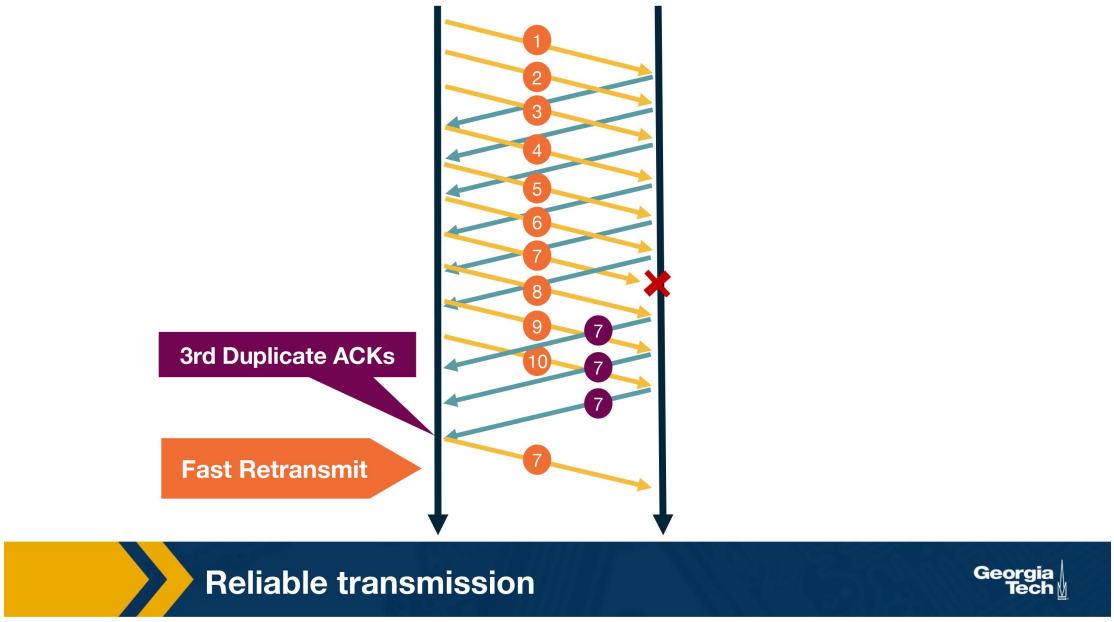
目录



Lesson 1

Why Study Computer Networks?

Internet growth

The Internet is one of the most exciting and influential inventions. Even though it started as a research experiment that escaped from a lab, it eventually evolved into a global communications infrastructure, that has been transforming almost all aspects of our lives with tremendous impact. Given the explosion of applications that are becoming available and the technologies that make it possible for different types of devices to connect to the Internet (for example IoT, vehicles, sensors, home devices, etc), the number of Internet users keeps increasing. As of June 2018, The number of Internet users has been estimated at about 3.2 billion, while the number of estimated Internet users by 2020 is 4 billion.

Networks play an instrumental a role in our society

Indeed, the Internet has been playing a transformative role in our lives. Just to name a few examples, it has been changing the way we do business; for example e-commerce, advertising and cloud computing applications. It has been changing the way we connect and communicate; for example, e-mail, instant messaging, social networking and virtual worlds applications have been extremely popular. The Internet has been changing even how we fight; for example, we frequently read headline news about large scale cyber attacks, incidents related to the distribution of fake news, censorship and nationwide attacks. In turn, these developments have law implications both nationwide and on a global scale that we hadn't considered in the past; for example, we have to examine questions such as which countries are responsible for the Internet traffic that is crossing national boundaries, while the end hosts are located in multiple countries across the globe?

Networking is a playground for interdisciplinary research innovations

The Internet's huge transformative role is connected with an ever-expanding and evolving collection of technologies, system and protocol architectures, algorithms, as well as powerful applications. Indeed, the Internet is an amazing "playground" of ongoing cross-disciplinary innovations that are coming from multiple fields such as distributed systems, operating systems, computer architecture, software engineering, algorithms and data structures, graph theory,

queuing theory, game theory and mechanism designs stemming from machine learning and AI, cryptography, programming languages, and formal methods, and more.

Networking offers multidisciplinary research opportunities with potential for impact

Networking is a field that offers tremendous opportunities for interdisciplinary research work that sometimes crosses fields that can be very different from each other. For example in the context of research projects that study how to incentivize Internet providers to keep their networks clean from infected hosts, the research work may span across fields such as Internet security, economics, and social sciences. What is perhaps the most exciting aspect about research in the networking field, is the ability to design innovative and impactful solutions, and immediately put them to test by leveraging existing platforms.

A Brief History of the Internet

J.C.R. Licklider proposed the "Galactic Network" (1962)

The first vision of a Network - proposed as “Galactic Network” - by J.C.R. Licklider was at MIT back in 1962. He envisioned that everyone could quickly access data through a set of interconnected computers. Licklider - as the head of the research program at Defense Advanced Research Projects Agency (DARPA) - led a group of researchers to experiment connecting two computers. An MIT researcher, Lawrence G. Roberts connected one computer in MA to another computer located in CA with a low-speed dial-up telephone line.

The ARPANET (1969)

The results of the first experiments showed that time-shared infrastructure was working sufficiently well at that moment. But also at the same time researchers indicated the need for packet switching technology. Roberts continued developing the computer network concept, which resulted in the first network which was connecting four nodes (from UCLA, Stanford Research Institute, UCSB and Univ. of Utah, respectively) into the initial ARPANET by the end of 1969.

Network Control Protocol (NCP), an initial ARPANET host-to-host protocol (1970)

As the number of computers that were added to the ARPANET increased quickly, research work proceeded to designing protocols. The initial ARPANET Host-to-Host protocol called Network Control Protocol (NCP) was introduced in 1970, and it allowed the network users to begin developing applications. One of the first applications that launched was email in 1972.

