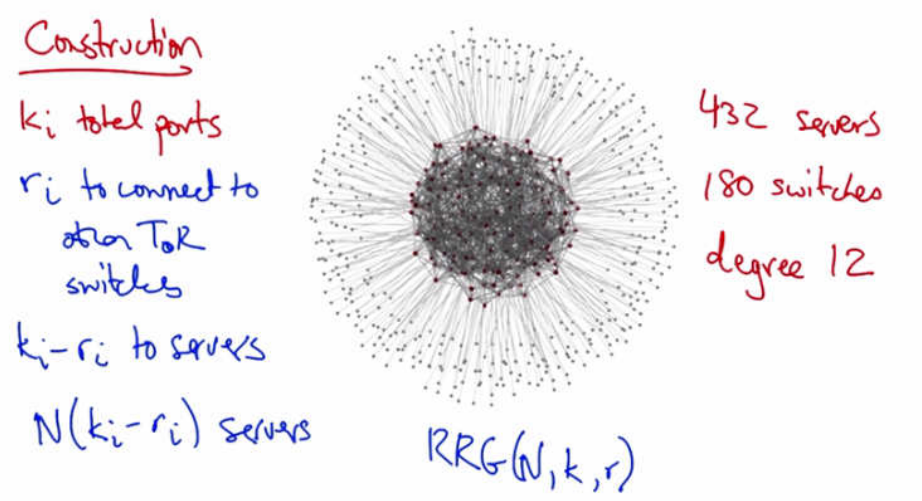


As we can see from the figure, most of the congestion occurs at the top level. Jellyfish's answer to how data structure constrains expansion is to simply have no structure at all.

Jellyfish Random Regular Graph

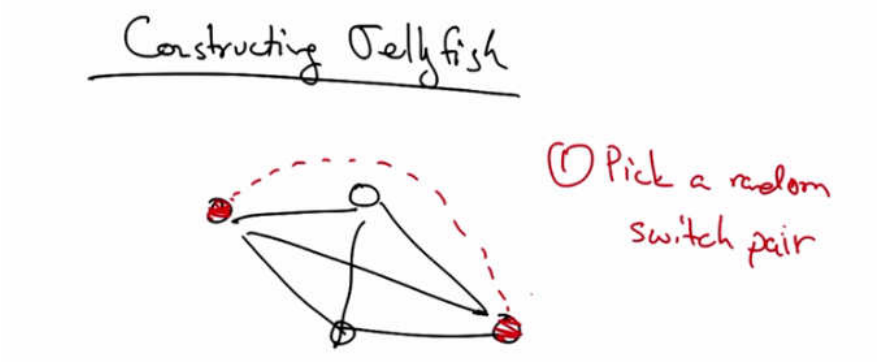


Jellyfish's topology is what is called a random regular graph. It's random because each graph is uniformly selected at random from the set of all regular graphs. A regular graph is simply one where each node has the same degree. And a graph in Jellyfish is one where the switches in the topology are the nodes. In contrast to the earlier data center topology diagram we saw, here is a picture of a Jellyfish random graph with 432 servers and 180 switches. Every node in this graph has a fixed degree of 12.

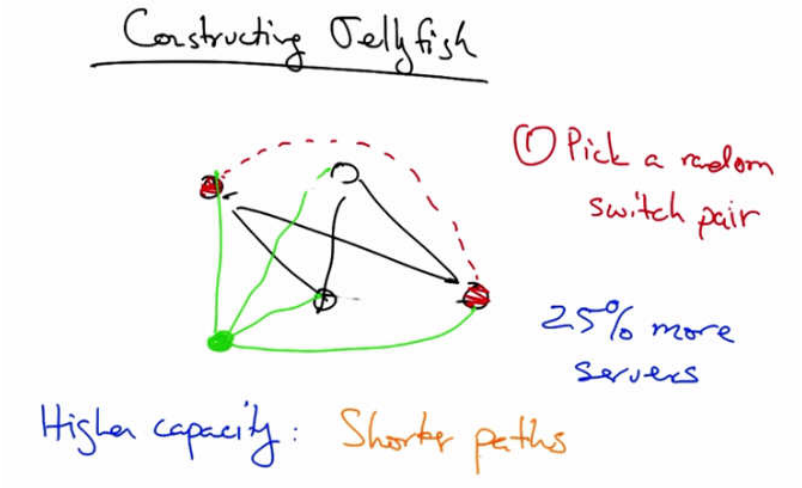


Jellyfish's approach is to construct a random graph at the Top-of-Rack switch layer. Every Top-of-Rack switch i, has some total number of K_i ports, of which it uses R_i to connect to other Top-of-Rack switches. The remaining K_i minus R_i ports are used to connect servers. With N racks, the network then supports N times K_i minus R_i servers. And the network is a random regular graph denoted as follows. Formally, random regular graphs are sampled uniformly from the space of all R regular graphs. Achieving such a property is a complex graph theory problem, but there's a simple procedure that produces a sufficiently uniform random graph that empirically have the desired properties.

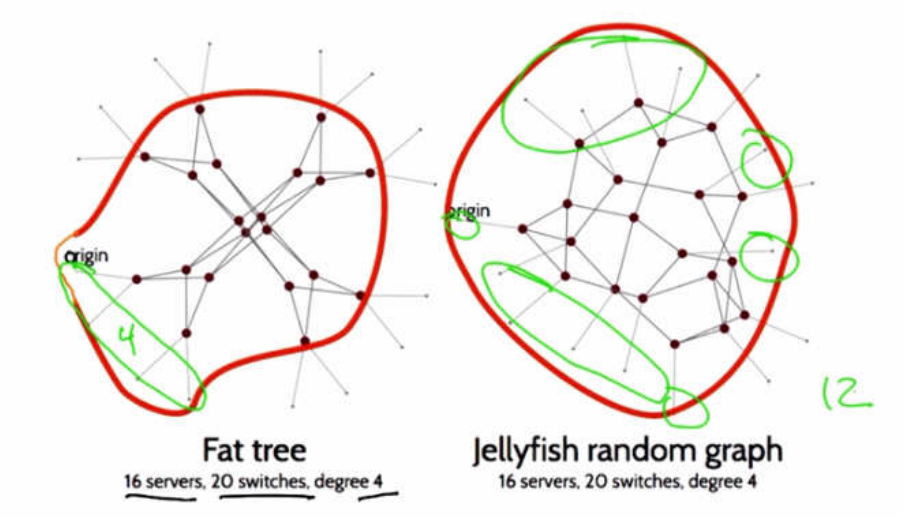
Constructing a Jellyfish Topology



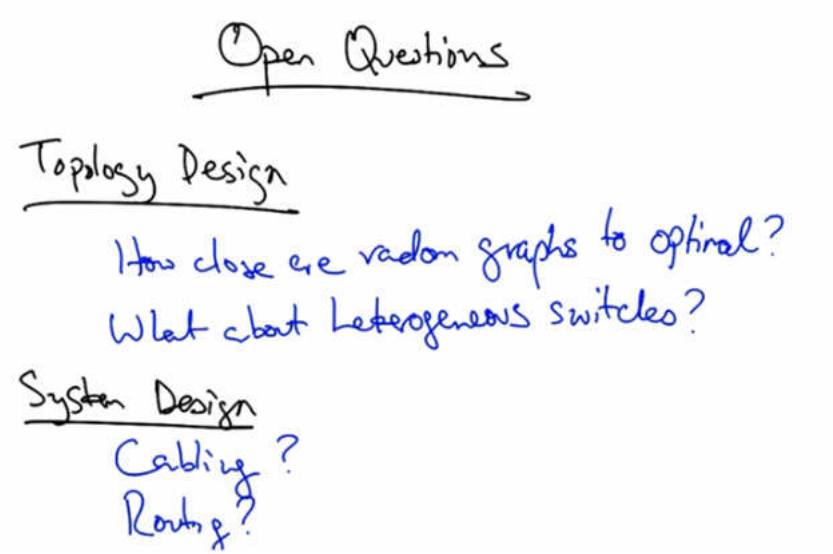
To construct a jellyfish topology, one can simply take the following steps. First, pick a random switch pair with free ports for which the switch pair are not already neighbors. Next, join them with a link, and repeat this process until no further links can be added. If a switch remains with greater than or equal to two free ports, which might happen during the incremental expansion by adding a new switch, these switches can be incorporated in the topology by removing a uniform random existing link and adding links to that switch.



For a particular equipment cost, using identical equipment, the jelly fish topology can achieve increased capacity by supporting twenty five percent more servers. This higher capacity is achieved because the paths through the topology are shorter than they would be in a Fat tree topology.



Consider a topology with sixteen servers, twenty switches, and a fixed degree of four for both the fat tree topology and the jellyfish random graph. In the fat tree topology, only four of 16 servers are reachable in less than five hops. In contrast, in the jellyfish random graph, there are 12 servers reachable. By making more servers reachable along shorter paths, jellyfish can increase capacity over a conventional Fat tree topology.



So while Jellyfish shows some promise, there are certainly some open questions. First, how close are these random graphs to optimal in terms of the optimal throughput that could be achieved for a particular set of equipment? Second, what about typologies where switches are heterogeneous with different numbers of ports or link speeds? From a system design perspective, the random topology model could create problems with physically cabling the datacenter network, and there are also questions about how to perform routing or congestion control without the structure of a conventional datacenter network like a fat tree.

CS 6250: Computer Networking

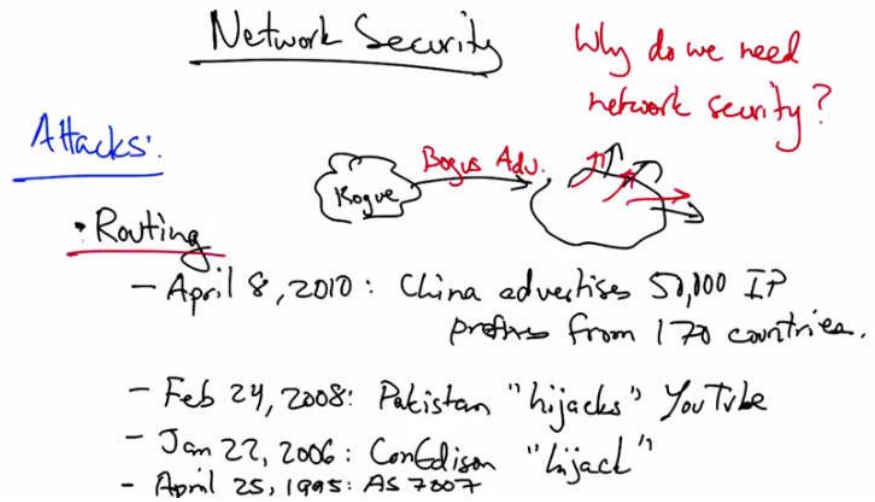
Lecture 11.0: Network Security

Networking Security Intro

We're on our last stretch of the course with network security. Network security is an extremely important topic, especially considering high-profile data loss from large companies and government organizations.

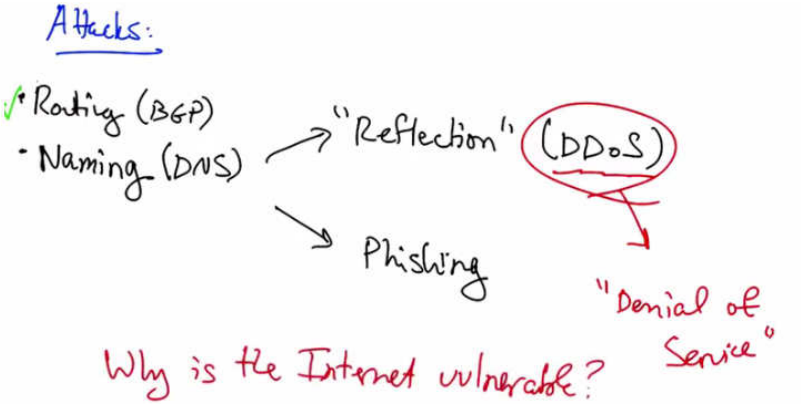
To accompany this section, you'll be building a program in Pyretic that mitigates one of the specific attacks that we're going to look at in this course.

Need for Network Security



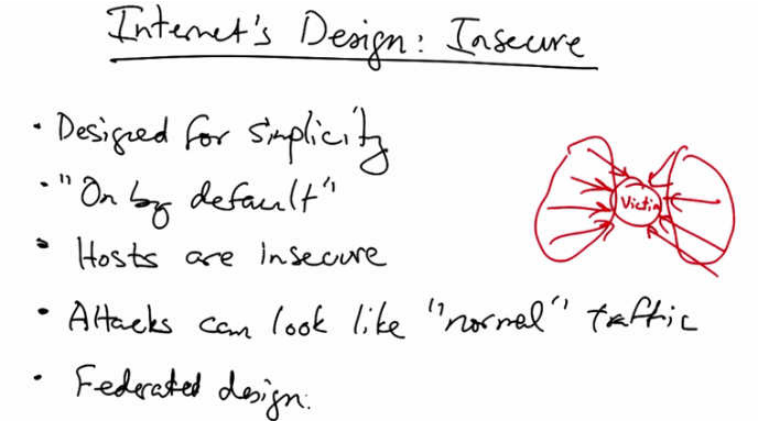
We are beginning a lesson on network security. Lets first talk about why we need network security in the first place. The Internet is actually subject to a wide variety of attacks on various parts of the infrastructure. One part of the infrastructure that can be attacked is routing. So the internet's routing protocol, the border gateway protocol, is notorious for being susceptible to different kinds of attacks. For example, on April 8, 2010, China advertised about 50,000 blocks of IP address from 170 different countries. The event lasted for about 20 minutes. In this particular case, the hijack appears to have been accidental because the prefixes were long enough such that they didn't disrupt existing routes. But the fact that the route advertisements were allowed to leak in the first place highlights the vulnerability of the border gateway protocol. Effectively, the border gateway protocol essentially allows any AS to advertise an IP prefix to a neighboring AS, and that AS will typically just believe that route advertisement and advertise it to the rest of the internet. These events that occur where an AS advertises a prefix that it does not own are called route highjacks. And they tend to occur more often than one might expect. In

addition to the event on April 8, 2010, another event in 2008 occurred when Pakistan higjacked YouTube prefixes, potentially as a botched attempt to block Youtube in the country following a government order. Unfortunately, the event resulted in disruption of connectivity to YouTube for people all around the world. In January of 2006 ConEdison accidentally hijacked a lot of transit networks, including level three Unet and several other large ISPs disrupting connectivity to many customers. And on April 25th in 1995, one of the more famous route hijack incidents was the AS7007 incident, where AS7007 advertised all of the IP prefixes on the entire internet as originating in its own AS, resulting in disruption of connectivity to huge fractions of the Internet. So we've surveyed some famous or, shall we say, notorious attacks on Internet routing, but another part of the infrastructure that's vulnerable is naming or the DNS.



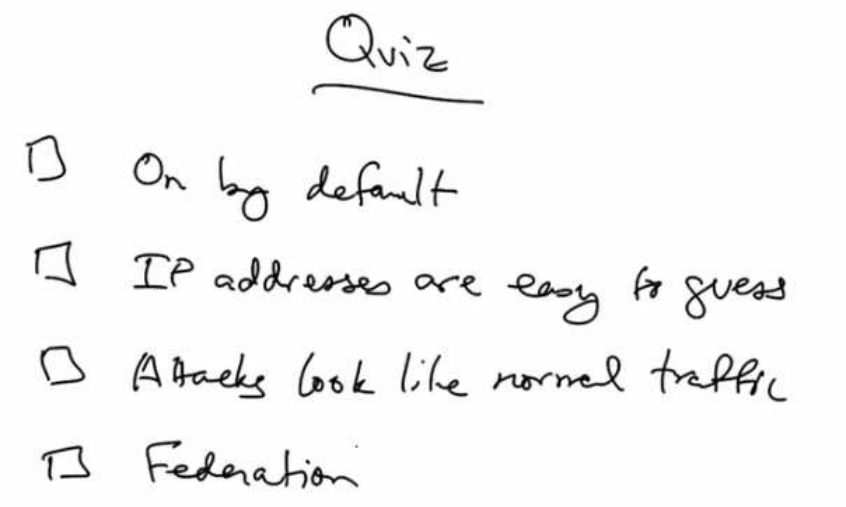
One very popular and effective means of mounting an attack on the naming system is through something called reflection. DNS reflection is a way of generating very large amounts of traffic targeted at a victim in an attack called Distributed Denial of Service, or DDoS attack. Another type of attack on the naming system is Phishing, whereby an attacker exploits the domain name system in an attempt to trick a user into revealing personal information, such as passwords on a rogue website. In general, denial of service attacks are extremely common and can be mounted in a variety of different ways. DNS reflection is just one way that distributed denial of service attacks are mounted. We'll explore some others later on in this lesson. It's worth asking why the internet is so vulnerable to different kinds of attacks.

Internet is Insecure



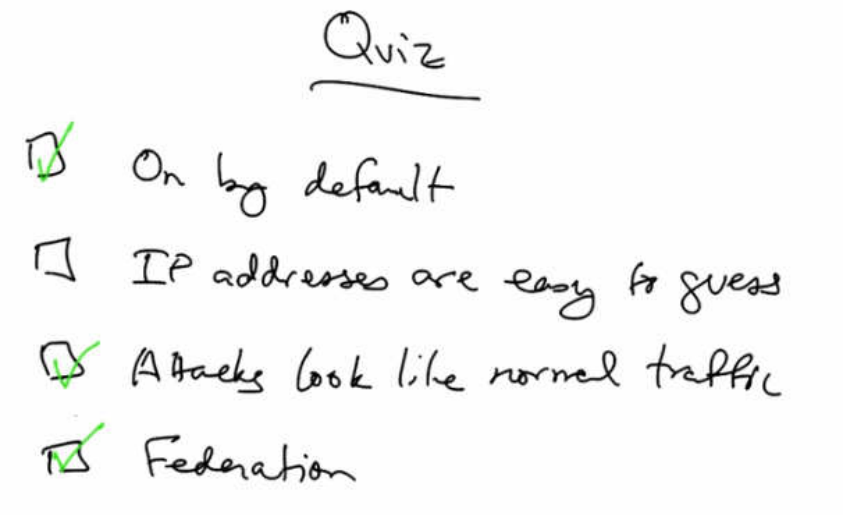
As it turns out, the internet's design is actually fundamentally insecure. Many explicit design choices have caused the internet to be vulnerable to different types of attacks. The internet was designed for simplicity, and as a result security was not a primary consideration when the internet was originally designed. Another aspect of the internet's design is that it's on by default. In other words, when a host is connected to the internet, it is by default reachable by any other host that has a public IP address. This means that if one has an insecure host, that host is effectively wide open to attack by other hosts on the internet. Now, this wasn't a primary design consideration when the internet consisted of a small number of trusted networks, but as the internet has continued to grow, this on-by-default design, or the notion that any host should always be reachable by any other host, has come under fire. Part of the reason that they're on-by-default model does not work that well is that hosts are insecure. This makes it possible for a remote attacker to compromise a machine that's connected to the internet and commandeer it for the purposes of attack. In many cases, an attack might actually just look like normal traffic. For example, in the case of an attack on a victim web server, every individual request to that web server might look normal, but the collection of requests together, mounted as part of a distributed denial of service attack, might add up to a volume of traffic that the server is unable to handle. Finally, the internet's federated design obstructs cooperation for diagnosis or mitigation. In other words, because the internet is run by tens of thousands of independently run networks, it can be very difficult to coordinate a defense against an attack because each of these networks is run by different network operators, sometimes in completely different countries

Internet Insecurity Quiz



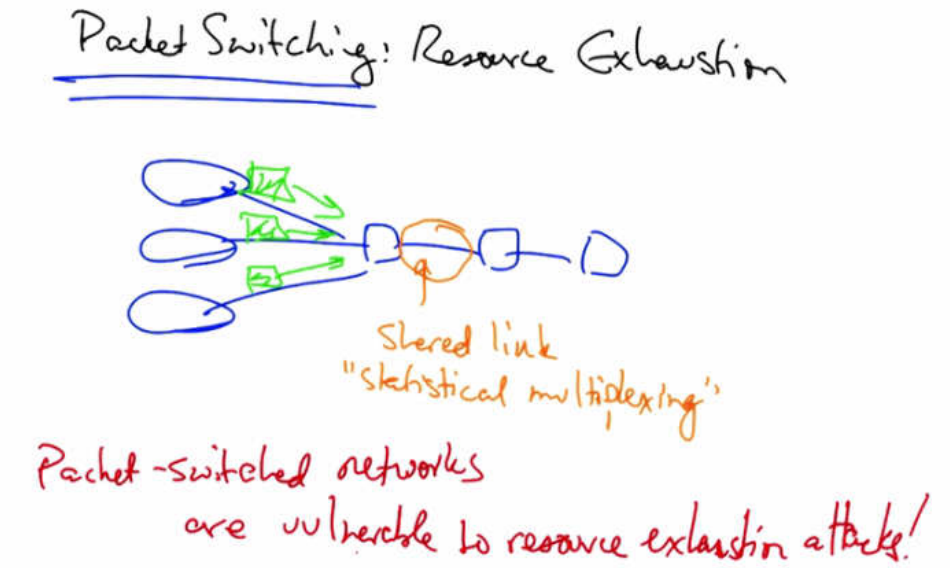
As a quick quiz, which of the following make the internet's design fundamentally insecure? The On by default nature of the design? The fact that IP Addresses might be easy for an attacker to guess? That attacks can look like normal traffic? Or that the internet is actually a federation of tens of thousands of independently operated networks?