Internetworking and TCP/IP (1973)

At the same time, a DARPA team of researchers led by Bob Kahn, introduced the idea of open-architecture networking so that the individual networks may be independently designed and developed, in accordance with the specific environment and user requirements of that network. This led researchers to develop a new version of the NCP protocol which would eventually be called the Transmission Control Protocol / Internet Protocol (TCP/IP). Khan collaborated with Vint Cerf in Stanford and presented the original TCP paper in 1973. The first version of TCP later split its functionalities into two protocols, the simple IP which provided only for addressing and forwarding of individual packets, and the separate TCP which focused on service features such as flow control and recovery from lost packets.

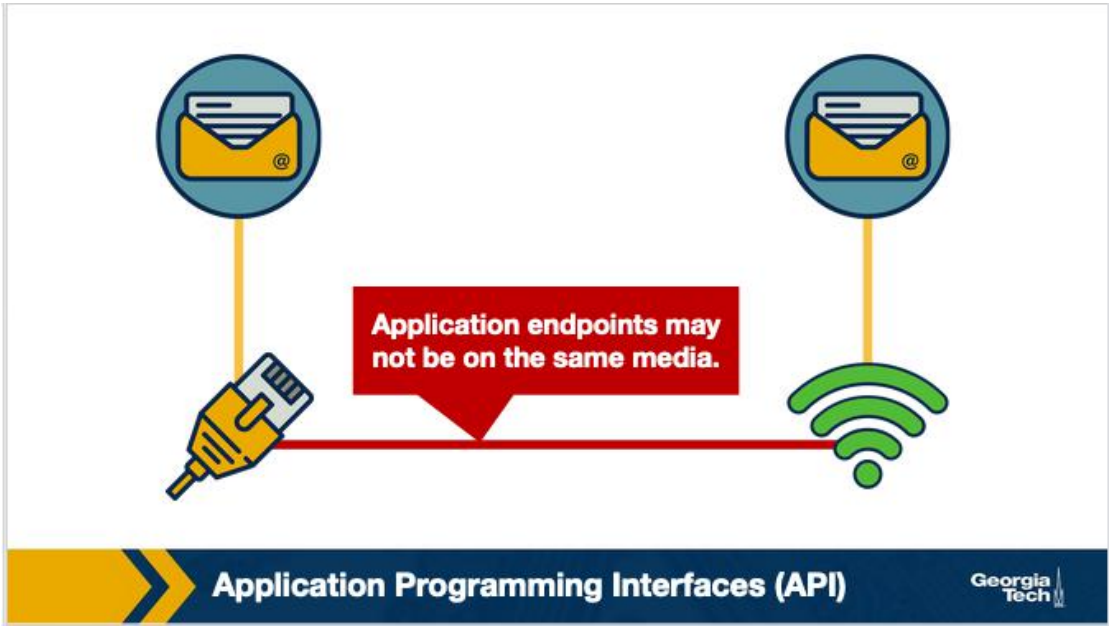
The Domain Name System (DNS) (1983) and the World Wide Web (WWW) (1990)

The scale of the Internet was increasing rapidly, and as a result it was no longer feasible to have a single table of hosts to store names and addresses. The Domain Name System (DNS) - which was designed to translate domain names to IP addresses by a scalable distributed mechanism - was introduced by Paul Mockapetris at USC in 1983. More applications sprung up quickly. One of the first and most popular applications was the World Wide Web (WWW), which was introduced by a team of researchers led by Tim Berners-Lee.

Internet Architecture Introduction

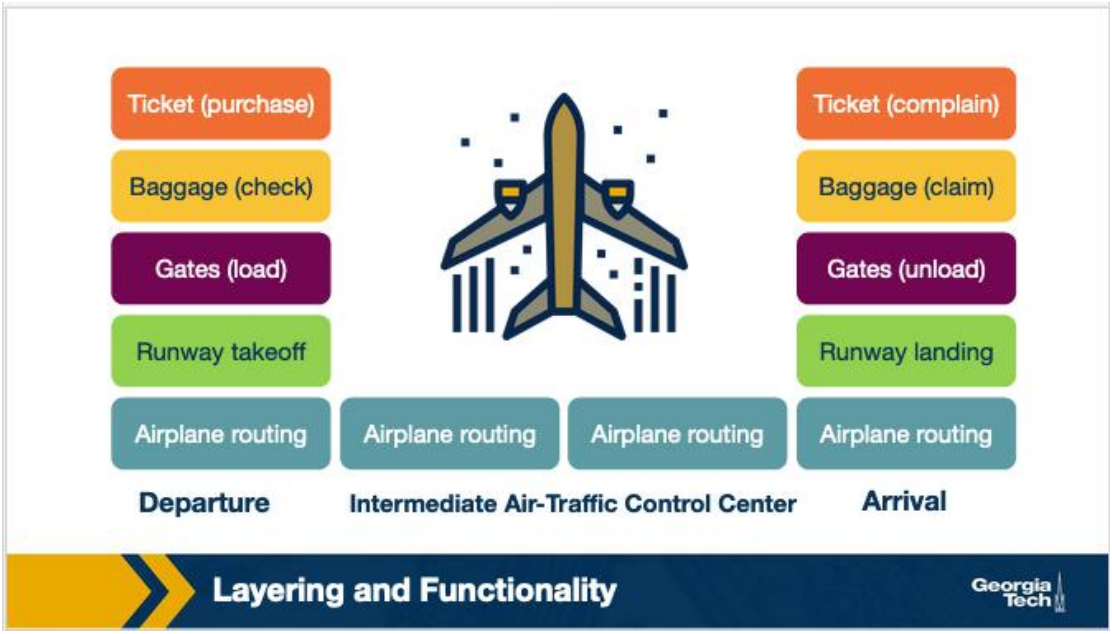
After looking at the major milestones in the history of the Internet, let's take a closer look into the current architectural design of the Internet.

Connecting hosts running the same applications but located in different types of networks. A computer network is a complex system that is built on top of multiple components. These components can vary in technologies making up different types of networks that offer different types of applications. For example in the figure below, we have two BitTorrent clients that communicate even though they are using very different networks/technologies (Wifi vs Ethernet). So, how do these technologies and components interconnect and come together to meet the needs of each application? The designers of the network protocols provide structure to the network architecture by organizing the protocols into layers.



Architecture, layers and functionalities. So, the functionalities in the network architecture are implemented by dividing the architectural model into layers. Each layer offers different services.

An analogy. An analogy we can use to explain a layered architecture is the airline system. Let's look first at the actions that a passenger needs to take to move from the origin to the destination place. The passenger purchases the ticket, checks the bags, goes through the gates and after the plane takes off, the passenger travels on the plane to their final destination. At the final destination, the passenger leaves the aircraft, goes through the gate and claims their baggag



e.

We can look at the above picture to identify a structure (or layers) and the services (or functionalities) that are offered at every component of the structure. Dividing the services into layers we get the framework below. We notice that in this framework every layer implements some functionality. Every layer works based on the service provided by the layer below it, and also it provides some service to the layer that is above.

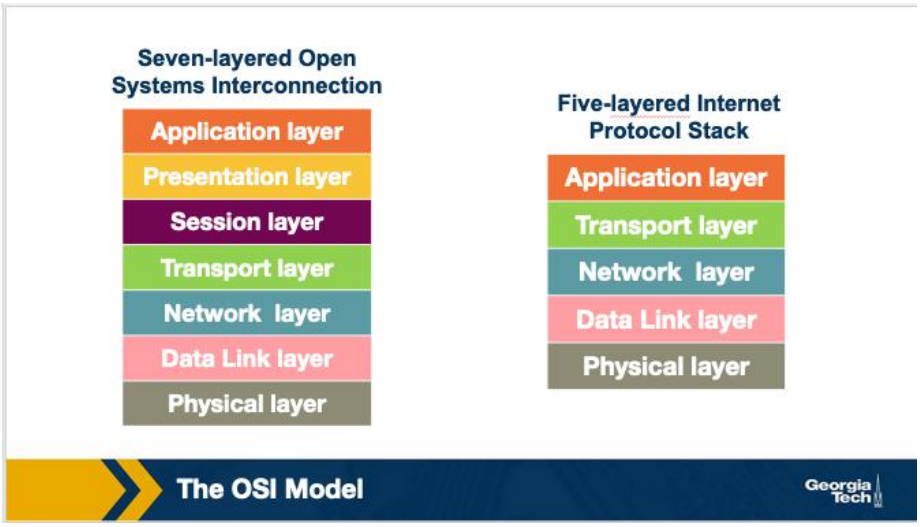
The same principle of layers and functionalities is implemented with the model of the Internet architecture.

Layered architecture advantages: scalability, modularity, and flexibility. Some of the advantages of having a layered network stack include scalability, modularity and the flexibility to add or delete components which makes it easier overall for cost-effective implementations.

The OSI Model

The Internet architecture follows a layered model, where every layer provides some service to the layer above.

The International Organization for Standardization (ISO) proposed the seven-layered OSI model shown below, which consists of the following layers: application layer, presentation layer, session layer, transport layer, network layer, data link layer, and physical layer.



We will see in later sections a possible explanation about why the Internet architecture came eventually to have this form. Separating the functionalities into layers offers multiple advantages. But, are there disadvantages of the layered protocol stack model? Some of the disadvantages include:

1. Some layers functionality depends on the information from other layers, which can violate the goal of layer separation.
2. One layer may duplicate lower layer functionalities. For example, the functionality of error recovery can occur in lower layers, but also on upper layers as well.
3. Some additional overhead that is caused by the abstraction between layers.

In the following sections, we will go through a brief overview of the layers, and more specifically we will focus on what each layer does (service), how the layer is accessed (interface), how the layer is implemented (example protocols), and how we refer to the packet of information it handles.

Application, Presentation, and Session Layers

The Application Layer:

The application layer includes multiple protocols, some of the most popular ones include: 1) The HTTP protocol (web), SMTP (e-mail), 2) The FTP protocol (transfers files between two end hosts), and 3) The DNS protocol (translates domain names to IP addresses). So the services that this layer offers are multiple depending on the application that is implemented. The same is true for the interface through which it is accessed, and the protocol that is implemented. At the application layer, we refer to the packet of information as a message.

The Presentation Layer:

The presentation layer plays the intermediate role of formatting the information that it receives from the layer below and delivering it to the application layer. For example, some functionalities of this layer are formatting a video stream or translating integers from big endian to little endian format.

The Session Layer:

The session layer is responsible for the mechanism that manages the different transport streams that belong to the same session between end-user application processes. For example, in the case of teleconference application, it is responsible to tie together the audio stream and the video stream.

Transport and Network Layer

The Transport Layer:

The transport layer is responsible for the end-to-end communication between end hosts. In this layer, there are two transport protocols, namely TCP and UDP. The services that TCP offers include: a connection-oriented service to the applications that are running on the layer above, guaranteed delivery of the application-layer messages, flow control which in a nutshell matches the sender's and receiver's speed, and a congestion-control mechanism, so that the sender slows its transmission rate when it perceives the network to be congested. On the other hand, the UDP protocol provides a connectionless best-effort service to the applications that are running in the layer above, without reliability, flow or congestion control. At the transport layer, we refer to the packet of information as a segment.

The Network Layer:

In this layer, we refer to the packet of information as a datagram. The network layer is responsible for moving datagrams from one Internet host to another. A source Internet host sends the segment along with the destination address, from the transport layer to the network layer. The network layer is responsible to deliver the datagram to the transport layer in the destination host. The protocols in the network layer are: 1) The IP Protocol, which we often refer to as "the glue" that binds the Internet together. All Internet hosts and devices that have a network layer must run the IP protocol. The IP protocol defines a) the fields in the datagram, and b) how the source/destination hosts and the intermediate routers use these fields, so the datagrams that a source Internet host sends reach their destination. 2) The routing protocols that determine the routes that the datagrams can take between sources and destinations.