Internet Insecurity Quiz Answer



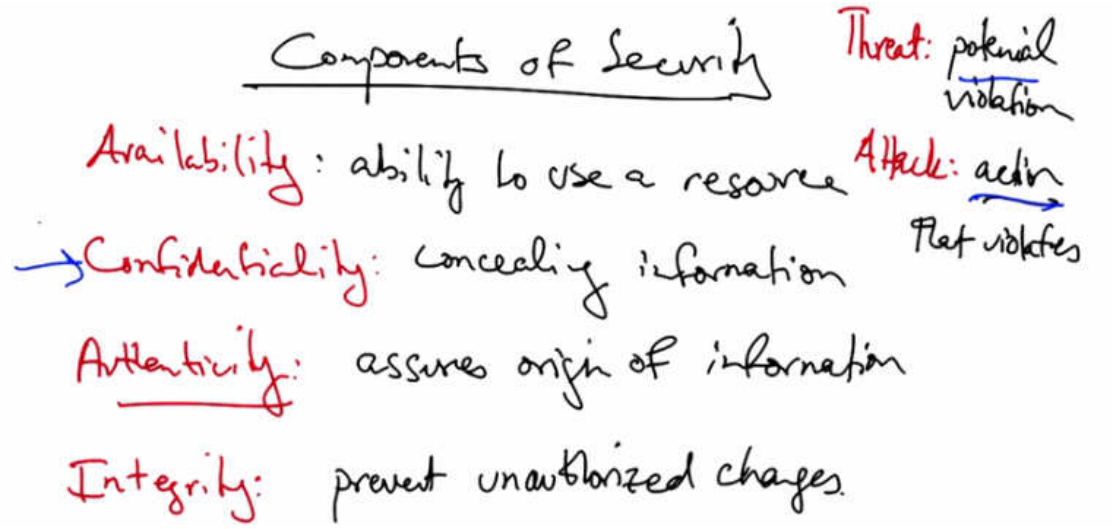
The fact that the Internet is on by default, that attacks can look like normal traffic, and that the Internet is in fact federated, collectively make it very difficult to design a secure Internet.

Resource Exhaustion Attacks



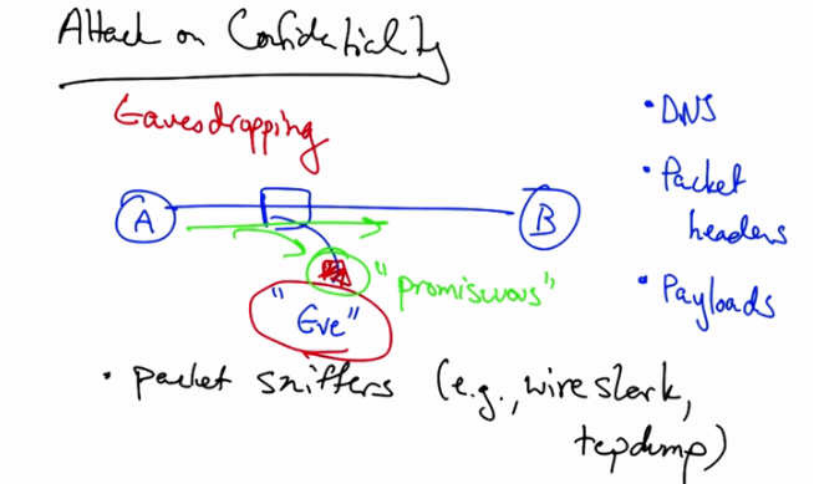
Recall from an earlier lesson that one of the internet's fundamental design tenants is packet switching. In a packet switch network, resources are not reserved and packets are self contained. Every packet has a destination IP address, and each packet travels independently to the destination host. In a packet switch network, a link may be shared by multiple senders at any given time, using statistical multiplexing as we learned in previous lessons. While packet switch networks have their advantages, in particular it makes it easy to achieve high utilization on a shared link, packet switch networks also have the drawback that a large number of senders can

overload a network resource, such as a node or a link. Note that circuit switch networks like the phone network do not have this problem because every connection effectively has allocated, dedicated resources for that particular connection until it is terminated. So this problem that an attacker who sends a lot of traffic might exhaust resources is unique to a packet switched network environment. So packet switched networks are extremely vulnerable to resource exhaustion attacks. Resource exhaustion attacks a basic component of security known as availability.

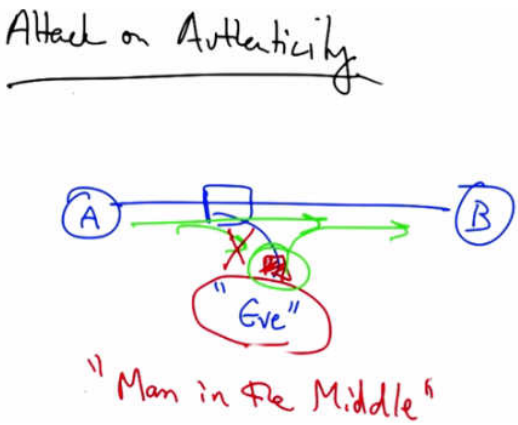


Let's take a look at other components of security as well. In addition to availability, we would like the network to provide confidentiality. For example, if you're performing a sensitive banking transaction or having a private conversation with a friend, you'd like the Internet to provide some level of confidentiality. Another component of security is authenticity. Authenticity ensures the identity of the origin of a piece of information. So, for example, if you're reading a particular news article, you really may want to know that the article came from the New York Times website as oppose to from some other place on the internet. Similarly, you might want to know that that information wasn't modified in flight. That property is called integrity, which prevents unauthorized changes to information as it traverses the network. Now a security threat is anything that might potentially cause a violation of one of these properties. An attack, on the other hand, is an action that results in the violation of one of these security properties. So the difference between a threat and an attack is simply the difference between a violation that could potentially occur versus an action that actually results in a violation. Let's look at a couple example attacks on different components of security. Let's start by looking at an attack on confidentiality.

Confidentiality and Authenticity Attacks

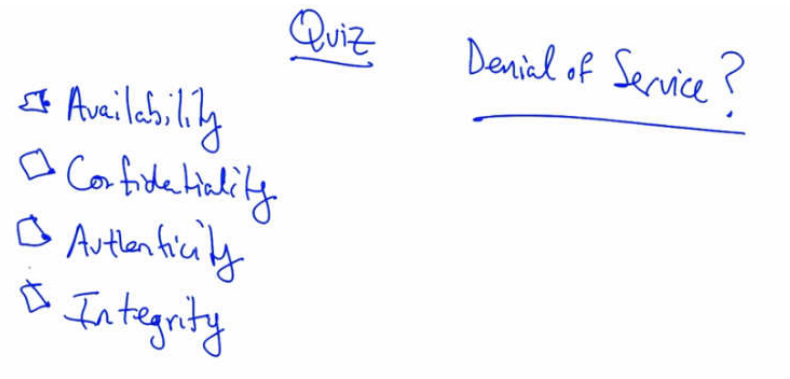


One attack on confidentiality is called eavesdropping, where an attacker, Eve, might gain unauthorized access to information being sent between Alice and Bob. So, for example, if Alice and Bob were chatting on instant message, or if Alice sends an email to Bob, the potential exists (in other words, there's a threat) that Eve might be able to hear that communication. There are various packet sniffing tools, such as wireshark and tcpdump, that set a machine's networking interface card into what's called promiscuous mode. If Alice, Bob, and Eve are on the same local area network, where packets are being flooded (for example, if they were being connected by a hub that flooded all packets everywhere, or if the learning switch did not have an entry for Alice or Bob) then Eve might be able to hear some of those packets. If the network interface card is in promiscuous mode, then Eve's machine will be able to capture some of the packets that are being exchanged between Alice and Bob. It's worth thinking about how different types of traffic might reveal important information about communication. For example, the ability to see DNS look-ups would provide the attacker information about, say, what websites you're visiting. The ability to capture packet headers might give the attacker information not only about where you're exchanging traffic, but what types of applications you're using. And the ability to see a full packet payload would allow an attacker to effectively see every single thing that you are sending on the network including content you're exchanging with other people, uch as private message, email communication, and so forth.



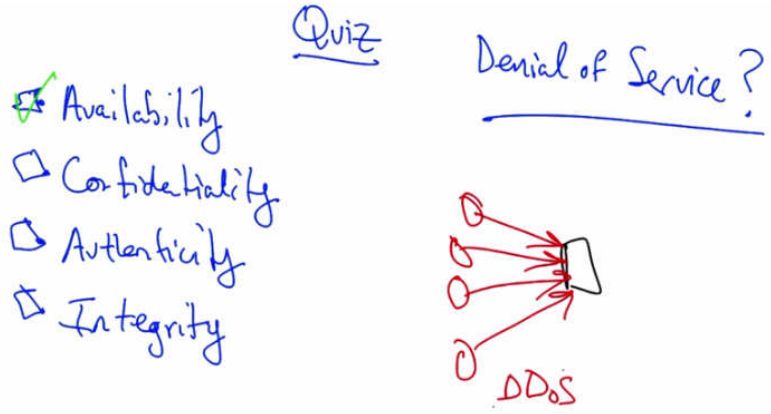
Given the ability to see a packet, Eve might not only listen to that packet, but might also modify it and re-inject it into the network, potentially after altering the state of the packet. If additionally Eve could suppress the original message, let's consider an attack on authenticity. If, in addition to being able to observe packets that traverse the network, Eve could re-inject packets after having modified them and suppress Alice's original message, then Eve could effectively impersonate Alice. This is sometimes called a 'Man in the Middle' attack. Eve could also make it appear as though this message came from Alice, in which case the attack would be an attack on message integrity.

Network Attack Quiz



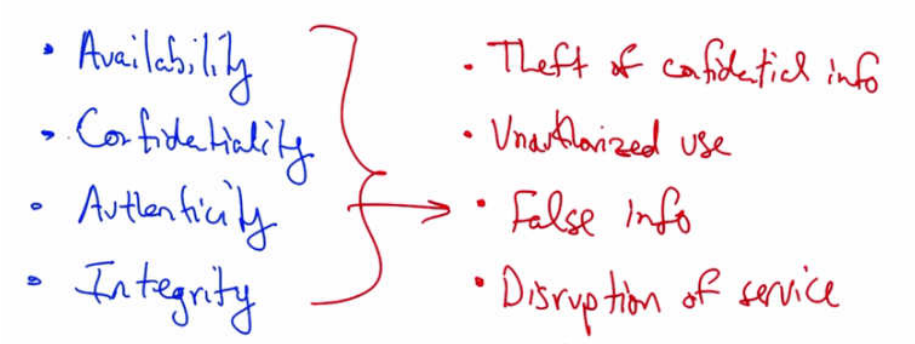
So we've considered attacks on availability, confidentiality, authenticity, and integrity. Let's have a quick quiz, on these concepts. A denial of service is an attack on what property of internet security?

Network Attack Quiz Answer



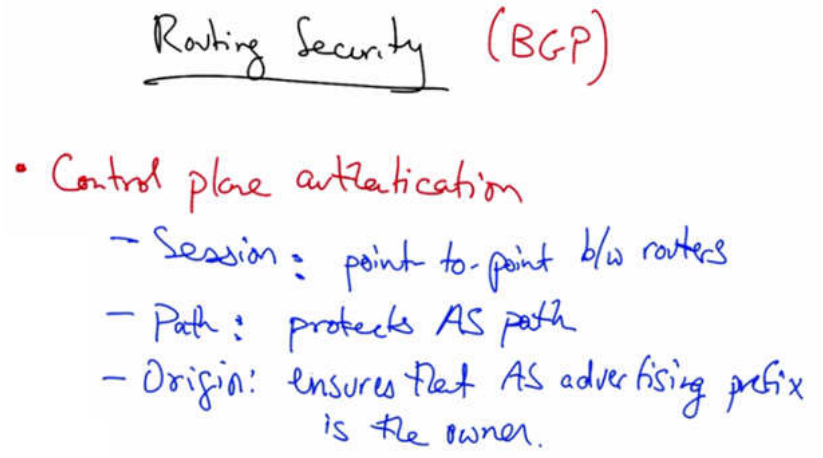
A denial of service attack is an attack on availability. Denial of service attacks typically are an attempt to overwhelm the network or a network host in some way by consuming its resources. A common way of launching a denial of service attack is to send a lot of traffic at a victim, often from many distributed locations. If the attacker is in fact distributed, this is called not just a denial of service attack, but a distributed denial of service attack.

Negative Impacts of Attacks



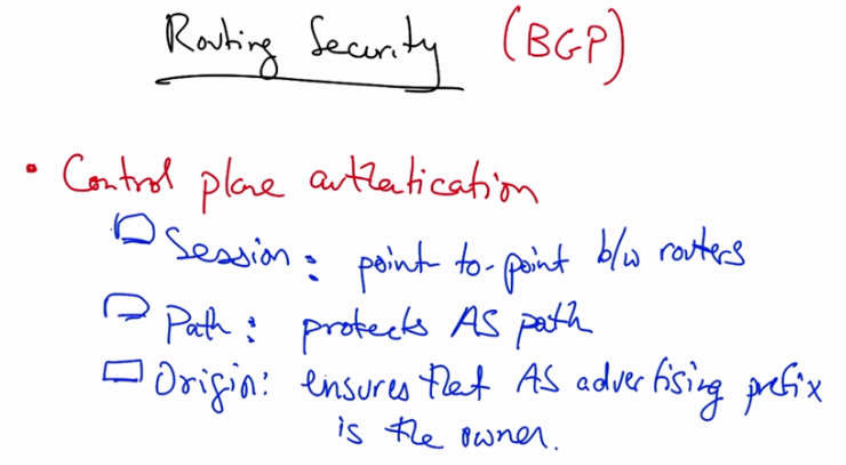
These attacks can have serious negative effects, including theft of confidential information, unauthorized use of network bandwidth or computing resources, the spread of false information, and the disruption of legitimate services. All these types of attack are related. They are all very dangerous and sometimes they come hand in hand. For example, all these attacks are, in some sense, related to one another, and they can come hand in hand with one another as well.

Routing Security



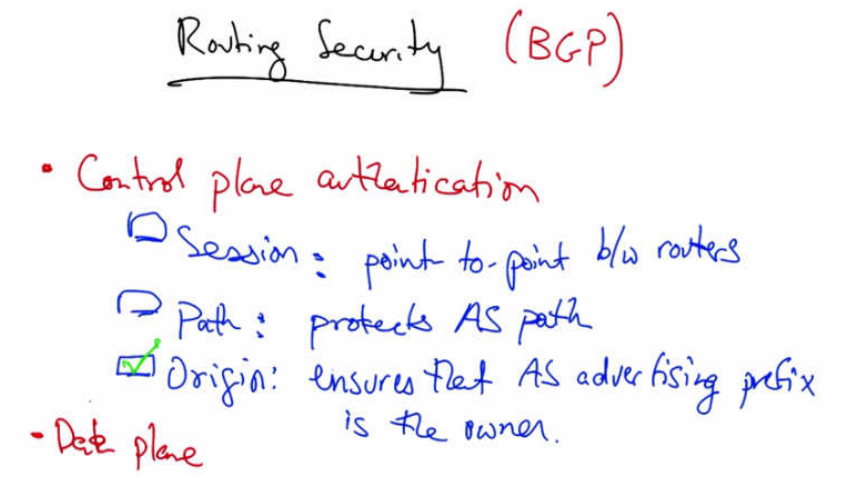
Let's now talk about internet routing security or problems involving securing the internet's routing protocol. We will primarily focus on inter-domain routing or the security of BGP. We will further focus on control plane security which typically involves authentication of the messages being advertised by the routing protocol. In particular, the goal of control plane security, or control plane authentication, is to determine the veracity of routing advertisements. There are various aspects of the routing protocol that we seek to verify. One is session authentication, which protects the point-to-point communication between routers. A second type of control plane authentication is path authentication, which protects the AS path and sometimes other attributes. Another type of authentication is origin authentication, which protects the origin AS in the AS path, effectively guaranteeing that the origin AS that advertises a prefix is, in fact, the owner of that prefix.

BGP Routing Security Quiz



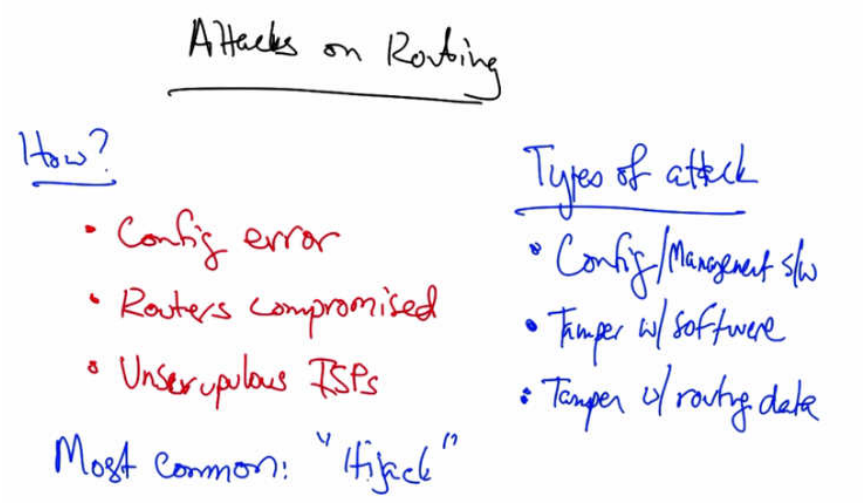
So as a quick quiz. From last lesson, we talked about route hijacks. A route hijack is an attack on which of the following three forms of authentication?