Data Link Layer and Physical Layer

The data link layer:

In this layer, we refer to the packets of information as frames. Some example protocols in this layer include Ethernet, PPP, WiFi.

The data link layer is responsible to move the frames from one node (host or router) to the next node. More specifically, assuming we have a sender and receiver host, the network layer will route the datagram through multiple routers across the path between the sender and the receiver. At each node across this path, the network layer passes the

datagram to the data link layer, which in turn delivers the datagram to the next node. Then, at that node, the link layer passes the datagram up to the network layer.

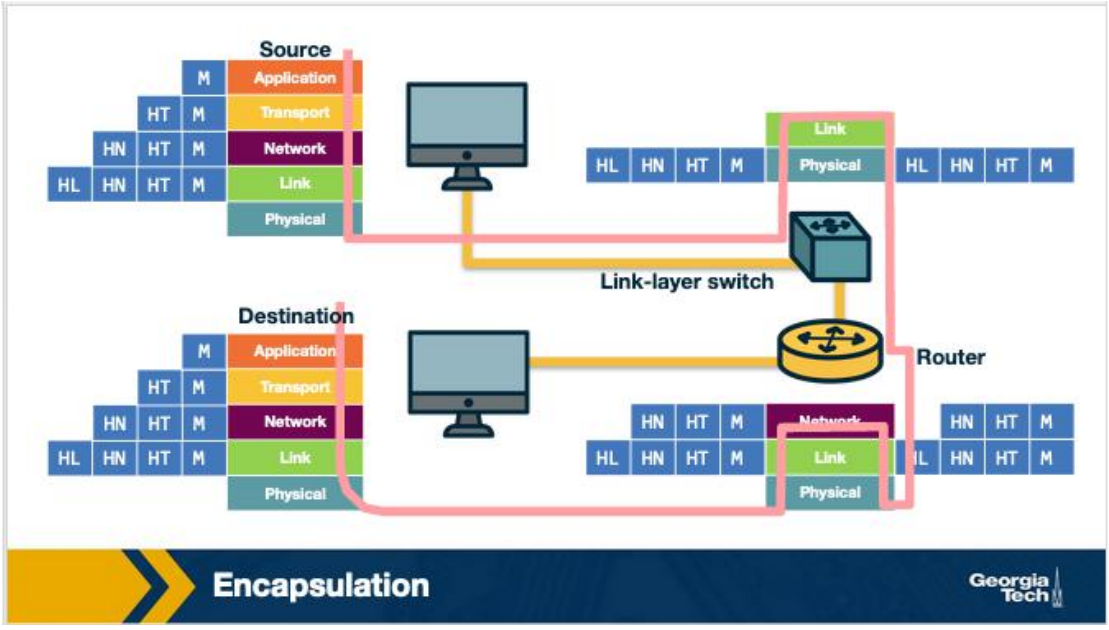
The data link layer offers services that depend on the data link layer protocol that is used over the link. Some example services include reliable delivery, that covers the transmission of the data from one transmitting node, across one link, and finally to the receiving node. We note that this specific type of reliable delivery service is different from the reliable delivery service that is offered by the TCP protocol which offers reliability from the source host to the destination end host.

The physical layer:

The physical layer facilitates the interaction with the actual hardware and is responsible to transfer bits within a frame between two nodes that are connected through a physical link. The protocols in this layer again depend on the link and on the actual transmission medium of the link. One of the main protocols in the data link layer, Ethernet, has different physical layer protocols for twisted-pair copper wire, coaxial cable, and single-mode fiber optics.

Layers Encapsulation

How do the layers and the protocols that run on each layer communicate with each other? To understand the concepts of encapsulation and de-encapsulation, let's take a look at the following diagram which shows the physical path that data take from the sending host to the receiving host.



Encapsulation and De-encapsulation. The sending host sends an application layer message M to the transport layer. The transport layer receives the message, and it appends the transport layer header information (Ht). The application message along with the transport layer header is called segment (or transport-layer segment). The segment thus *encapsulates* the application layer message. This added information can help the receiving host to a) inform the receiver-side transport layer about which application to deliver the message up to, and b) perform error detection and determine whether bits in the message have been changed along the route.

The segment is then forwarded to network layer which in turn, adds it's own network header information (Hn). The entire combination of the segment and the network header is called datagram. We say that the datagram encapsulates the segment. The header information that the network layer appends includes the source and destination addresses of the end hosts. The same process continues for the link layer which in turn it appends its own header information(Hl). The message at the link layer is called frame, which is transmitted across the physical medium. At each layer the message is a combination of two parts: a) the payload which is the message from the layer above, and b) the new appended header information. At the receiving end, the process is reversed, with headers being stripped off at each layer. This reverse process is known as de-encapsulation.

Intermediate devices and encapsulation. The path that connects the sending and the receiving hosts may include intermediate layer-3 devices, such as routers, and layer-2 devices such as switches. We will see later how switches and routers work, but for now we note that both routers and layer-2 switches implement protocol stacks similarly to end-hosts. The difference is that routers and layer-2 switches do not implement all the layers in the protocol stack; routers implement layers 1 to 3, and layer-2 switches implement layers 1 to 2. So, going back to our diagram, when the data leave the sending host and they are received by the layer-2 switch, the switch implements the same process of de-encapsulation to process the data and encapsulation to send the data forward to the next device.

A design choice. We note again that end-hosts implement all five layers while the intermediate devices don't. This design choice ensures that the Internet architecture puts much of its complexity and intelligence at the edges of the network while keeping the core simple. Next, we will look deeper into the so-called end-to-end principle.

The End to End Principle

The end-to-end (e2e) principle is a design choice that characterized and shaped significantly the current architecture of the Internet. The e2e principle suggests that specific application-level functions usually cannot, and preferably should not be built into the lower levels of the system at the core of the network.