BGP Routing Security Quiz Answer



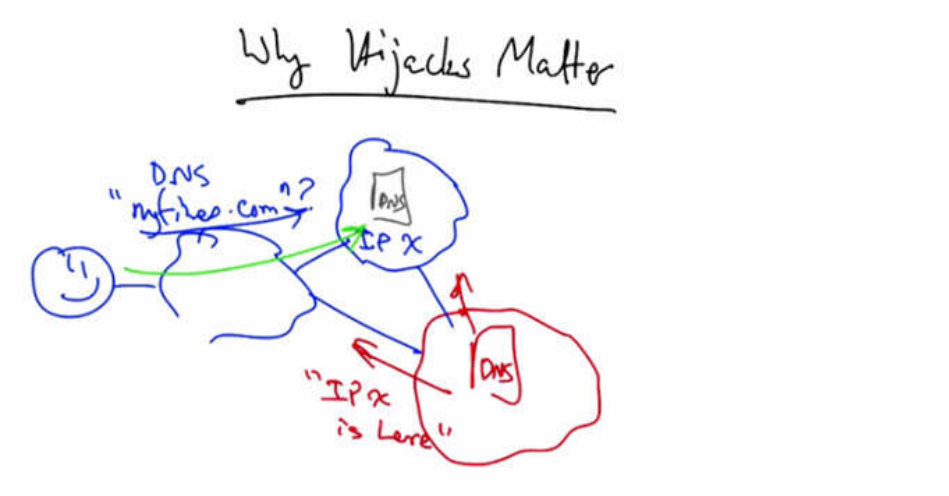
A route hijack is an attack on origin authentication because in a route hijack, the AS that is advertising the prefix is actually not the rightful owner of that prefix. In addition to control plan security, we also have to worry about data plan security or determining whether data is traveling to the intended locations. In general, it can be extremely hard to verify that packets or traffic is traveling along the intended route to the destination, or that it, in fact, even reaches the intended destination in the first place. Guaranteeing that traffic actually traverses the advertised route remains an important open problem in internet security. So how do these attacks on routing happen in the first place?

Route Attacks

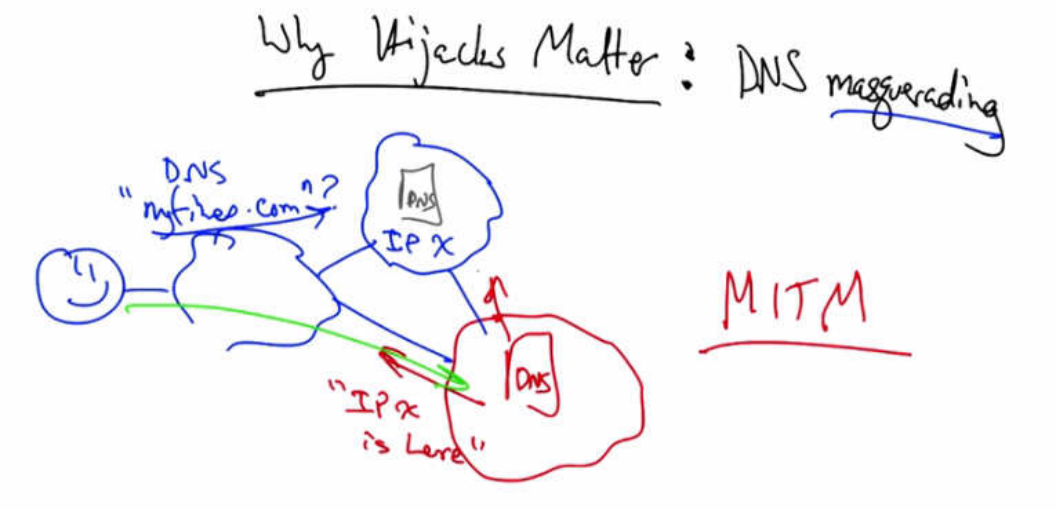


One possible explanation is simply that the router is misconfigured. In other words, no one actually intended for the router to advertise a false route, but because of a misconfiguration the router does so. The AS 7007 attack that we discussed last time was actually the result of a configuration error. Second, a router might be compromised by an attacker. Once a router is compromised, the attacker can reconfigure the router to, for example, advertise false routes. Finally, unscrupulous ISPs might also decide to advertise routes that they should not be advertising. To launch the attack, an attacker might reconfigure the router, which is typically the most common way an attacker might launch an attack. The attacker might also tamper with software, or an attacker could actively modify a routing message. In addition to tampering with the configuration, the attacker might tamper with the management software that changes the configuration. And the most common attack is a route hijack attack or an attack on origin authentication.

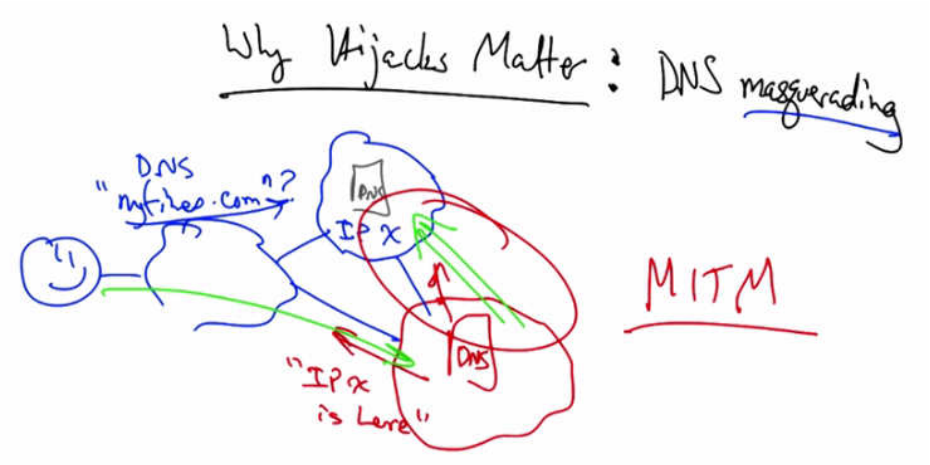
Route Hijacking



Let's talk about why hijacks matter. Let's suppose that you would like to visit a particular website. To do so you first need to issue a DNS query. Now the authoritative DNS server for a particular domain might be located in a distant network. As we've discussed in previous lessons, the DNS uses a hierarchy to direct your query to the location of the authoritative name server, but ultimately that authoritative name server has an IP address, and you use the internet's routing protocol, the border gateway protocol, to reach that IP address. What if an attacker were running a rogue DNS server and wanted to hijack your DNS query or to return a false IP address? Well, the attacker might use BGP to advertise a route for the IP prefix that contains that authoritative DNS server.

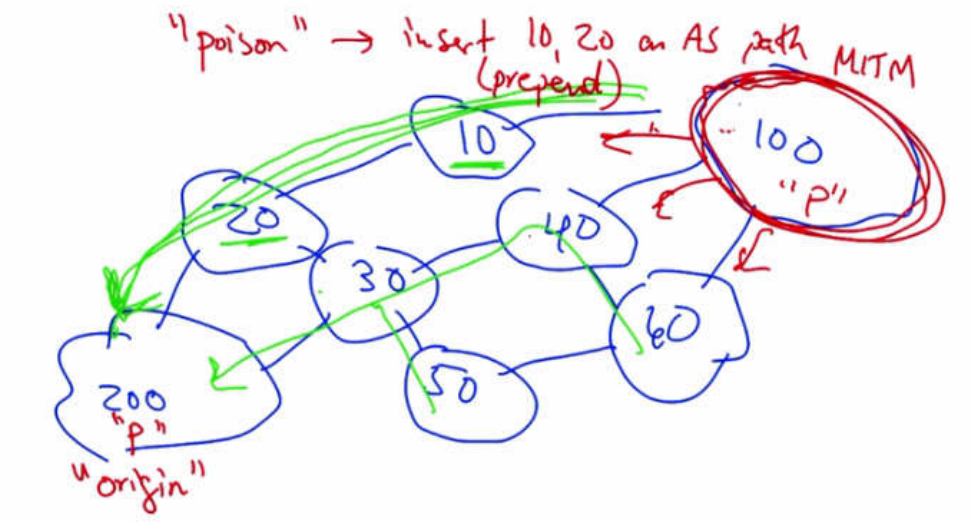


And suddenly your DNS queries that were previously going to the legitimate server, are instead redirected to the rogue DNS server. So we might think of this as an attack whereby an attacker can use the BGP infrastructure to hijack a DNS query, and masquerade as a legitimate service. It can get even worse than this. Let's now look at how a BGP route hijack can result in a Man in the Middle attack, whereby your traffic ultimately reaches the correct destination, but the attacker successfully inserts themselves on the path.

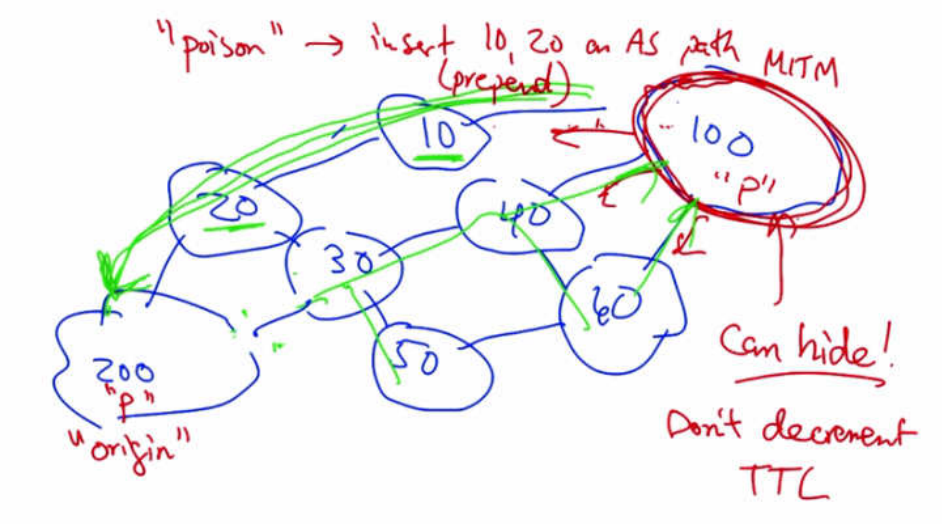


The problem with this particular route hijack is that all traffic destined for IP X is going to head for the attacker, even the traffic from the legitimate network. What we'd like to instead have happened is that traffic for IP X first goes to the hijack location and then goes to the legitimate location. So the attacker effectively becomes a Man in the Middle. The problem is that we need to somehow disrupt the routes to the rest of the internet while leaving the routes between the attacker and the legitimate location intact so that traffic along this path can still head towards the legitimate AS.

Route Hijacking (cont)

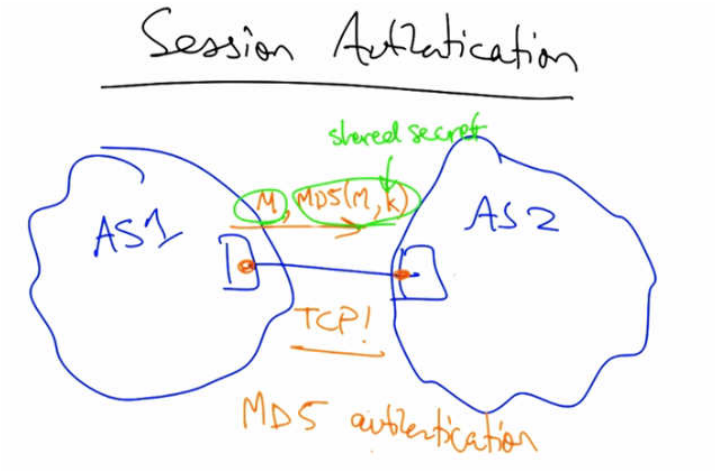


Let's suppose that AS200 originates a prefix and that the paths that result from the original BGP routing are shown in green. Let's now suppose that AS100 seeks to become a man in the middle. If the original prefix being advertised was P, AS100 could also advertise the prefix P. But we want to make sure that AS100 maintains a path back to AS200. Now that path already exists, it's right here. So what we want to do is make sure that neither AS 10 nor AS 20 accept this hijacked route. The way that we can do that is through a technique called AS-path poisoning. So, if AS 100 advertises a route that includes AS 10 and AS 20 in the AS path, both of these AS's will drop the announcement because they will think they've already heard the announcement and don't want to form a loop.



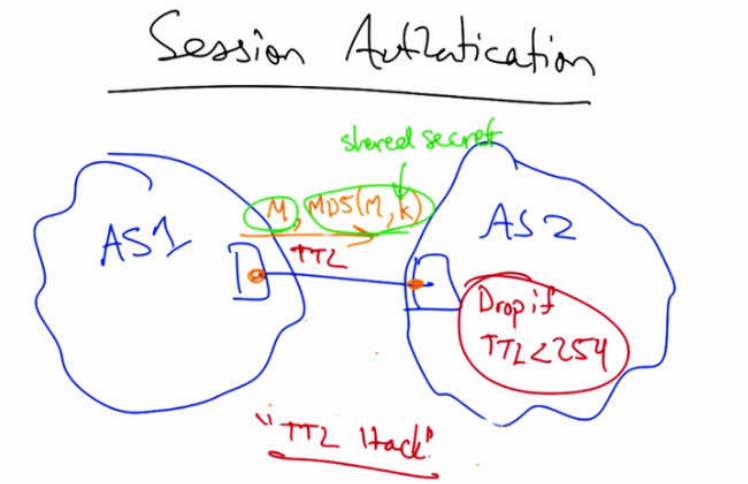
On the other hand, the other AS's on the internet (in other words, every other AS that's not on the path back from 100 to 200) will switch and now all of the traffic from other AS's enroute to AS 200 will traverse the attacker AS100. Now a trace route might look awfully funny taking this circuitous route, but actually the attacker can hide its presence even if the sender is running a trace route. Recall that a trace route simply consists of ICMP time exceeded messages that result when a particular packet reaches a TTL of 0. Now typically each router along a path will decrement the TTL at each hop. But if the routers in the attacker's network never decrement the TTL, then no time exceeded messages would be generated by routers in AS 100. Therefore the traceroute would never show AS on the path at all. So now that we've talked about the importance of origin authentication and attacks against it, let's talk a little bit about session authentication.

Autonomous System Session Authentication



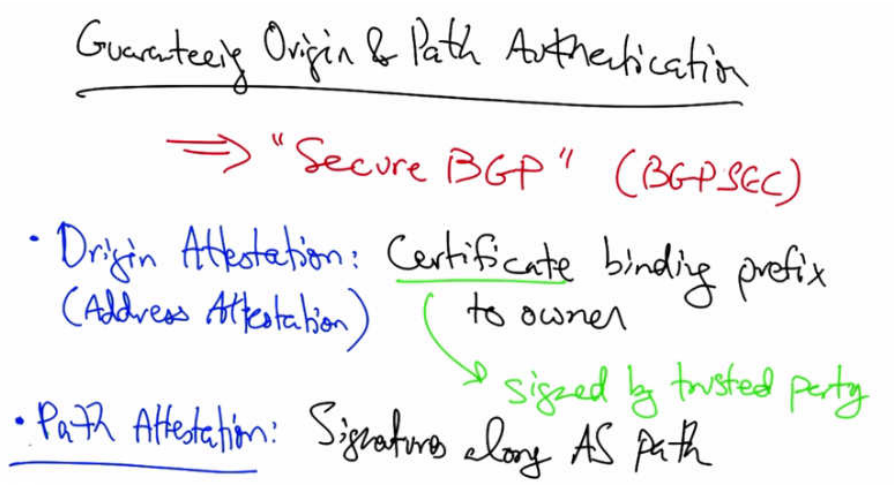
Session Authentication simply attempts to ensure that BGP Routing messages sent between routers between AS's are authentic. Now, this turns out to be a little bit easier than it might appear, because the session between these routers is a TCP session. Therefore, all we have to do

is authenticate this session. The way that this is done, in practice, is done using TCP's MD5 authentication option. In such a setup, every message exchanged on the TCP connection not only contains the message, but also a hash of the message with a shared secret key. Now this key distribution is manual. The operator in AS1 and the operator in AS2, must agree on what the key is, and typically they do that out of band, for example, by calling each other on the phone and manually setting that key in the router configuration. But once that key is set, all messages between this pair of routers is authenticated.



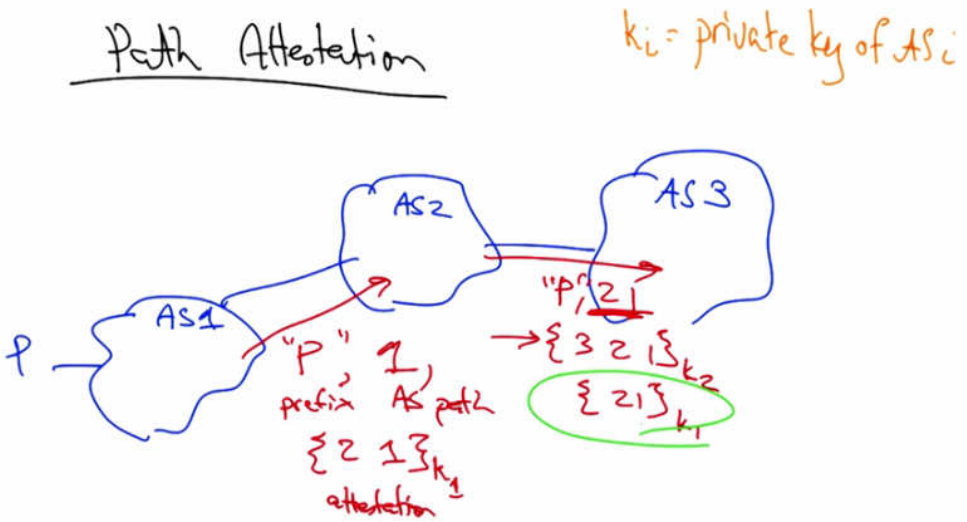
Another way to guarantee session authentication, is to have AS1 transmit packets with a TTL of 255 and have the receiving AS drop any packet that has a TTL less than 254. Because most eBGP sessions are only a single hop and attackers are typically remote, it is not possible for the recipient AS to accept a packet from a remote attacker, because likely that attacker's packets will have a TTL value of less than 254. This defense is aptly called the TTL hack defense for BGP Session Authentication.

Origin and Path Authentication

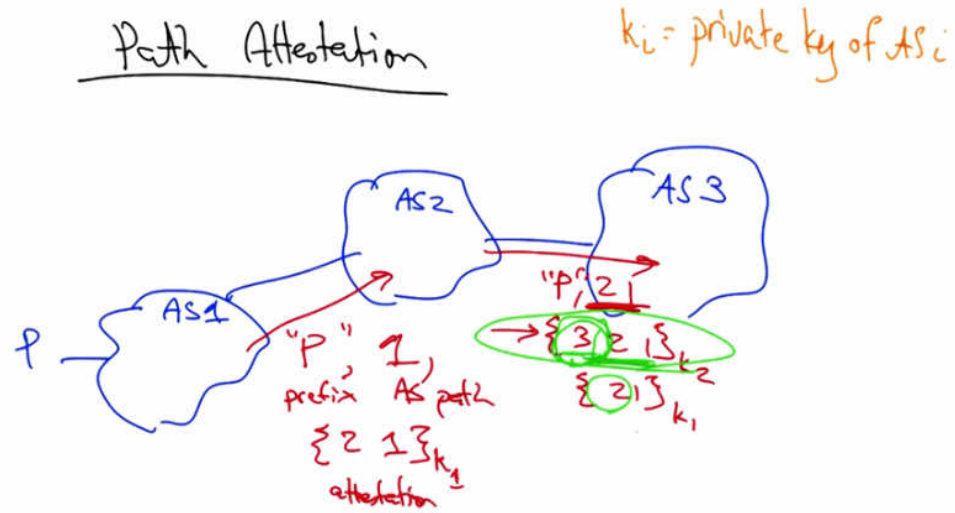


Let's return to the problem of guaranteeing origin and path authentication. To guarantee these properties there is a proposal to modify the existing border gateway protocol to add signatures to various parts of the route advertisement. This proposal is sometimes called Secure BGP or BGPSEC. The proposal has two different parts. The first part is an origin attestation, which is a certificate that binds the IP prefix to the organization that owns that prefix, including the origin AS. This is sometimes also called an address attestation. Now, this certificate must be signed by a trusted party. That trusted party might be, for example, a routing registry or the organization that allocated that prefix to that organization in the first place. The second part of BGPSEC is what's called a path attestation, which are a set of signatures that accompany the AS path as it is advertised from one AS to the next. Let's have a closer look at BGPSEC's path attestation and the types of attacks that it can and cannot prevent.