In simple terms, the e2e principle is summarized as: the network core should be simple and minimal, while the end systems should carry the intelligence. As mentioned in the seminal paper "End-to-End Arguments in System Design" by Saltzer, Reed, and Clark: "*The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the endpoints of the communications system. Therefore, providing that questioned function as a feature of the communications systems itself is not possible.*"

The same paper reasoned that many functions can only be completely implemented at the endpoints of the network, so any attempt to build features in the network to support specific applications must be avoided, or only viewed as a tradeoff. The reason was that not all applications need the same features and network functions to support them. Thus building such functions in the network core is rarely necessary. So, systems designers should avoid building any more than the essential and commonly shared functions into the network.

Many people argue that the e2e principle allowed the internet to grow rapidly, because evolving innovation took place at the network edge, in the form of numerous applications and a plethora of services, rather than in the middle of the network, which could be hard to later modify.

What were the designers' original goals that led to the e2e principle? Moving functions and services closer to the applications that use them, increases the flexibility and the autonomy of the application designer to offer these services to the needs of the specific application. Thus, the higher-level protocol layers, are more specific to an

application. Whereas the lower-level protocol layers are free to organize the lower-level network resources to achieve application design goals more efficiently and independently of the specific application.

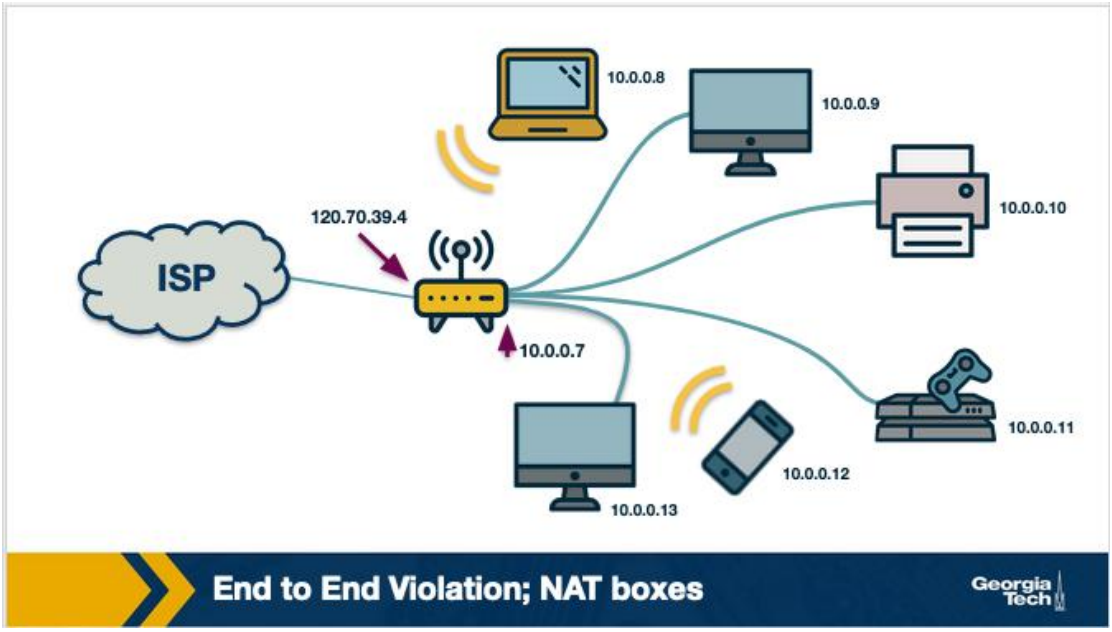
Violations of the End-to-End Principle and NAT Boxes

Despite the fact that the e2e principle offers multiple advantages to the Internet and its evolution, there have still been cases where this principle needs to be violated.

Some examples of the e2e violation:

Examples include firewalls and traffic filters. The firewalls usually operated at the periphery of a network and they monitor the network traffic that is going through, to allow or drop traffic, if the traffic is flagged as malicious. Firewalls violate the e2e principle since they are intermediate devices that are operated between two end hosts and they can drop the end hosts communication.

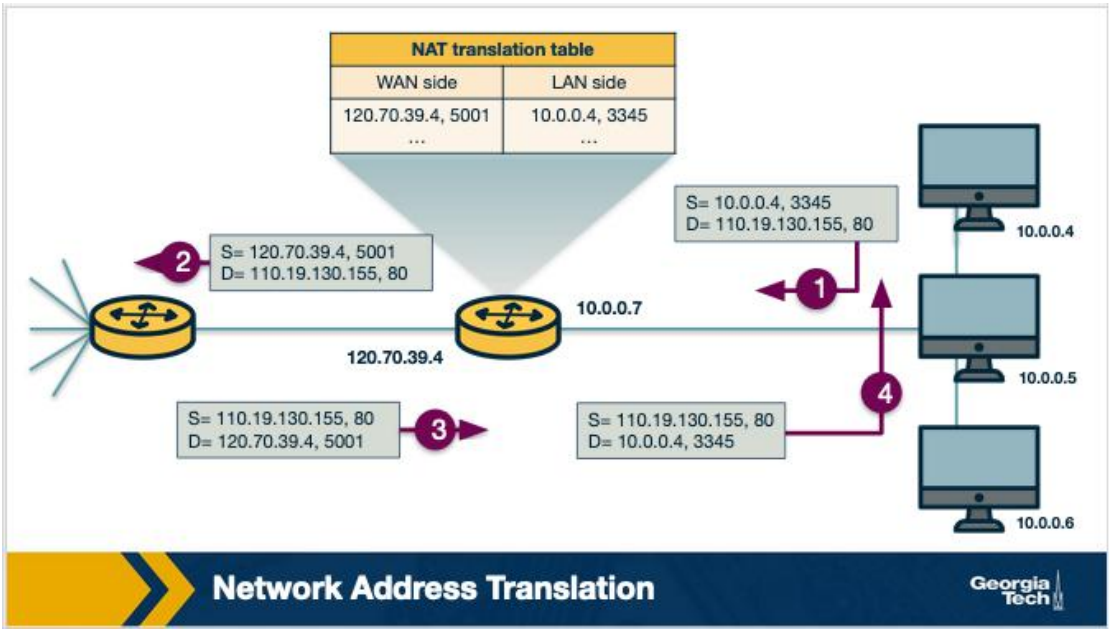
Another example of an e2e violation is the Network Address Translation (NAT) boxes. NAT boxes help us as a bandaid measure to deal with the shortage of Internet addresses. Let's see in more detail how a NAT-enabled home router operates. Let's assume we have a home network, where we have multiple devices we want to connect to the Internet. An internet service provider typically assigns a single public IP address (120.70.39.4) to the home router and specifically to the interface that is facing the public global Internet, as shown in the figure below.



The other interface of the NAT-enabled router that is facing the home network (along with all other device-interfaces in the home network) gets an IP address that belongs to the same private subnet. This subnet must belong to the address spaces that are reserved as private, eg 10.0.0/24 or 192.168.0.0/24. This means that the IP addresses that belong to this private subnet only have meaning to devices within that subnet. So we can have hundreds of thousands of private networks with the same address range (eg 10.0.0.0/24). But, these private networks are always behind a

NAT, which takes care of the communication between the hosts on the private network and the hosts on the public Internet.

All traffic that leaves the home router and it is destined to hosts in the public Internet must have as the source IP address the IP of the public facing interface of the NAT-enabled router. Similarly, all traffic that enters the home network through the router, must have as the destination address the IP of the public facing interface of the NAT-enabled router. The home router plays the role of a translator maintaining a NAT translation table, and it rewrites the source and destination IP addresses and ports.



The translation table provides a mapping between the public facing IP address/ports, and the IP addresses/ports that belong to hosts inside the private network. For example, let's assume that a host 10.0.0.1 inside the private network, uses port 3345 to send traffic to a host in the public Internet with IP address 128.119.40.186 and port 80. Then the NAT table says that packets with the source IP address of 10.0.0.1 and source port 3345, they should be rewritten to a source address 138.76.29.7 and a source port of 5001 (or any source port number that is not currently used in the NAT translation table). Similarly, packets with a destination IP address of 138.76.29.7 and destination port of 5001, they will be rewritten to destination IP address 10.0.0.1 and destination port 3345.

Why the NAT boxes violate the e2e principle?

The hosts that are behind NAT boxes are not globally addressable, or routable. As a result, it is not possible for other hosts on the public Internet to initiate connections to these devices. So, if we have a host behind a NAT and a host in the public Internet, then by default they cannot communicate without the intervention of a NAT box.

There are some workarounds to allow hosts to initiate connections to hosts that behind NATs. Some example tools and protocols include STUN (a tool that allows hosts to discover NATs and the public IP address and port number that the NAT has allocated for the application that the host wants to communicate with), and UDP hole punching (it established bidirectional UDP connections between hosts behind NATs).

The Hourglass Shape of Internet Architecture

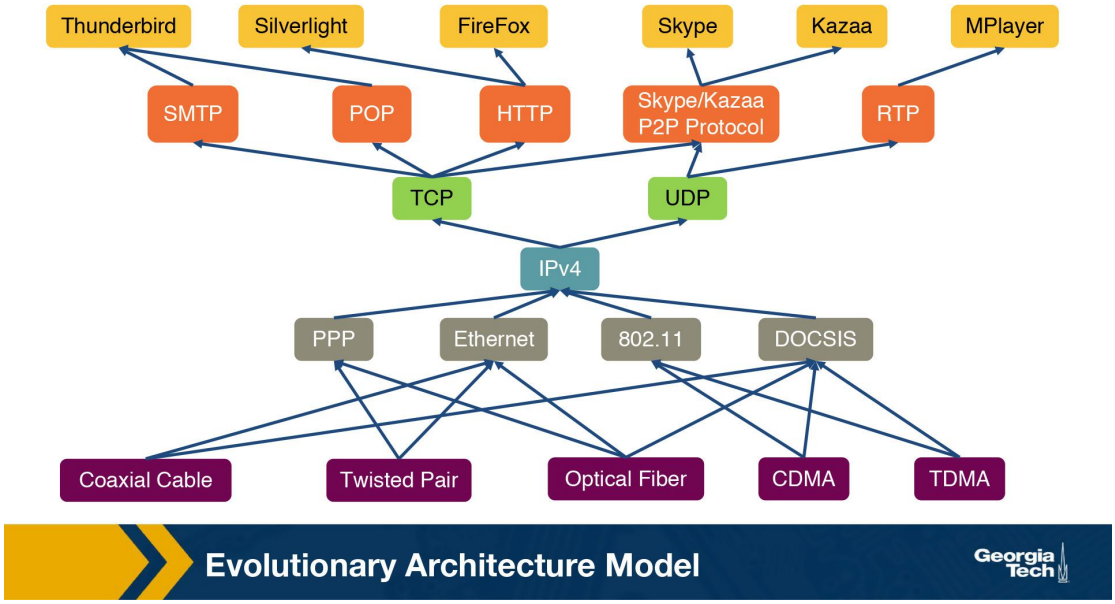
The Internet protocol stack has a layered architecture that resembles an hourglass shape. Was the Internet architecture always shaped like an hourglass, and there has always been a single protocol at the network layer? If we look back in the early nineties, we will see that there were several other network-layer protocols that were competing with IPv4. For example, Novell's IPX and the X.25 network protocol used in Frame Relay. So the network layer did not include only one protocol, but there were multiple protocols that were competing with each other at that time.