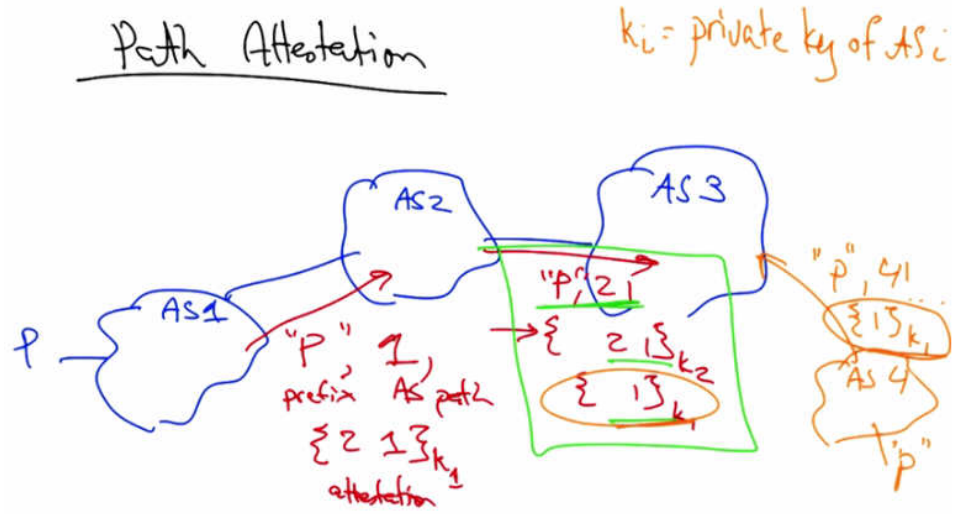
Autonomous System Path Attestation



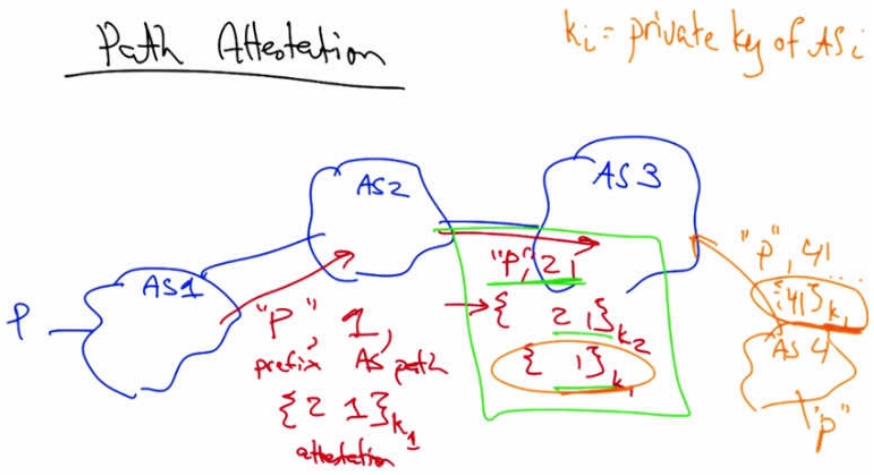
Let's assume that we have a path with three ASes, one, two, and three, and that each AS has a public-private key pair. Let's assume that we have a network with three ASes and that each AS along the path has a public-private key pair. An AS can sign a message or a route with its own private key, and any other AS can check that signature with the AS's public key. So let's suppose that AS1 advertises a route for prefix p. So that route would contain the prefix as well as an address attestation, which we're not showing. Bbut let's look at the path attestation. So, as usual, the BGP announcement would contain the prefix p, and the AS path, which so far is just 1. And, the path attestation, which is actually the path to 1 signed by the private key of AS1. When AS2 re-advertises that route announcement, it of course advertises the new AS path 2 1. It adds its own route attestation, 3 2 1, signed by its own private key, and it also includes the original path attestation signed by AS1. A recipient of a route along this path can thus verify every step of the AS path. AS3 can use the first part of the path attestation to verify that the path in fact, goes from AS2 to AS1, and does not contain any other AS's in between.



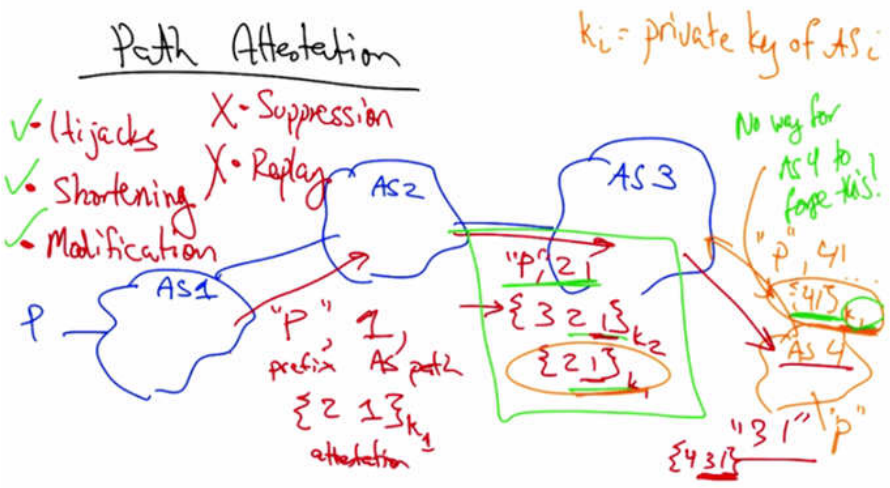
It can use the second part of the path attestation to ensure that the path between it, AS3, and the next hop, is in fact, AS2, and that no other AS's could've inserted themselves on the path between 2 and 3. This is precisely why the AS signs a path attestation with not only its own part of the AS path in the path attestation, but also, the hop of the AS that is intended to receive the BGP route advertisement.



To see the importance of this part of the path attestation, suppose, that these AS's were not there in the path at station. In this case we have a very nice, well-formed BGP route advertisement for a prefix with the AS path suffix 2 1, and we have each segment signed. So an attacker could, in fact, take such an announcement and advertise sub strings of this route advertisement as their own. Thus an attacker, AS4, could claim that it was connected to prefix P via AS1 when in fact no such link existed simply by stealing or replacing the path attestation 1 that's signed by K1.



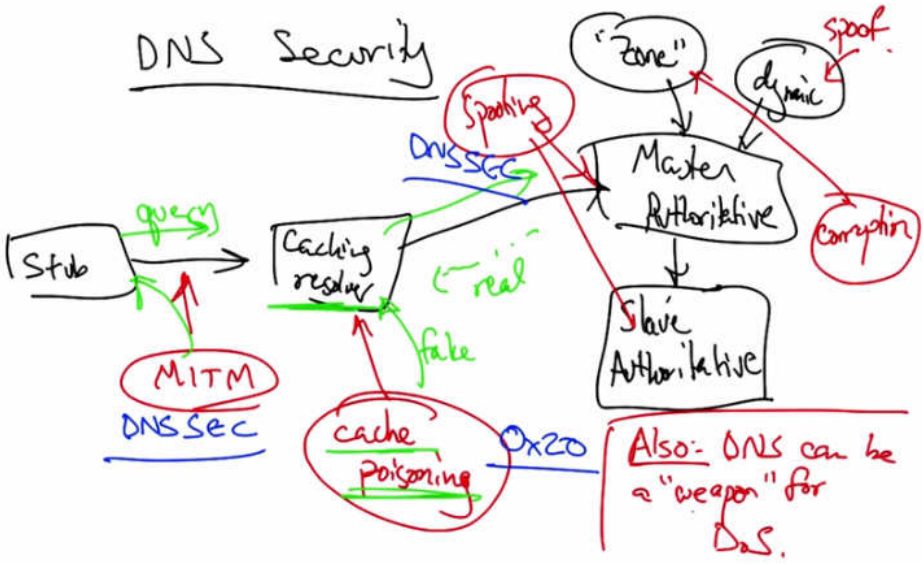
But, note that in reality AS1 never generates this signature. In fact, it generates the signature, 21. Or in this case, it would somehow have to generate the signature 41 signed by AS1's private key, whereas if AS1 only signed a message with its own AS in the message, such a segment or attestation could easily be replayed.



There's actually no way that AS4 Could forge the path attestation for 1, signed by AS1's private key because it doesn't own this private key and AS1 never generated a path attestation with this particular signed path. This is the reason that each AS not only signs a path attestation with its own AS on the AS path, but also the next AS along the path. This particular mode of signing not only prevents the type of hijacking that we explored, but it also prevents path shortening attacks. For example, when AS4 receives the legitimate route to ASP through the path 3 2 1, it would be impossible for the AS to shorten that advertisement to say 3 because it would somehow have to generate a path attestation 4 3 1, signed by its own secret key. However, if it did that, the receiving AS would look for another path attestation with just 3 1 signed by AS3. Yet, such a path attestation would not actually exist. So, these path attestations can prevent against some kinds of hijacks (as we've seen), they can prevent against these path shortening attacks, and they can also prevent against modification of the AS path. However, there are certain attacks that path attestations cannot defend against. So, if an AS fails to advertise a route or a route withdrawal,

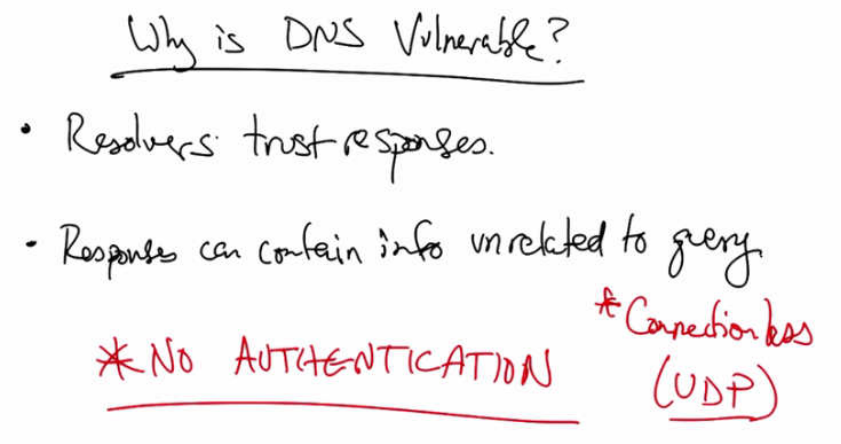
there is no way for the path attestation or BGPSEC to prevent from that kind of attach. Certain types of replay attacks such as a premature re-advertisement of a withdrawn route also cannot be defended against and of course, there is no way to actually guarantee that the data traffic travels along the advertised AS path, which is a significant weakness of BGP that is yet to be solved by any routing protocol.

DNS Security



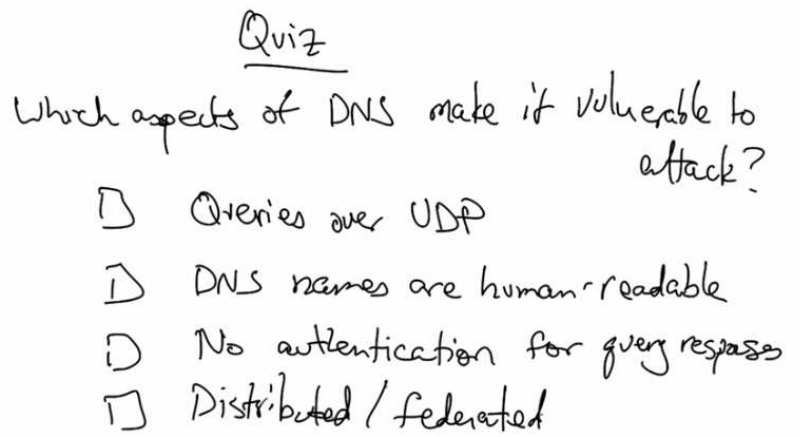
Let's now talk about DNS security. To understand the threats and vulnerabilities of DNS, let's take a look at the DNS architecture. So we have a stub resolver which issues a query to a caching resolver. At this point, we could have a man in the middle attack, or an attacker which observes a query and forges a response. If a query goes further than the local caching resolver, say for example to an authoritative name server, an attacker could try to send a reply back to that caching resolver before the real reply comes back to try to poison, or corrupt, the cache with bogus DNS records for a particular name. This attack is particularly virulent and we will look at a cache poisoning attack in this lecture. Masters and slaves can both be spoofed. Zone files could be corrupted. Updates to the dynamic update system could also be spoofed. We will look at some defenses to cache poisoning, including the OX20 defense, as well as DNSSEC, which can protect against some of these spoofing and man in the middle attacks. In addition to these attacks, we'll look at an attack called DNS reflection where the DNS can be used to mount a large distributed denial of service attack.

Why is DNS Vulnerable



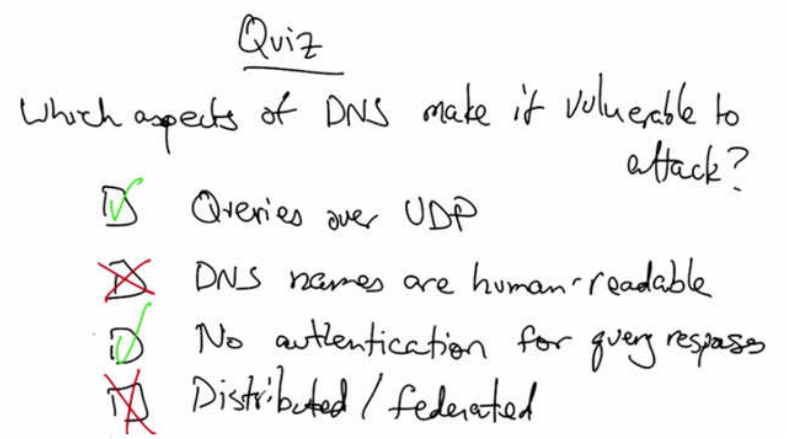
So why is DNS vulnerable in the first place? So the fundamental reason is that the resolvers that issue the DNS query trust the responses that are received after they send out a query regardless of where that response comes from. So sometimes these responses can be forged. When a resolver sends out a query, it typically generates what's called a race condition. And if the attacker replies before the legitimate responder, then the resolver is likely to believe the attacker. DNS responses can also contain additional DNS information that's unrelated to the query. The fundamental problem is that the basic DNS protocols have no means for authenticating responses. This allows an attacker to forge responses after a resolver sends a query. A secondary reason that these types of spoofed replies are possible is that DNS queries are typically connectionless unlike BGP, where routing messages are transmitted over a reliable TCP connection. UDP queries are sent over a connectionless UDP connection. Therefore, a resolver does not have a way of mapping the response that it receives for a query other than the query ID, which can be forged by the attacker. Let's look at how the combination of the lack of authentication and the connectionless nature of a DNS query allows the possibility of cash poisoning.

DNS Vulnerability Quiz



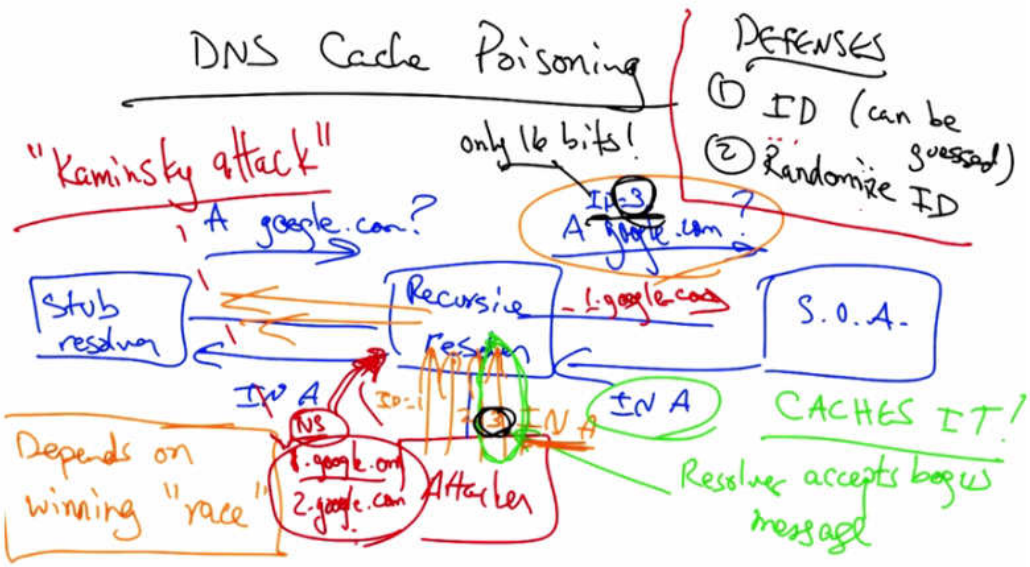
So as a quick quiz, which aspects of DNS make it vulnerable to attack? The fact that queries are sent over UDP? The fact that DNS names are human-readable? The fact that responses to DNS queries are not authenticated? Or, that the DNS is distributed or federated over many organizations?

DNS Vulnerablitiy Quiz Answer



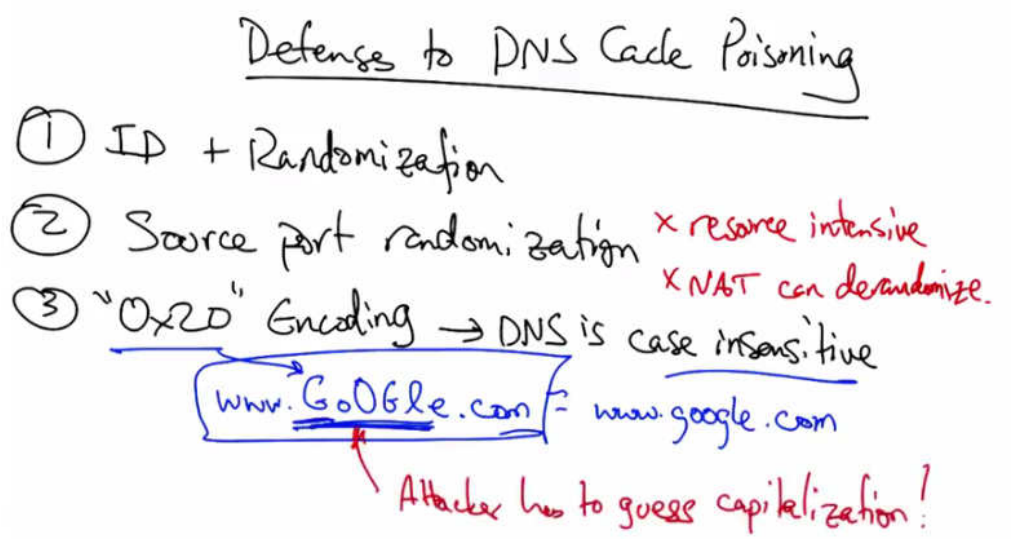
As we discussed, the fact that the queries are sent over a connectionless channel and that there is no way to authenticate the query responses, makes the DNS vulnerable to various kinds of spoofing and cache poisoning attacks. The fact that DNS names are human readable does not make the DNS inherently insecure. Nor does the fact that it's distributed. There are certainly very well understood ways of securing distributed systems and that does not inherently make DNS insecure.

DNS Cache Poisoning



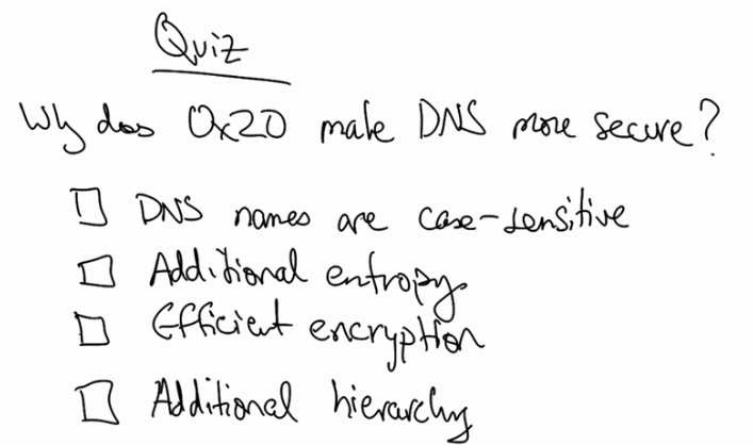
To see how see how a DNS cache poisoning attack works, consider a network where a stub resolver issues a query to its recursive resolver, and the recursive resolver in turn sends that A record query to the start of authority for that domain. Now, in an ideal world, the authoritative name server for that domain would reply with the correct IP address. If an attacker guesses that a recursive resolver might eventually need to issue a query for say, www.google.com, the attacker can simply reply with multiple, specially crafted replies, each with different id's. Although this query has some query id, the attacker doesn't need to see that query because the attacker can simply flood the recursive resolver with a bunch of bogus replies, and one of them, in this case the response with id3, will match. As long as this bogus response reaches the recursive resolver before the legitimate response does, the recursive resolver will accept this bogus message, and worse, it caches the bogus message. And DNS, unfortunately, has no way to expunge a message once it has been cached. So now this recursive resolver will continue to send bogus A record responses for any query for this particular domain name until that entry expires from the cache. Now there's several defenses against DNS cache poisoning, and we've already seen one, which is the query ID. But of course, the query ID can be guessed. The next defense is to randomize the ID. So rather than having a resolver send queries where the ID's increment in sequence, the resolver can pick a random ID. This makes the ID tougher to guess, but still, the query ID is only 16 bits, which still makes it possible for an attacker to flood the recursive resolver with many possible responses. And, it's likely that with relatively few responses, one of these bogus responses will match the ID for the real query. Due to the birthday paradox, the success probability for achieving a collision between the query ID of the query and of the response actually only requires sending hundreds of replies, not a complete 32,000. Due to the birthday paradox, the probability that such an attack will succeed, using only a few hundreds of replies, is relatively close to one. The attacker does not need to send replies with all two to the 16th possible IDs. The success of a DNS cache poisoning attack not only depends on the ability to reply to a query with a correct matching ID, but it also depends on winning this race. That is, the attacker must reply to that query before the legitimate authoritative name server replies. If the bad guy, or the attacker, loses the race, then the attacker has to wait for that correct cached entry to expire before trying again. However, the attacker can generate his own DNS query. For example, he could query one.google.com, two.google.com and so forth. Each one of these bogus queries will generate a new race. And eventually the attacker will win one of these races for an A record query. But who cares? Nobody necessarily cares to own one.google.com, or two.google.com. The attacker really wants to own the entire zone. Well the trick here is that instead of just simply responding with A records in the bogus replies, the attacker can also respond with NS records for the entire zone of google.com. So by creating one of these races using an A record query, and then responding not only with the A record response, but also with the authoritative of the NS record for the entire zone, the attacker can in fact own the entire zone. This idea of generating a stream of A record queries to generate a bunch of races and then stuffing the A record responses for each of these with a bogus authoritative NS record for the entire zone, is what's called the Kaminsky Attack, after Dan Kaminsky, who discovered the attack. The defenses of picking a query ID and randomizing the ID, help, but remember the randomization is only 16 bits, so let's think about other possible defenses.

DNS Cache Poisioning Defense



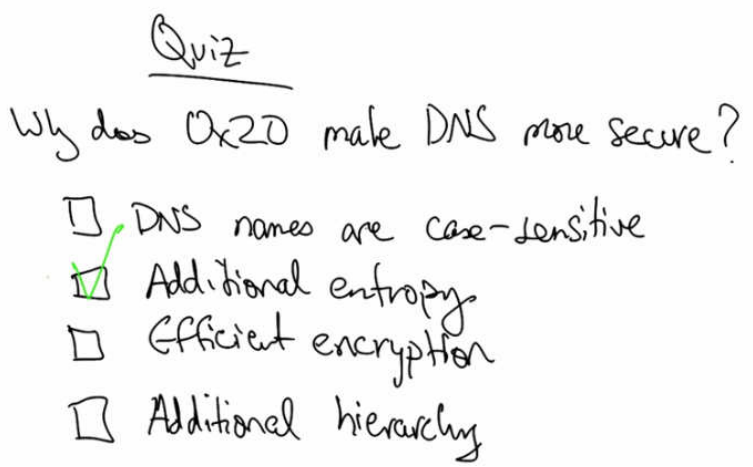
In addition to having a query ID and randomization of that ID, the resolver can randomize the source port on which it sends the query, thereby adding an additional 16 bits of entropy to the ID that's associated with the query. Unfortunately, picking a random source port can be resource intensive and also a network address translator or a NAT, could derandomize the port. Another defense is called the 0x20 or the 0x20 encoding, which is based on the intuition that DNS matching and resolution is entirely case insensitive. So capitalization of individual letters in the domain name do not affect the answer that the resolver will return. This 0x20 bit, or the bit that affects whether a particular character is capitalized or in lower case can also be used to introduce additional entropy. When generating a response to a query such as this one, the query is copied from the DNS query into the response exactly as it was in the query. The mixed pattern of upper and lower case letters thus constitutes a channel. If the resolver and the authoritative server can agree on a shared key, then the resolver and the authoritative are the only ones who know the appropriate pattern of upper and lower case letters for a particular domain name. Because no attacker would know the appropriate combination of upper and lower case letters for a particular domain, it becomes even more difficult for the attacker to inject a bogus reply because not only would the attacker have to guess the ID, but the attacker would also have to guess the capitalization sequence for any particular domain name.

DNS Security Quiz



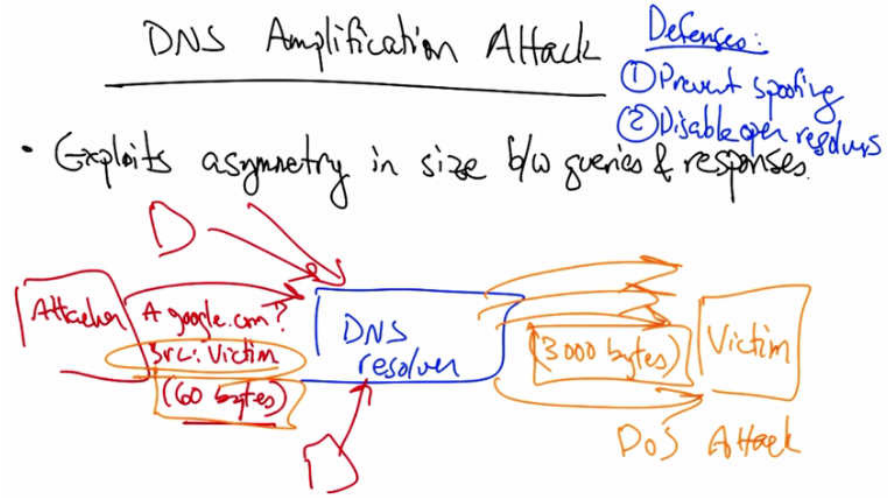
So why does the 0x20 encoding make DNS more secure? Is it because DNS names are case-sensitive? Is it because the encoding adds additional entropy to the query? Is it because the encoding make it easier to encrypt the queries and replies? Or is it because the encoding adds the requirement for an additional layer of hierarchy into the DNS resolution infrastructure?

DNS Security Quiz Answer



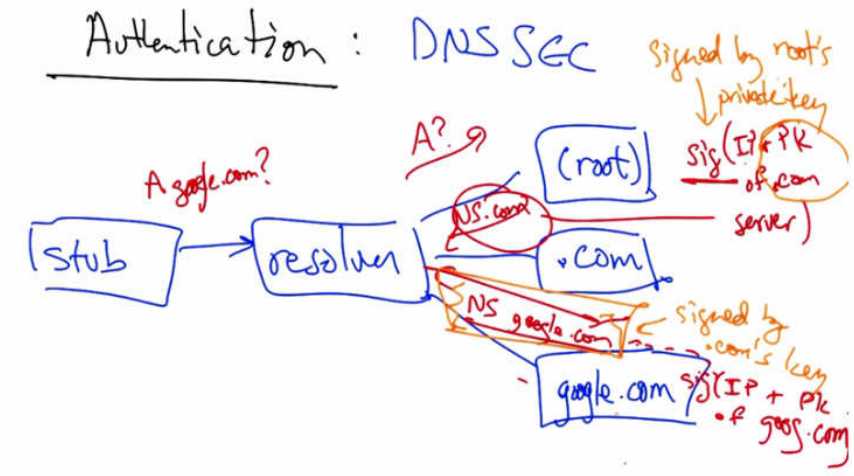
The 0x20 bit encoding adds additional entropy to the queries that a DNS resolver sends by tweaking the capitalization on a DNS name in such a way that only the resolver and the authoritative name server know the particular sequence of upper and lower case letters in the reply.

DNS Amplification Attacks



Let's look at another attack called the DNS amplification attack. This attack exploits the asymmetry in size between DNS queries and their responses. So an attacker might send a DNS query for a particular domain, and that query might only be 60 bytes. In sending the query, however, the attacker might indicate that the source for this query is some victim IP address. Thus, the resolver might send a reply which is nearly two orders of magnitude larger to a victim. So the name of the attack amplification comes from the fact that the query is only 60 bytes and a reply is considerably larger. So, by simply generating a small amount of initial traffic, the attacker can cause the DNS resolver to generate a significantly larger amount of attack traffic. If we start adding other attackers, all of which specify the victim as the source, then all of these giant replies start heading towards the victim, and we have a denial of service attack on the victim. Two possible defenses against this attack are to prevent IP address spoofing in the first place, using, for example, the appropriate filtering rule, or to disable the ability for a DNS resolver to resolve queries from arbitrary locations on the Internet.

DNSSEC DNS Security

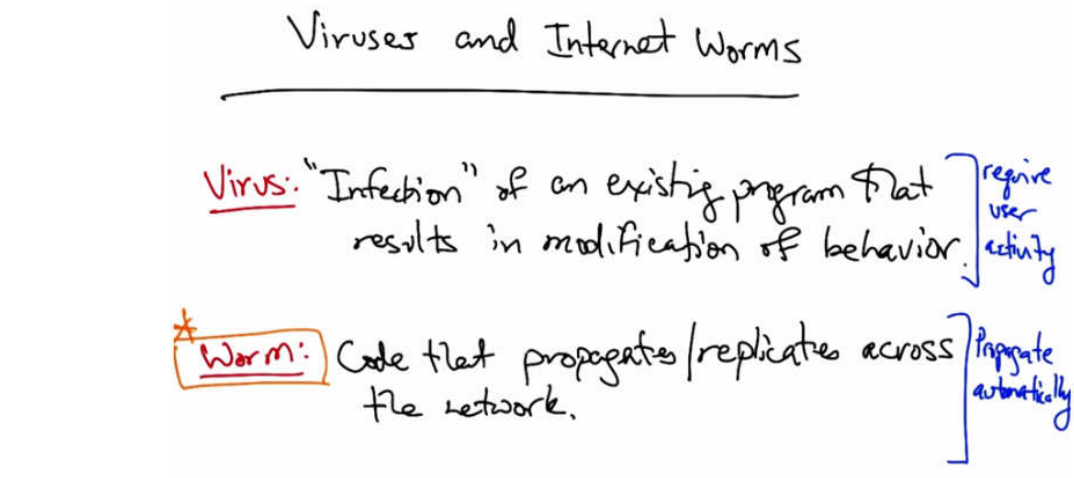


As we discussed, one of the major reason for DNSs vulnerabilities is a lack of authentication. The DNSSEC protocol adds authentication to DNS responses simply by adding signatures to the responses that are returned for each DNS reply. When a stub resolver issues a query, assuming there is no caching, the query is relayed by the recursive resolver to the root name server, which, as we know, sends a referral to .com, but this referral includes the signature by the root of the IP address and the public key of the .com server. As long as this resolver knows the public key corresponding to the route, it can check the signature and it knows then that the referral is to the correct IP address for .com. It also now knows the public key corresponding to the .com server. Thus when the .com server sends the next referral to Google.com, that referral is signed by .com's private key. But the root has told the resolver the public key corresponding to .com, and thus the resolver can check that this referral is not bogus and in fact came from the .com server. Similarly, the .com server will return not only the IP address for Google.com, but also the IP address and public key for the Google.com authoritative name server so that when Google returns its answers, the resolver can check the signatures coming from google.com. In other words, each authoritative name server in the DNS hierarchy returns not only the referral, as it would with regular DNS, but also a signature containing the IP address for that referral, and the public key for the authoritative name server that corresponds to that referral. That public key then allows the resolver to check the signatures at the next lowest level of the hierarchy, until we finally get to the answer.

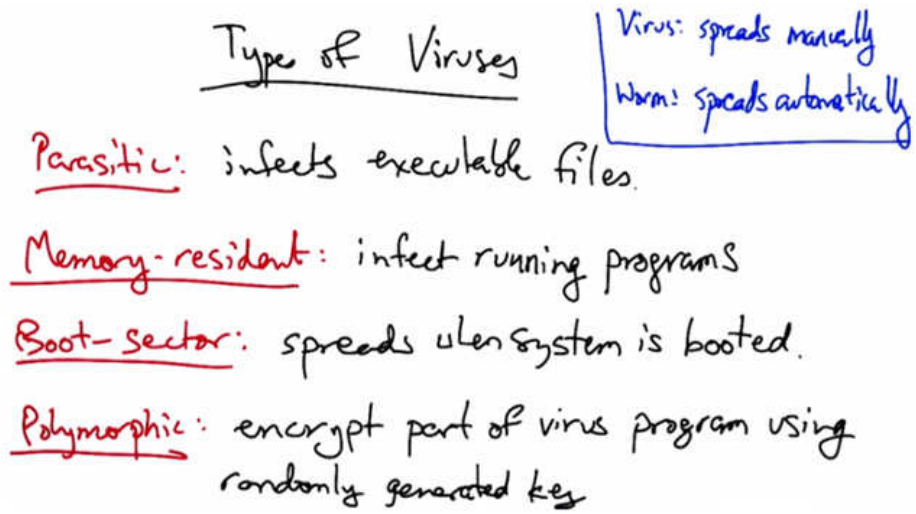
CS 6250: Computer Networking

Lecture 11.1: Internet Worms

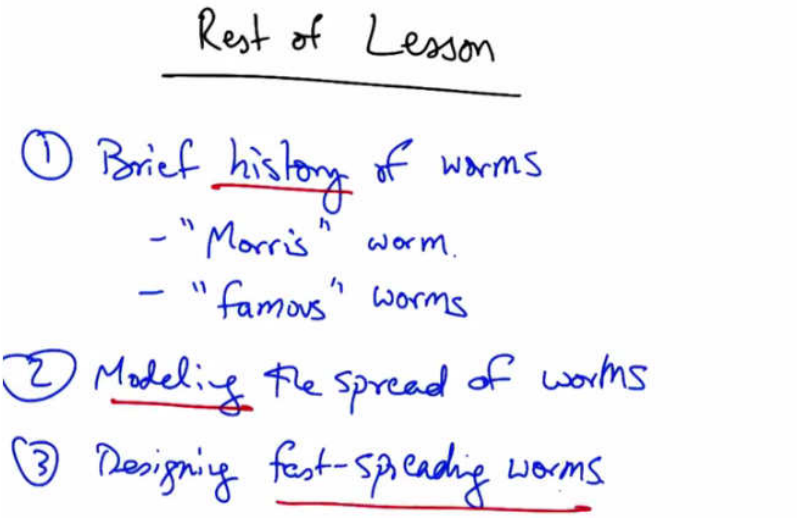
Types of Viruses and Worm Overview



In this lesson we will talk about viruses and internet worms. Let's first define what a virus is, and then define what a worm is. A virus is effectively an infection of an existing program that results in the modification of the original program's behavior. A worm is code that propagates and replicates itself across the network. A worm is usually spread by exploiting flaws in existing programs or open services whereas viruses typically require user action to spread, for example, opening an attachment on an email or running an executable file that a friend gave you on an USB key. Worms propagate automatically. We will focus most of our attention on internet worms.

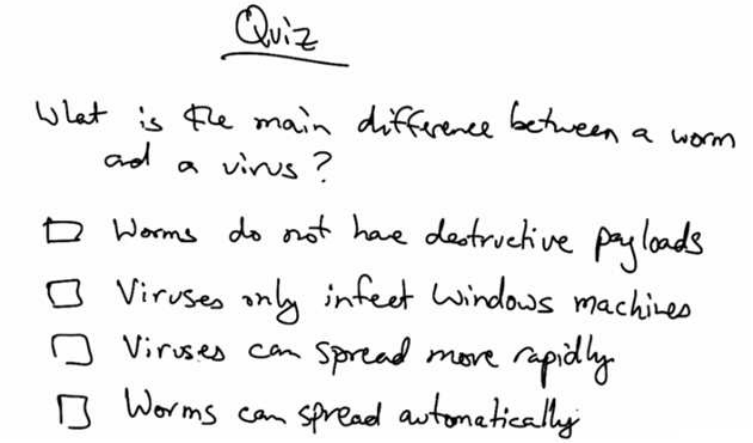


But before we dive into the details of internet worms, let's first talk about the different types of viruses. A parasitic virus typically infects an existing executable file. A memory- resident virus infects running programs. A boot-sector virus spreads whenever the system is booted. A polymorphic virus encrypts part of the virus program using a randomly generated key. So one of the key differences between viruses and worms is that viruses typically spread with manual user intervention. Worms typically spread automatically by scanning for vulnerabilities and infecting vulnerable hosts when those vulnerabilities are discovered. A worm might use any of these techniques to infect a particular host before spreading further.



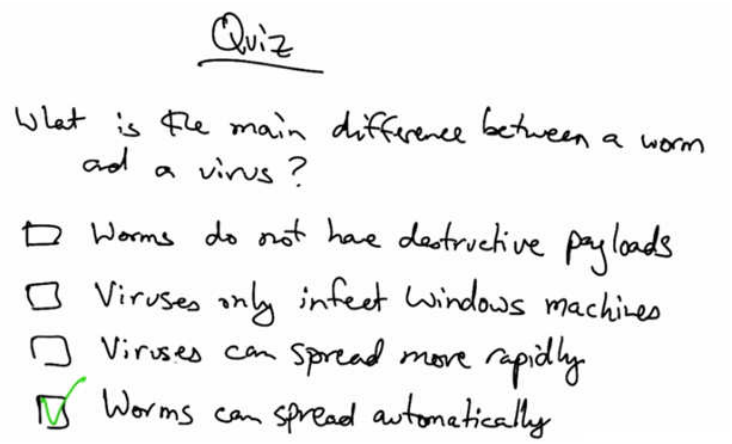
In the rest of the lesson we will first talk about a brief history of internet worms, including the first Internet worm, called the Morris worm, and other famous Internet worms from the early days of Internet worms in the early 2000's, including Code Red and other well-known Internet worms of the time. We'll then talk about how to model the spread of a worm in terms of scanning and infection rates, using analogies from epidemiology. Finally, we'll talk about design techniques for designing super fast-spreading worms, and we'll look at an example of a super fast-spreading worm.

Worm and Virus Quiz



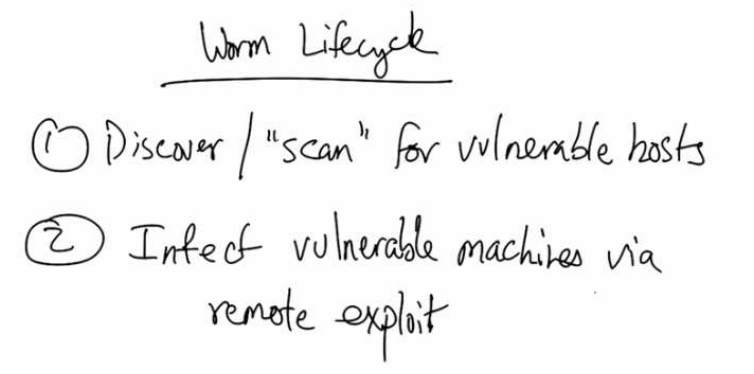
So let's have a quiz to review our knowledge of the difference between worms and viruses. So what's the main difference between a worm and a virus? Is it that worms do not have destructive payloads, whereas viruses do? Is it that viruses only infect Windows machines, whereas worms can infect any kind of machine? Is it that viruses can spread more rapidly than worms? Or is that worms can spread automatically without human intervention, whereas viruses require human intervention to spread?