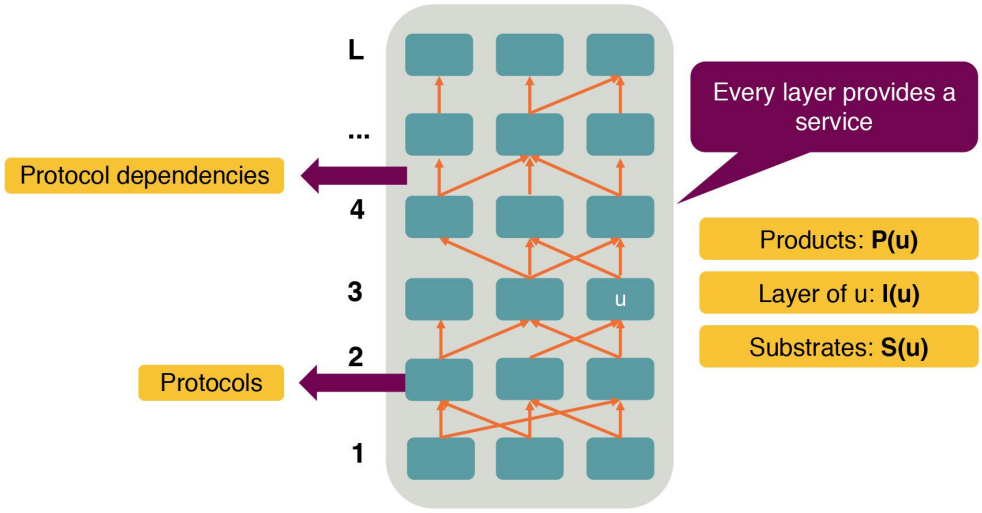
Why has there have been more frequent innovations at the lower or higher layers of the protocol hourglass? Why have the protocols at the waist of the hourglass (mostly IPv4, TCP, and UDP) been difficult to replace, and have they outcompeted any protocols that offer the same or similar functionalities? Looking ahead, and assuming that we want to design and introduce new and potentially better protocols, how can we make it more likely that the new protocols will outcompete and replaces existing and widely used incumbent protocols?

Researchers have suggested a model called the Evolutionary Architecture model, or *EvoArch*, that can help to study layered architectures and their evolution in a quantitative manner. Through this model researchers were able to explain how the hierarchical structure of the layer architecture eventually lead to the hourglass shape.

In the next topic, we will talk about the details of the model and how it can help us explain the evolution of the Internet architecture.

Evolutionary Architecture Model





Layered Acyclic Network

In this section, we will talk about a model that attempts to answer our previous questions. Researchers have suggested a model - the Evolutionary Architecture model or EvoArch - that can help to study layered architectures, and their evolution in a quantitative manner. The EvoArch model considers an abstract model of the Internet's protocol stack that has the following components:

Layers. A protocol stack is modeled as a directed and acyclic network with L layers.

Nodes. Each network protocol is represented as a node. The layer of a node u is denoted by l(u).

Edges. Dependencies between protocols are represented as directed edges.

Node incoming edges. If a protocol u at layer l uses the service provided by a protocol w at the lower layer l−1, then this is represented by an "upwards" edge from w to u.

Node substrates. We refer to substrates of a node u, S(u), as the set of nodes that u is using their services. Every node has at least one substrate, except the nodes at the bottom layer.

Node outgoing edges. The outgoing edges from a node u terminate at the products of u. The products of a node u are represented by P(u).

Layer generality. Each layer is associated with a probability s(l), which we refer to as layer generality. A node u at layer l+1 selects independently each node of layer l as the substrate with probability s(l). The layer generality decreases as we move to higher layers, and thus protocols at lower layers are more general in terms of their functions or provided services than protocols at higher layers. For example, in the case of the Internet protocol stack, layer 1 is very general and the protocols at this layer offer a very general bit transfer service between two connected points, which most higher layer protocols would use.

Node evolutionary value. The value of a protocol node, v(u), is computed recursively based on the products of u. By introducing the evolutionary value of each node, the model captures the fact that the value of a protocol u is driven by the values of the protocols that depend on it. For example, let's consider again the Internet protocol stack. TCP has a high evolutionary value because it is used by many higher layer protocols and some of them being valuable themselves. Let's assume that we introduce a brand new protocol, at the same layer as TCP, that may have better performance or other great new features. The new protocol's evolutionary value will be low if it is not used by important or popular higher layer protocols, regardless of the great new features it may have. So the

evolutionary value determines if the protocol will survive the competition with other protocols, at the same layer, that offer similar services.

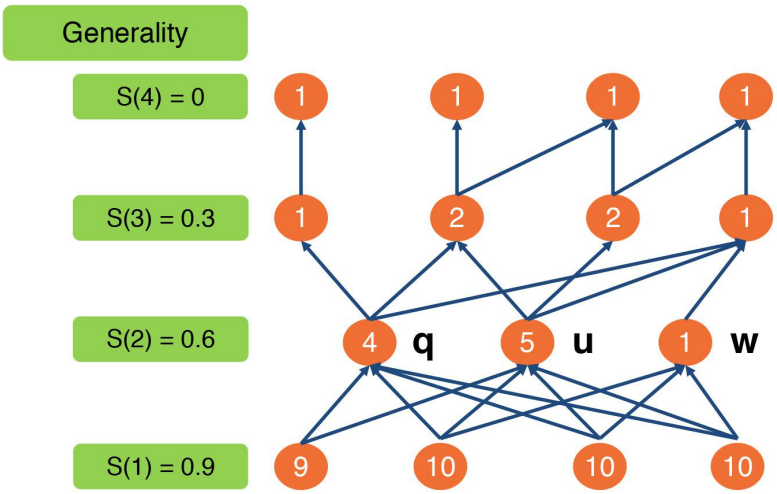
Node competitors and competition threshold. We refer to the competitors of a node u, C(u), as the nodes at layer l that share at least a fraction c of node u's products. We refer to the fraction c, as the competition threshold. So, a node w competes with a node u, if w shares at least a fraction c of u's products.

Node death rate. The model has a death and birth process in place, to account for the protocols that cease or get introduced respectively. The competition among nodes becomes more intense, and it is more likely that a protocol u dies if at least one of its competitors has a higher value than itself. When a node u dies, then its products also die, if their only substrate is u.