Worm and Virus Quiz Answer



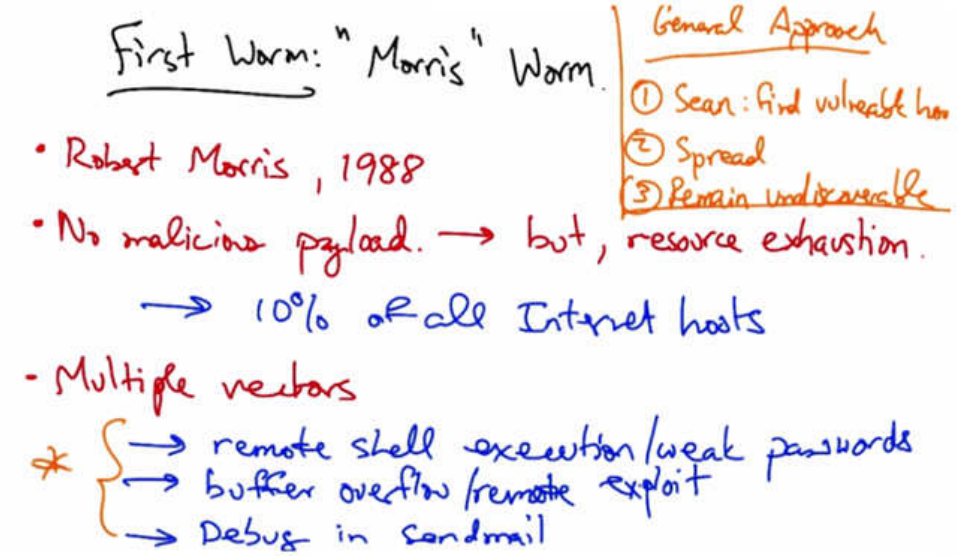
The main difference between worms and viruses is that, worms can spread automatically by scanning for vulnerable hosts and spreading, whereas viruses typically require user intervention to spread such as clicking on an executable file in an email attachment or installing a particular program from a USB stick.

Internet Worm Lifecycle



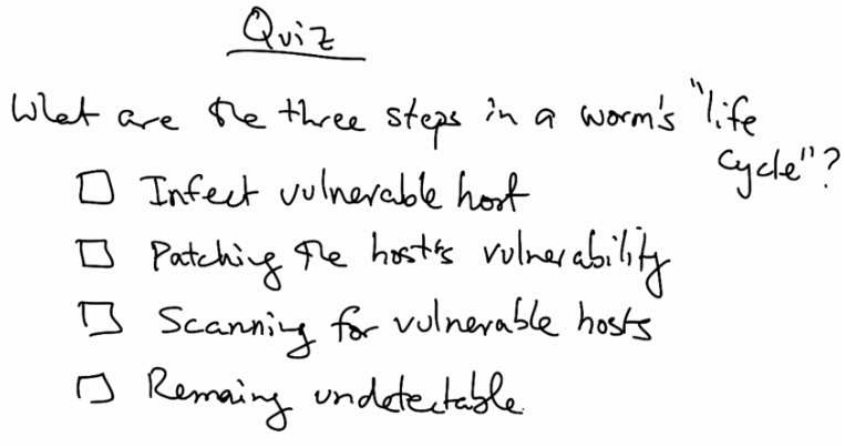
A worm's spread on the internet has the following life cycle. First, the infected machine might scan other machines on the internet to discover vulnerable hosts and subsequently infect the vulnerable machines that it discovers via remote exploit. Let's take a look at a couple of well known early worms and how they spread, as well as how one might design a super fast spreading worm.

First Worm



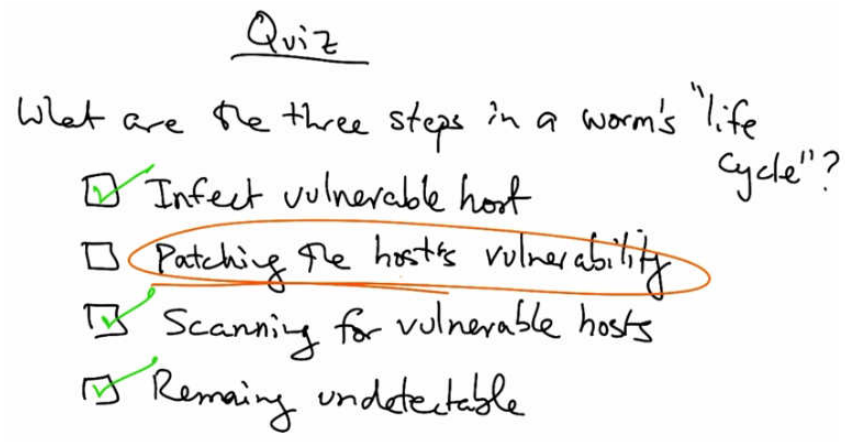
The first worm was designed by Robert Morris, Jr. in 1988. The worm itself had no malicious payload, but it ended up bogging down the machines that it infected by spawning new processes uncontrollably and exhausting resources. And at the time it was released, it affected ten percent of all Internet hosts. It spread through three different propagation vectors. The worm tried to crack passwords using a small dictionary and a publicly readable password file, and also targeted hosts that were already listed in a trusted host file on the machine that was already infected. This ability to perform remote execution was one way that the worm was allowed to spread. The second way that it spread was in a buffer overflow vulnerability in the finger demon. This was a standard buffer overflow exploit. And if you don't know about buffer overflows, I would urge you to take a computer security course, but essentially, this is a very common attack that makes remote exploits possible, effectively resulting in the ability to run arbitrary code at the root level privilege. The third way that worm spread, was via the debug command in send mail, which is a mail sending service. In early send mail versions, it was possible to execute a command on a remote machine by sending an SMTP message. The worm used this capability to spread automatically. A key theme that we'll see In the design of other worms, is this use of multiple vectors. Now any particular worm may end up using a different set of vectors depending on the remote vulnerabilities that it's trying to exploit. But the idea that any worm should be able to exploit multiple weaknesses in a system gives it more ways to spread and often also speeds up the propagation of the worm. This worm design also followed the following general approach, which we see showing up over and over again in worm designs. First, the worm needs to scan other hosts to find potentially vulnerable hosts. In the second step, it needs to spread by infecting other vulnerable hosts. And in the third step it needs to remain undiscoverable and undiscovered so that it can continue to operate and spread without being removed from systems.

Worm Lifecycle Quiz



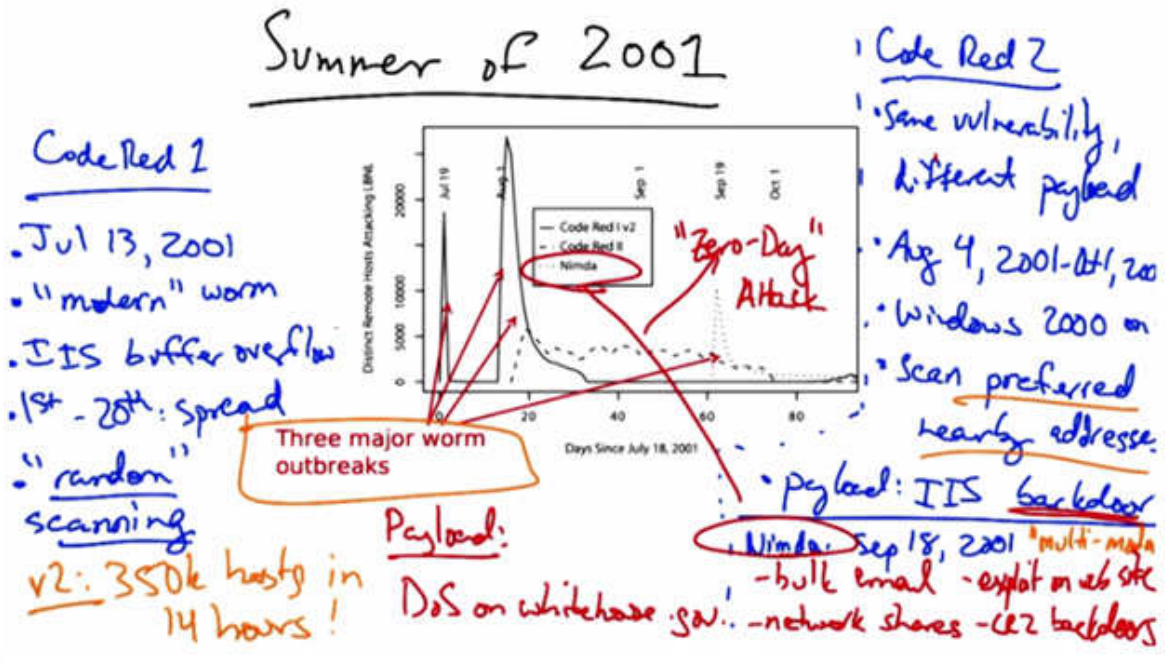
So to review, what are the three steps in a worm's life cycle? Please check three of the following options. Infecting a vulnerable host, patching the hosts vulnerability after infection, scanning for other vulnerable hosts to infect, or remaining undetectable.

Worm Lifecycle Quiz Answer



An Internet worm first scans for vulnerable hosts, then infects them, and finally, typically takes steps to remain undetectable. A worm does not necessarily need to patch the host vulnerability, although some Internet worms have been known to do so to prevent other worms from subsequently infecting the machine and interfering with the original worm infection. For example, if a worm was intending to spread to construct a botnet that launched a particular attack or was being used by a botmaster for a particular attack, then whoever had commandeered the machine probably wouldn't want other attackers to come in behind him and also infect the machine and interfere with the planned attacks.

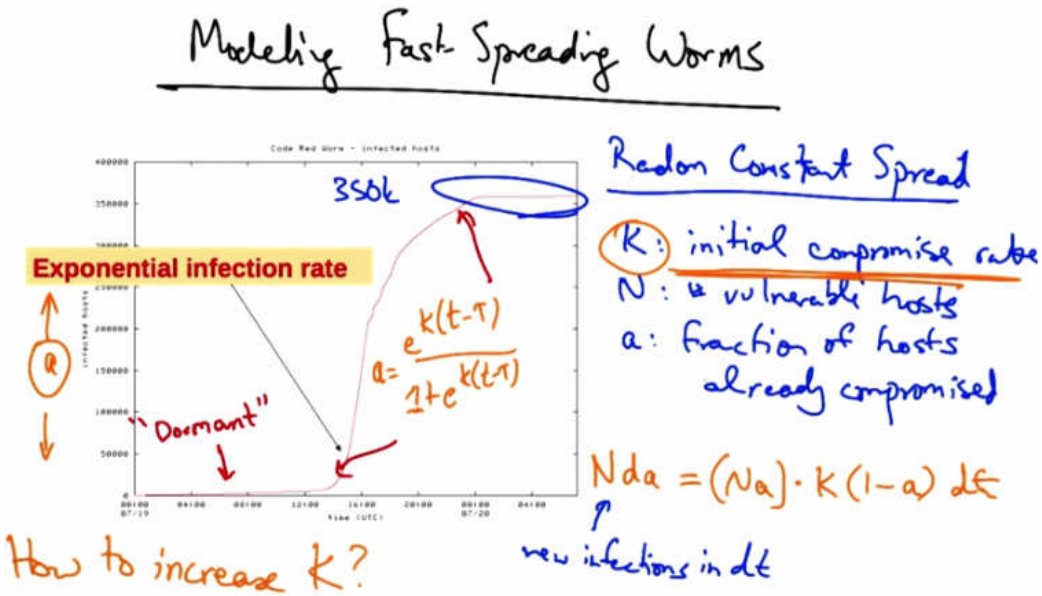
Worm Outbreaks in Detail



The summer of 2001 essentially saw a new era in internet security with three major worm outbreaks. These three major worms were Code Red 1, version two, Code Red 2, and Nimda. Let's take a quick look at each of these worms. Code Red 1 was released on July 13th, 2001, and was the first modern worm. It exploited a buffer overflow in Microsoft's IIS server. From the first through the twentieth of each month it would spread by finding new targets using a random scan of IP address space, it would spawn 99 new threads, which generated IP addresses at random, and then looked for vulnerable instances of IIS. Now version 2 of Code Red 1 was actually released six days later and fixed that random scanning bug so that each instance of the worm scanned a different part of IP address space. After the scanning bug was fixed, the worm was able to compromise 350,000 vulnerable hosts in a matter of only fourteen hours. By most estimates that was the complete set of hosts running the vulnerable version of IIS on the entire internet. The payload of this worm was to mount a denial of service attack on whitehouse.gov. But a bug in the coding caused the worm to die on the 20th of each month. If the victim's clock was wrong, however, the worm would actually resurrect itself on the first. Fortunately in this case, the payload which launched the denial of service attack on whitehouse.gov actually was launched at a particular IP address, not at the domain name. So the operators of the website needed only to move the web server to another IP address to defend against the denial of service attack. A better worm design would have been much more catastrophic. Code Red 2 exploited the same vulnerability but had a completely different payload. It was released on August 4th, 2001, and was called Code Red 2 mainly because of a comment in the code. The worm actually only spread on Windows . The scan actually preferred nearby addresses. It would choose addresses from the same /8 with probability one half from the same /16 with probability three eighths, and randomly from the entire internet with the remaining one eighth probability. The reason for preferring nearby IP addresses is that if there was one vulnerable host on the network, there was likely to be more because the same administrator that failed to patch the compromised

machine might have other machines on the same network that were also vulnerable. This notion of preferential scanning can speed up infections in some cases by increasing the probability that scanning will find another vulnerable host. The payload of this worm was an IIS backdoor, and the worm was completely dead, by design, by October 1, 2001. Nimba was released on September 18, 2001, and was interesting mostly because it spread using multiple propagation vectors. It was effectively multi-model. So in addition to using the same IIS vulnerability as Code Red 1 and Code Red 2, there were some additional vectors that it used. It could spread by bulk email as an attachment. It copied itself across open network shares. It installed an exploit code on webpages on the corresponding web server running on the machine, so that any browser that visited the webpage for that server would become infected itself and it would scan for the Code Red 2 backdoors that that worm had installed. The interesting thing about the multi-modal nature of the Nimda worm is that signature based defenses don't necessarily help because of the many ways that it could spread, for example, by email or via a website exploit. Nimda actually needs firewalls. Most of the firewalls pass the email carrying Nimda completely untouched, using brand new infection with an unknown signature, and those scanners couldn't detect it. This was the first instance of a worm that exploited what we would call a zero day attack which is when a worm first appears in the wild and the signature of the worm is not extracted until minutes, or hours later. Zero day attacks are particularly virulent because the worm can spread extremely quickly before any type of signature-based antivirus has a chance to catch up and prevent the infections in the first place.

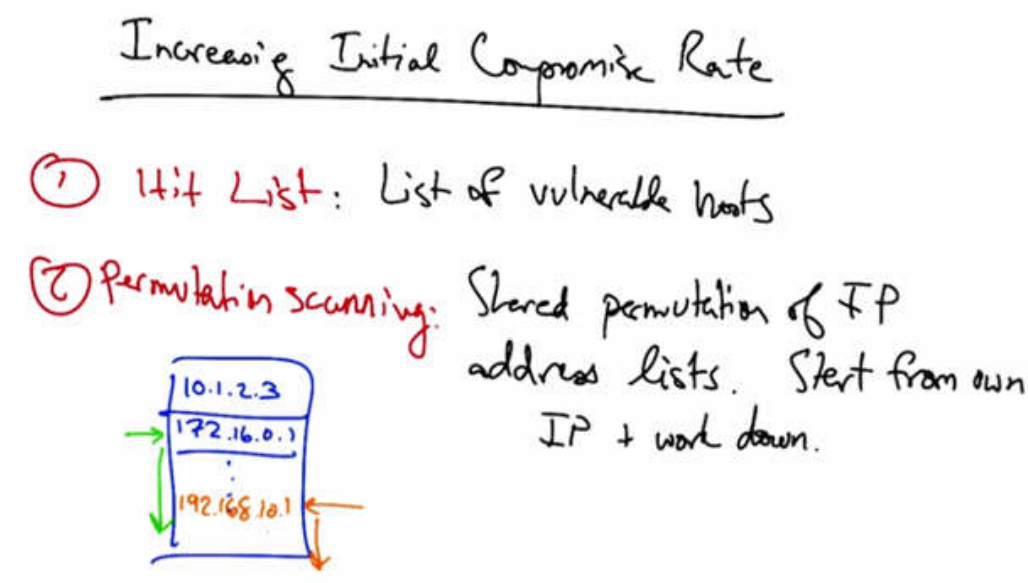
Modeling Fast-Spreading Worms



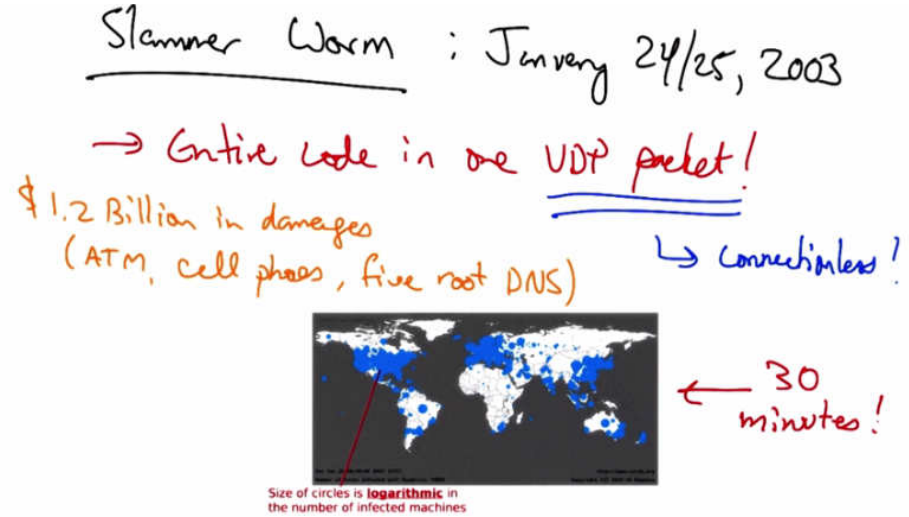
Here is a plot showing infection rate of the Code 1 Version Two Worm which ultimately infected 350,000 vulnerable hosts. Note the shape of this curve. The worm is effectively dormant or spreading extremely slowly for quite a period of time. And then there's a inflection point at which point the infection rate becomes exponential. At some point then, infections slow and the infection rate ultimately plateaus, presumably after all of the infected hosts have been found. We can actually model the spread of these worms using the random constant spread model. If 'K is

the initial compromised rate, 'N is the number of vulnerable hosts, and 'a' is the fraction of hosts already compromised, we can now express the number of hosts infected at a particular time increment in terms of the machines already infected and the rate at which uninfected machines become compromised. So if 'Nda' is the number of newly infected machines in dt, we can express that in terms of the number of machines already infected, which is 'N' times 'a'. So these are the host already capable of doing more scanning, and now 'K' times 1 minus 'a' is the rate at which uninfected machines become compromised in a particular time interval dt. If we solve for 'a', the fraction of hosts compromised, which is effectively the y-axis of this graph, we get the following. You get an exponential curve that is exponential where the growth rate depends only on K, or the initial compromise rate. This is very interesting because it tells us that if we want to design a very fast spreading worm, then we should design a worm such that the initial compromise rate is as high as possible. So how do we increase K? Or how do we increase that initial compromise rate?

Increasing Compromise Rate

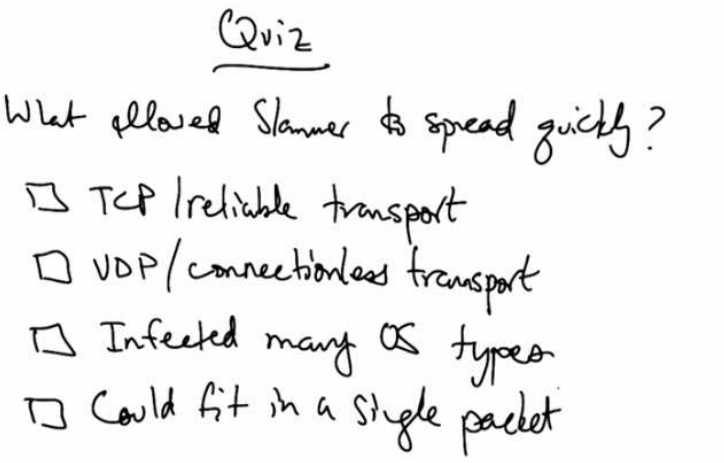


One possibility for increasing the initial compromise rate, or designing a very fast spreading worm, is to create a hit list, or a list of vulnerable hosts ahead of time. That curve we just saw shows that the time to infect the first 10,000 hosts dominates infection time. So if we start by performing stealthy scans or some reconnaissance to construct a list of vulnerable hosts before we start spreading, then we can get rid of that initial flat part of the curve where the worm is effectively dormant. The second approach is to use something called permutation scanning where every compromised host has a shared permutation of an IP address list to scan for vulnerabilities. Now if this list is randomly permuted and a particular host starts scanning from its own IP address in the list and works down, then different affected hosts will start scanning from different parts of this list ensuring that compromised hosts don't duplicate each other's work.



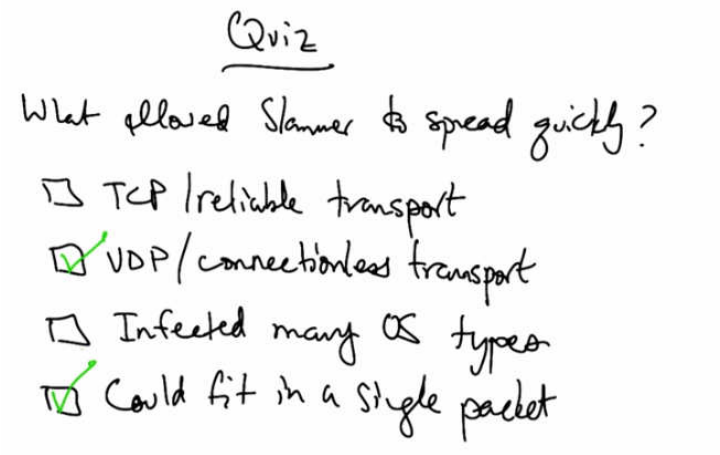
One worm that exploited these techniques to spread particularly quickly was the Slammer worm, which spread in January of 2003, exploiting a buffer overflow in Microsoft's SQL server. In addition to using fast scanning techniques, the entire slammer code fit in a single, small UDP packet. The UDP packet contained the worm binary, followed by an overflow pointer back to itself. It was a classic buffer overflow combined with random scanning. Once the control is passed to the worm code, it randomly generated IP addresses and attempted to send a copy of itself to Port 1434 on other hosts. One brilliant aspect of the slammer worm is that because it was spread via a single UDP packet, it was connectionless, meaning that it could spread and was no longer limited by the latency of network round trip time, but only by the bandwidth of the network. The worm caused $1.2 billion dollars in damage and temporarily knocked out many elements of critical infrastructure including Bank of America's ATM network, an entire cell phone network in South Korea, and five route DNS servers, as well as Continental Airlines' ticket processing software. The worm actually did not have a malicious payload, but the bandwidth exhaustion on the network caused resource exhaustion on the infected machines. Here's a picture of the hosts around the world that Slammer infected. This damage was inflicted in just thirty minutes, due to the very lightweight nature in which Slammer spread.

Slammer Worm Quiz



So as a quick quiz, what allowed the Slammer worm to spread so quickly? Was it that TCPs reliable transport ensured a clean copy of the worm payload would spread to different vulnerable hosts? Was it that the worm spread by UDP, thereby enabling itself to spread with limited network overhead? Was it that it could infect many different types of operating systems including Linux and Mac OS? Or, was it that it could fit in a single packet?

Slammer Worm Quiz Answer

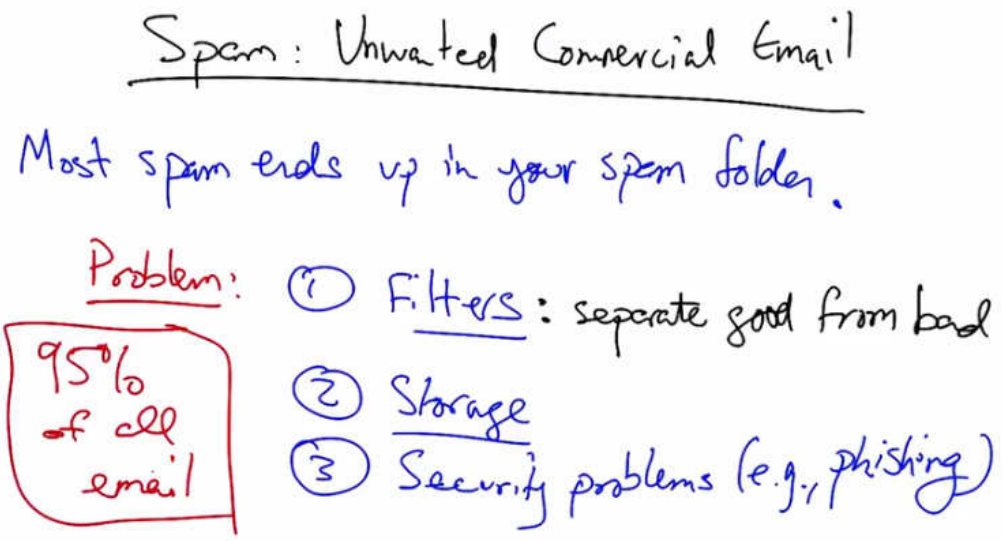


Slammer was able to spread quickly because it spread via connectionless transport, or UDP, and because it could fit in a single packet.

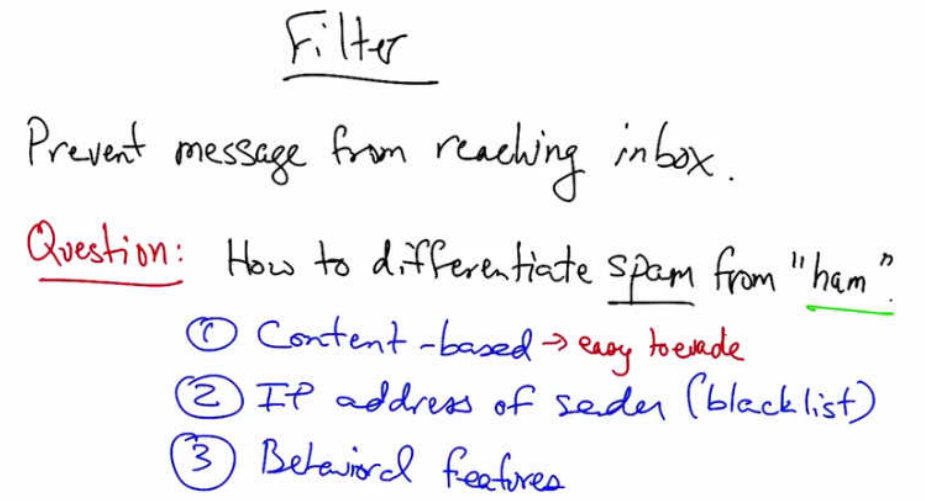
CS 6250: Computer Networking

Lecture 11.2: Spam

Spam

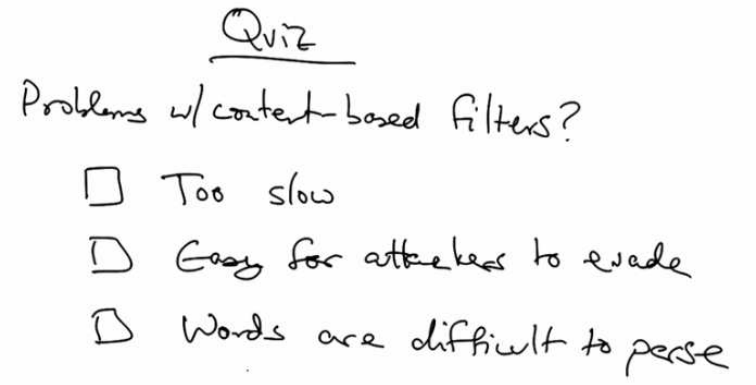


Okay, in this lesson, we will talk about spam or unwanted commercial email. Now, you might not think that you receive a lot of spam, but the fact of the matter is that most of it goes to your spam folder. So one might think, what's the problem? Well, in fact, spam remains a scourge for network operators. In particular, someone has to design the filters that separate the good traffic from the bad traffic. Additionally, even if email is classified as spam, if it's accepted for delivery, the Internet's mail protocols dictate that the server has to keep the mail, because it's told the receiver that it has accepted the mail. This creates the potential for spam to consume a significant amount of storage space on email servers. Finally, spam can create security problems for users who receive spam emails. If the spam messages contain a payload that could be harmful, such as malware or a phishing attack, or an attempt to steal a user's private or sensitive information, such as a password. Now even though you don't see the mail because of these filters, something like 95% of all email traffic is spam. Some reports from the Anti-Phishing Working Group suggests that something like 1 in every 87 emails was a phishing attack. And there's something like 50,000 unique fishing attacks in a month.



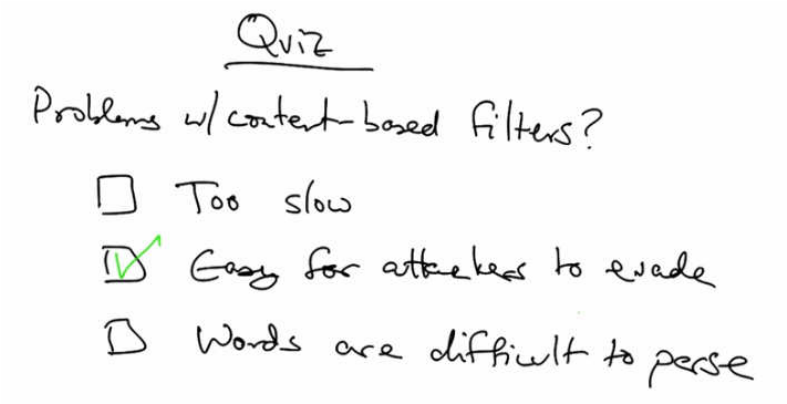
A common approach for getting rid of spam messages is to filter. In other words, prevent the message from reaching the user's inbox in the first place. Now this begs the question of how to differentiate spam, or the bad messages, from ham, or the legitimate messages. There are three different ways to construct filters. One is content-based. In other words, you can look at what's being said in the mail. For example, if the mail contains particular words, such as Viagra or Rolex, a content-based filter might pick up on those terms and decide to filter the mail. Second, a filter might make a decision about whether an email message is spam or ham based on the IP address of the sender. This method is often called blacklisting. Third, we can construct filters based on behavioral features, or how the mail is sent. So, for example, if the mail is sent at a particular time of day, or if it's sent in a batch of emails that are all roughly the same size, then we may be able to figure out that a message is likely spam simply based on the sender's sending behavior. Now each of these approaches are complimentary, but content-based filtering and IP-based filtering each have problems. Content-based filters are relatively easy for attackers to evade. A recent large commercial mail operator recently told me that he saw something like 80,000 different spellings of Viagra. But additionally, messages can be carried not only in text, but in images, Excel spreadsheets, or even MP3s or movies. Therefore, spammers can easily alter the features of an email's content and adjust those features and change them to evade content-based filters. On the flip side, those maintaining the filters suffer a relatively high cost, because the filters must be continually updated as content changes and the means of carrying the content becomes more sophisticated.

Content-based Email Filter Quiz



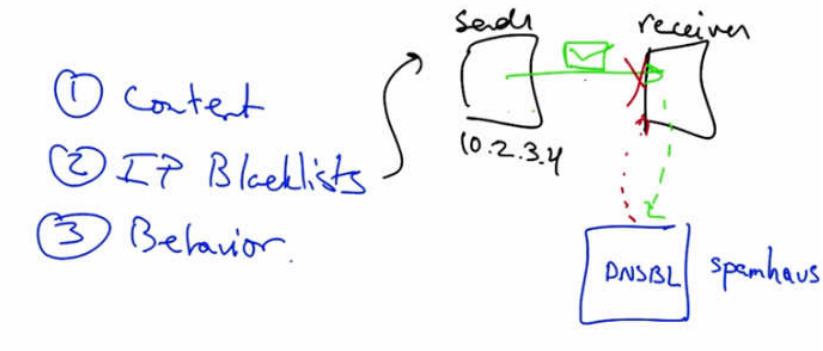
So, as a quick quiz, what are some problems with content-based email filters? Are they too slow? Are they easy for attackers to evade? Or are words in texts of emails difficult to parse? In this case, please choose the single best answer.

Content-based Email Filter Quiz Answer

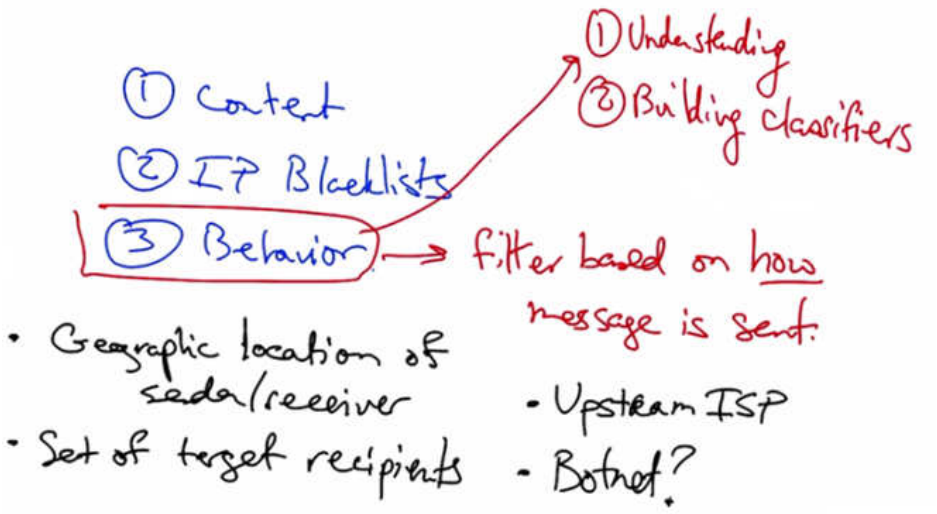


As we discussed, content-based filters are easy for attackers to evade because they can very easily change the content of the message that is carrying the spam that they wish to deliver. They can embed their message in things like images, mp3's, Excel spreadsheets, and so forth, making it relatively difficult for the filter maintainers to keep up.

IP Blacklisting

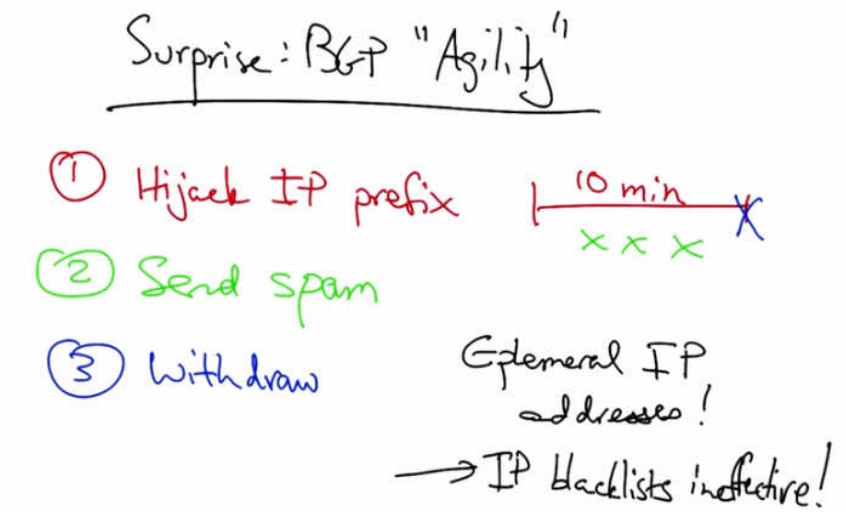


So we've talked about problems with content-based filtering. What about IP blacklists? Well, first, the way an IP blacklists works is that when a sender sends an email to the receiver, the receiver sends a query for that IP address to a blacklist or a DNS-based blacklist sometimes called a DNSBL such as spamhaus. Depending on whether or not that IP address appears in the blacklist, the receiver can then decide to accept the message, or if the IP address turns out to be on the blacklist, the receiver can decide to terminate the connection and not even accept the mail in the first place, thereby saving the operator the trouble of even having to store the message.

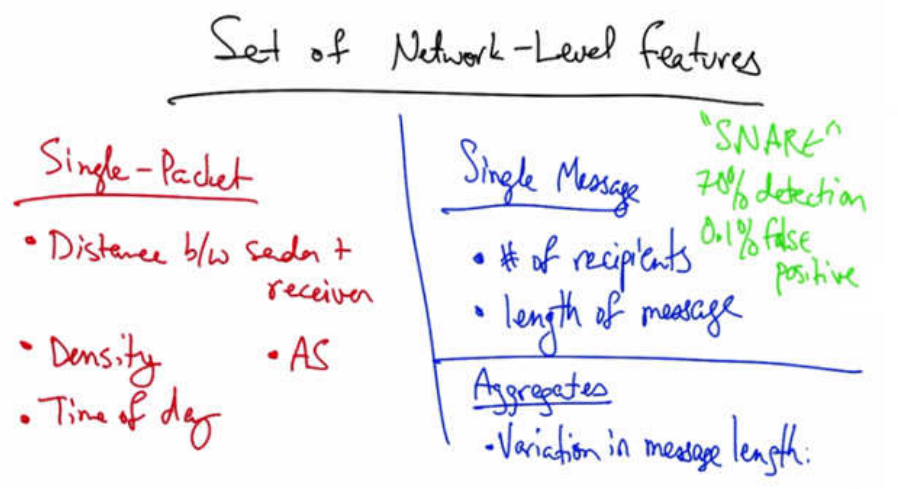


The third approach is to filter a message on how it is sent. In particular, we can look at such features as the geographic locations of the sender and receiver, the set of target recipients, the sender's upstream ISP, or our inference as to whether the sender is a member of a botnet or a network of comprised hosts that are doing the bidding of some command and control server. Now the challenges of building a filter around this notion is first, understanding network level behavior and second, building classifiers using network level features to execute the filtering.

Spam Blacklisting (cont)



A surprising finding from our earlier work is that spammers can perform behavior on the network that is extremely uncanny and unlikely to be performed by a legitimate network user. For example, what we saw is that the spammer could hijack an IP prefix for a very short period of time, such as 10 minutes, send the spam or potentially multiple spam messages from IP addresses inside that IP prefix, and at the end of the attack, withdraw the prefix. This allows attackers to use ephemeral IP addresses, essentially rendering IP blacklists ineffective. In fact, we saw on any given day about 10% of the email senders are from previously unseen IP addresses. This ephemerality or transience of the IP addresses of the spam senders makes it particularly difficult to maintain a blacklist.



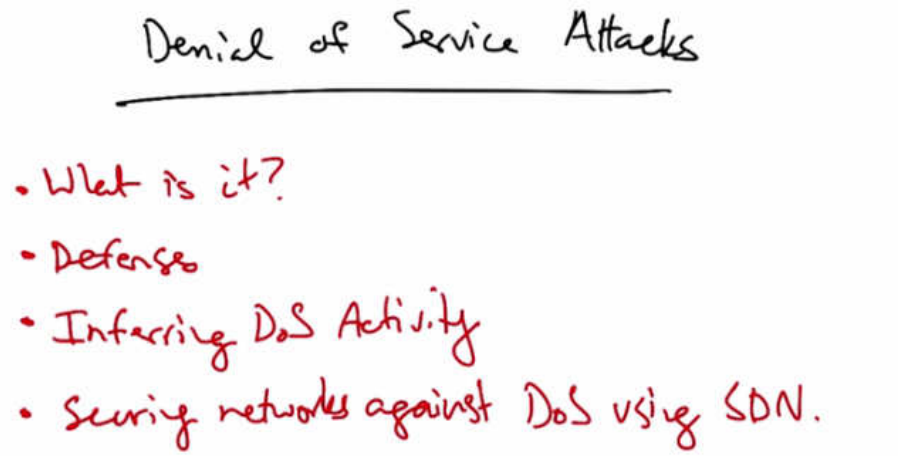
In fact, we've found many single-packet features that tended to work well. In other words, features that a receiver could make a decision on just based on the first packet that a sender sends. Such single-packet features include the distance between the sender and the receiver, the density in IP space in terms of how many other mail senders are nearby, and the local time of day at the sender. Other features, such as the AS of the sender's IP, also worked well. If we're willing

to look beyond a single packet and look at a single message, the number of recipients, and the length of the message also prove to be effective in distinguishing spammers from legitimate senders. Finally, we can look at aggregates. For example, if we're willing to look at a group of email messages, we can see how message length varies over time or across a group of different messages. Putting these features together in a system called SNARE, or Spatiotemporal Network Level Automated Reputation Engine, achieved a 70% detection rate for a false positive rate of about one-tenth of 1%. This level of accuracy is good enough to be used in practice. It provides comparable performance to state of the art IP-based blacklists such as spamhaus. But it only uses network-level features, thus making it less susceptible to the ephemeral nature of IP-based blacklisting.

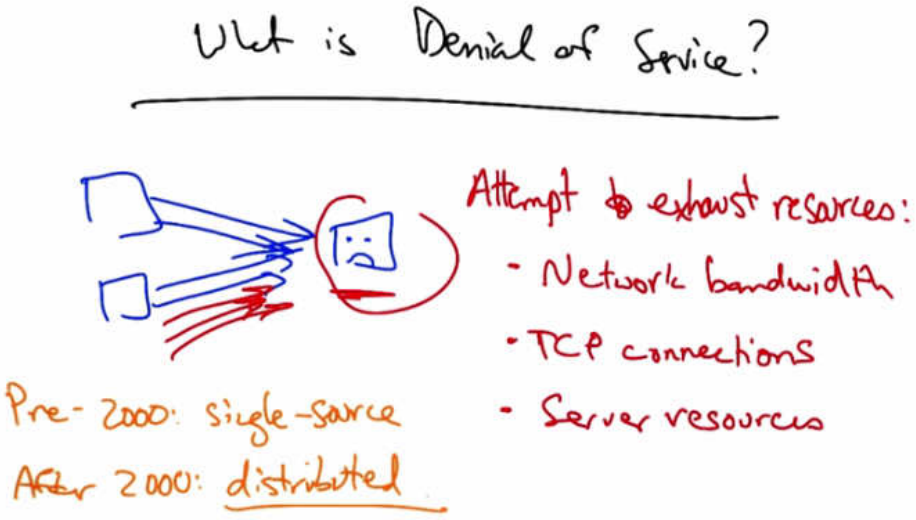
CS 6250: Computer Networking

Lecture 11.3: Denial Of Service Attacks

Denial of Service Attacks Overview



So in this lesson, we will talk about Denial of service attacks and defenses. We'll talk about what denial of service attacks are and various defenses. We'll talk about how to infer denial of service activity, and we'll talk about how to secure networks against denial of service attacks using Software Defined Networking.

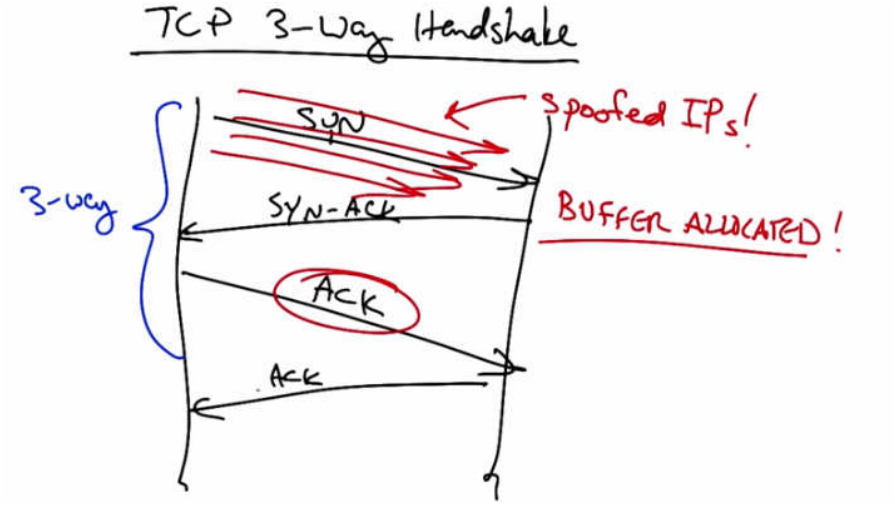


So what is denial of service? Denial of service is simply an attack that attempts to exhaust various resources. One resource that a Denial of Service attack might exhaust is network bandwidth. Another is TCP connections. For example, a host might only have a limited number of TCP connections that it can open to various clients, or the Denial of Service attack might attempt to exhaust various server resources. For example, this victim might be a web server running complicated scripts to render web pages, and if the web server suddenly becomes the target of a bunch of bogus requests, the server may spend a lot of resources rendering pages for requests that are not legitimate. Before 2000, these Denial of Service attacks were typically single source. After 2000, with the rise of internet worms as we saw in an earlier lesson, these attacks could become distributed, effectively being launched from many attackers.

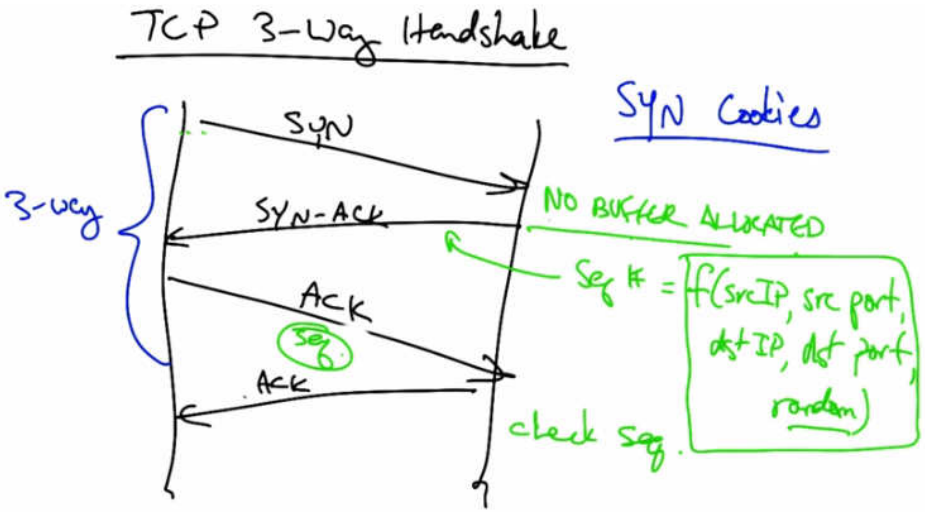
Let's talk about three different types of defenses against denial of service attacks. First we have something called ingress filtering. Then we have something called URPF, or reverse path filtering checks. And then in the case of an attack on TCP connection resources, we can use something called TCP syn cookies to defend against Denial of Service. Let's suppose that we have a stub autonomous system whose IP prefix was 204.69.207.0/24. Now this is a stub network that has no other networks connected to it and this is the only IP address space that this network owns. Then, the router that is immediately upstream of that internet service provider can simply drop all traffic for which the source IP address is not in the IP address range of that particular network. So this is foolproof and it works at the edges of the internet where it's very easy to determine the IP address range that's owned by a downstream stub autonomous system. Unfortunately it doesn't work well in the core, where a particular router might have a lot of difficulty determining whether packets from a particular source IP address could be allowed on a particular incoming interface. So the solution that operators try to use in the core is to use the routing tables to determine whether a packet could feasibly arrive on a particular incoming interface. So if a router had a routing table that said all packets for 10. 0.1.0/24, should be sent via interface one, and all packets destined for 10.0.18.0/24 should be sent via interface two, then URPF says if we see a packet for/with a particular source IP address on an incoming interface that is different than where we would have sent the packet in the reverse direction, then we should go ahead and drop this packet. So the benefits of URPF is that it's automatic, but the

drawbacks are that it requires symmetric routing. And we know from earlier lessons that routing in the internet is often asymmetric. Therefore in any situation where asymmetric routing is a possibility, it is not possible or reasonable to use URPF. So we've talked about ingress filtering and URPF checks, and let's now talk about the use of Syn cookies to defend against TPC based denial of service attacks.

TCP 3-Way Handshake Review

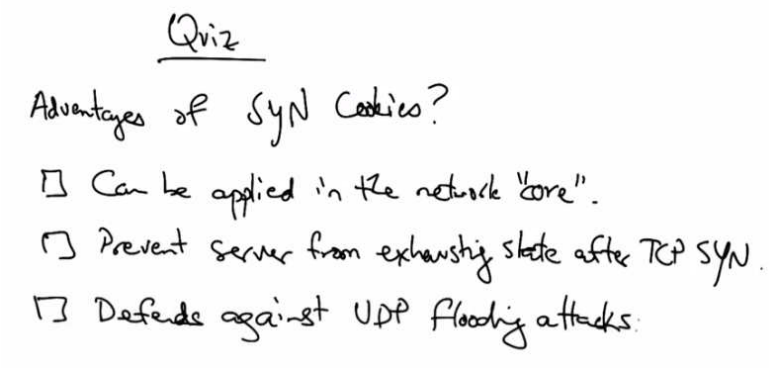


So in a typical TCP three-way handshake, the client sends a SYN packet to the server, the server responds with the SYN-ACK, and the client then returns with an ACK to the SYN-ACK, at which point the connection is established. The problem in a typical TCP three-way handshake is that the client can send a SYN and cause the server to allocate a socket buffer for that TCP connection. But then if the client never returns, the client can force the server to allocate many, many socket buffers simply by sending a lot of SYNs and never returning. In fact, these could even be from spoofed IP addresses. So in other words, the client has absolutely no accountability and no obligation to return to send the final ACK, and yet can cause the server to allocate resources.



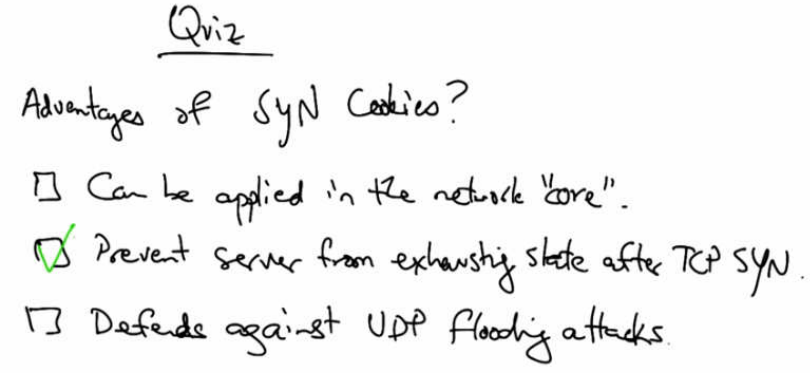
This is a problem, and the solution to this is called SYN cookies. In the TCP SYN cookie approach, when the server receives a SYN from the client, the server, instead of allocating a socket buffer for the tuple associated with the connection, the server keeps no state, and instead picks an initial sequence number for the connection that's a function of the client's IP address and port, and the server's IP address and port, as well as a random knots to prevent replay attacks. An honest client that returns can then reply with an acknowledgement with that sequence number in the packet. The server can check that sequence number simply by rehashing all of the information that it already has, thereby determining that the acknowledgement here corresponded to the previous SYN-ACK that it had sent the client without requiring the server to store any state. Only if the sequence number matches the one picked by the server in the SYN-ACK does the server actually establish the connection.

TCP SYN Cookie Quiz



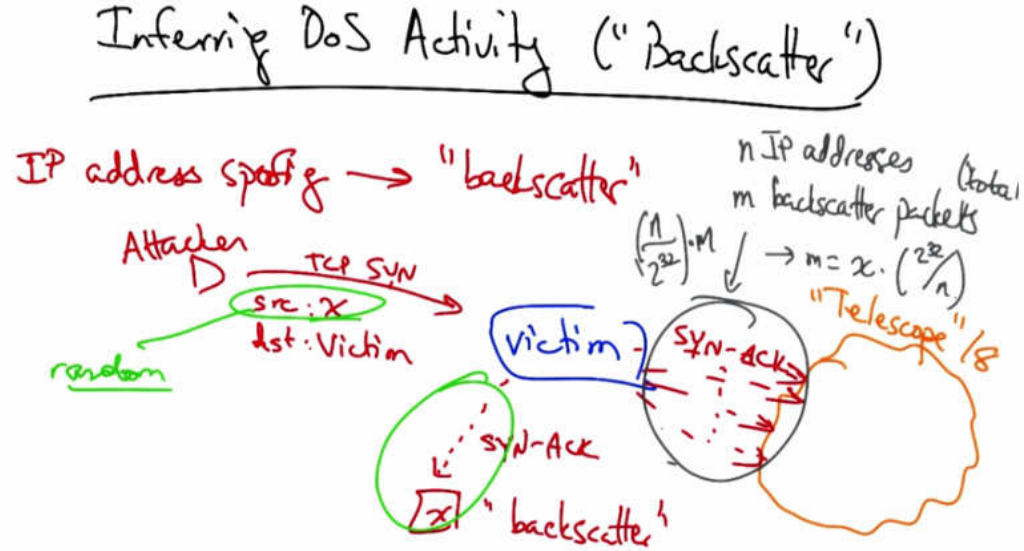
So as a quick quiz, what are some of the advantages of TCP Syn cookies? Is it that they can be applied to filter traffic in the network core? Is it that they can prevent the server from exhausting state by setting up socket buffers after receiving a TCP Syn? Or is it that they can defend against UDP flooding attacks?

TCP SYN Cookie Quiz Answer



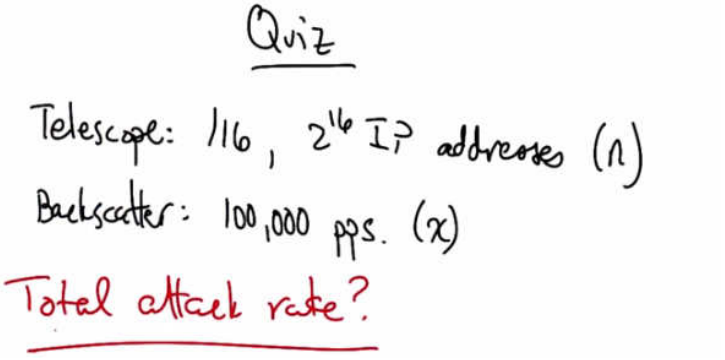
TCP SYN cookies can prevent a server from exhausting state after receiving the initial TCP SYN packet.

Inferring Denial of Service using Backscatter



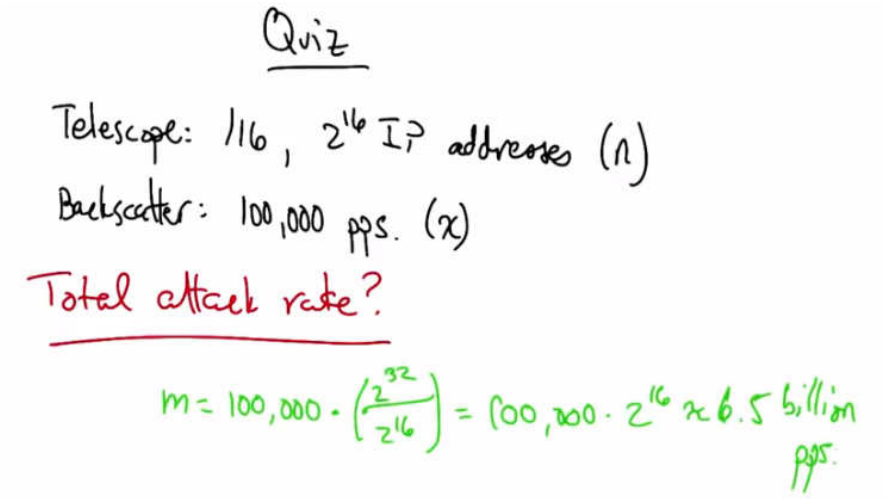
Let's talk about how to infer denial of service activity using a technique called backscatter. The idea behind backscatter is that when an attacker spoofs a source IP address, say on a TCP SYN flood attack, that the replies to that initial TCP SYN from the victim will go to the location of the source IP address. This replies to forged attack messages are called "backscatter". Now the interesting thing about backscatter is that if we can assume that the source IP addresses are selected by the attacker at random, and we could set up a portion of the network where we could monitor this back scatter traffic coming back as SYN-ACK replies to forged source IP addresses, if we assume that these source IP addresses are picked uniformly at random, then the amount of traffic that we see as back scatter represents exactly a fraction that's proportional to the size of the overall attack. So, for example, if we monitor N IP addresses and we see M attack packets, then we expect to see here N over two to the 32 of the total back scatter packets, and hence of the total attack rate. If we want to compute the total attack rate, we simply invert this fraction. So for example, in this case, if our telescope were a slash eight, or two to the 24th IP addresses, we would simply multiply our observed attack rate x by two to the 32 divided by two to the 24 or 255.

Backscatter Quiz



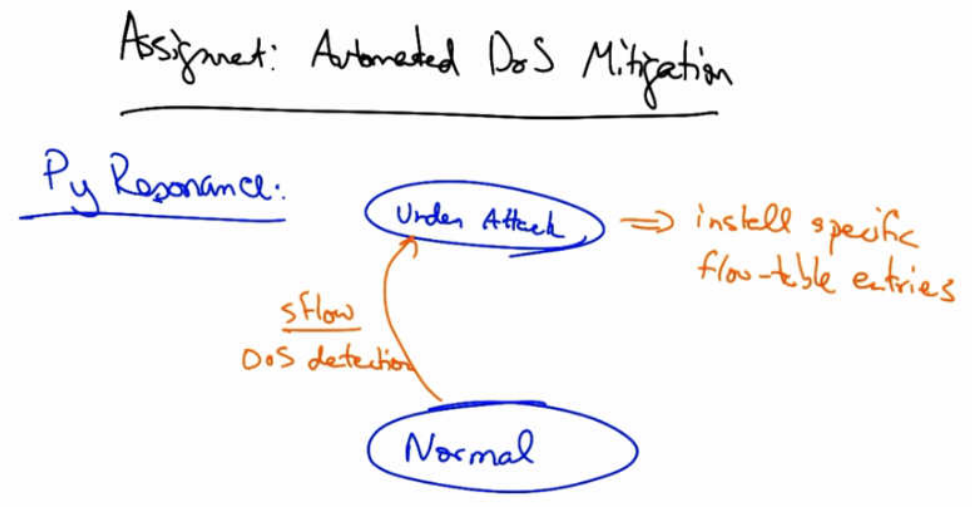
As a quick quiz, let's suppose that our telescope is monitoring two to the 16th IP addresses. And let's suppose that in that telescope, we see 100,000 packets per second. What's the total attack rate?

Backscatter Quiz Answer



Since we're monitoring one 2 to the 16th of the entire internet, or 1 over 65,535 of the total internet, we simply need to take the rate that we've observed and invert that. In this case, that rate would be roughly 6.5 billion packets per second.

Automated Denial of Service Attack Mitigation



In the assignment you will use a Pyretic controller to mitigate a DOS attack. We will use an extension of Pyretic called Py Resonance, which allows for composition of finite state machines that run various programs depending on the state of the network. Your network will start in a normal state, but will use an sFlow-based Denial of Service detector to indicate that the network

has come under attack. Your sFlow event will cause the controller to change states, and hence it will install specific flow-table entries that mitigate the effects of the Denial of Service attack. The assignment that is spelled out on the home page has links to some more in-depth descriptions of this particular assignment and your task is writing a Py Resonance application to mitigate the DOS attack.