Node basic birth process. The model, in its simplest version, has a basic birth process in place, where a new node is assigned randomly to a layer. The number of new nodes at a given time is set to a small fraction (say 1% to 10%) of the total number of nodes in the network at that time. So, the larger a protocol stack is, then the faster it grows.

Toy example:

To illustrate the above model and the parameters, let's consider a toy network example with L equal to 4 layers. The evolutionary value of each node is shown inside each circle. The generality probability for each layer is shown at the left of each layer, and it is denoted as s(l). As we noted earlier, the generality of the layers decreases as we move to higher layers, so on average, the number of products per node decreases as well. Let's further assume that we have a competition threshold c = ⅝ . Nodes u, q and w compete in layer 2. U and q compete, but this is unlikely to cause q to die because u and q have comparable evolutionary values. In contrast, it is likely that w will die because its value is much less than that of its maximum-value competitor, u.



Toy Network with Four Layers

EvoArch iterations:

EvoArch is a discrete-time model that is executed over rounds. At each round, we perform the following steps: A) We introduce new nodes, and we place them randomly at layers. B) We examine all layers, from the top to the bottom, and we perform the following tasks: 1) We connect the new nodes that we may have just introduced to that layer, by choosing substrates based on the generality probabilities of the layer below s(l−1), and by choosing products for them

based on the generality probability of the current layer s(l). 2) We update the value of each node at each layer l, given that we may have new nodes added to the same layer l. 3) We examine all nodes, in order of decreasing value in that layer, and remove the nodes that should die. C) Finally, we stop the execution of the model when the network reaches a given number of nodes.

The figure above shows the width of each layer we execute the EvoArch model for a network of 10 layers over multiple rounds. The main takeaway message from this figure is that the layer width decreases as we move from the bottom layer to a middle layer, around layer 5, and then it increases again as we move towards the top layer.

Implications for the Internet Architecture and future Internet architecture:

With the help of the EvoArch model, how can we explain the survival of the TCP/IP stack given that it appeared around the 70s or 80s when the telephone network was very powerful? The EvoArch model suggests that the TCP/IP stack was not trying to compete with the telephone network services. The TCP/IP was mostly used for applications such as FTP, E-mail, and Telnet, so it managed to grow and increase its value without competing or being threatened by the telephone network, at that time that it first appeared. Later it gained even more traction, with numerous and powerful applications relying on it.

IPv4, TCP, and UDP provide a stable framework through which there is an ever-expanding set of protocols at the lower layers (physical and data-link layers), as well as new applications and services at the higher layers. But at the same time, these same protocols have been difficult to replace or even modify significantly. EvoArch provides an explanation for this. A large birth rate at the layer above the waist can cause death for the protocols at the waist if these are not chosen as substrates by the new nodes at the higher layers. The waist of the Internet architecture is narrow, but also the next higher layer (the transport layer) is also very narrow and stable. So, the transport layer acts as an “evolutionary shield” for IPv4, because any new protocols that might appear at the transport layer are unlikely to survive the competition with TCP and UDP which already have multiple products. In other words, the stability of the two transport protocols adds to the stability of IPv4, by eliminating any potential new transport protocols, that could select a new network layer protocol instead of IPv4.

Finally, in terms of future and entirely new Internet architectures, the EvoArch model predicts that even if these brand new architectures do not have the shape of an hourglass initially, they will probably do so as they evolve, which will lead to new ossified protocols. The model suggests that one way to proactively avoid these ossification effects, that we now experience with TCP/IP, a network architect should try to design the functionality of each layer so that the waist is wider, consisting of several protocols that offer largely non-overlapping but general services, so that they do not compete with each other.

Optional Reading: Architecture Redesign

Why a clean-slate design approach?

Some of the major design principles of the current Internet architecture are layering, packet switching, a network of collaborating networks, intelligent end-systems as well as the end-to-end argument. Despite our initial intentions, the Internet is currently facing major challenges in multiple areas such as security, resilience and availability, scalability and management, quality of service, user experience, and economics. But what if we redesigned the Internet

architecture from scratch? Many researchers believe that it is necessary, and timely, to rethink the fundamental assumptions and design decisions via a clean-slate design approach. That clean-slate approach would be based on out of the box thinking, the design of new network architectures, and experimentation to evaluate the new ideas, to improve them and also to give them a realistic chance of deployment.

The clean-slate design as a process:

An important aspect about designing new Internet architectures through a clean-slate approach, is the ability to deploy and thoroughly test them, which can take place at an appropriate experimental facility. Such a facility has to fulfill some requirements such as offer a large scale infrastructure, include different technologies, attract real users and their traffic, enable parallel experiments with distinct networking architectures such as different naming schemes, different layering approaches, and different forwarding techniques, while at the same time, new services should be able to explore the new capabilities and should be made available to users who opt-in. In this context, a clean-slate should be viewed as a design process, rather than as a result in itself. Through this process, we may identify innovative services and applications, which may become mature enough to be commercially deployed on the existing Internet. Another possibility may be that we may create an entirely new network architecture, which eventually replaces today's architecture. Or perhaps, the most "conservative" outcome may even be that we learn that the current Internet architecture is the "best" possible solution. On the other hand, the most "radical" outcome can be that such an experimental facility, which allows multiple sub-system architectures and network services to co-exist, becomes the blueprint for the future Internet.

Redesigning the Internet architecture to optimize for control and management:

One research group ("4D") for example, started from a small set of clean slate design principles different from those of the Internet today: network-level objectives, network-wide views, and direct control, with functionality in four components: the data, discovery, dissemination, and decision planes. In their work, the decision plane has a network-wide view of the topology and traffic, and exerts direct control over the operation of the data plane – radically different from today's Internet - no decision logic is hardwired in protocols distributed among the network elements. The output of the decision logic is communicated to routers/switches by the dissemination plane. Their work investigates an *extreme design point* where the decision logic is completely separated from distributed protocols. The 4D group argues that technology trends toward ever-more powerful, reliable, and inexpensive computing platforms make their design point attractive in practice. [Greenberg 2005]

Redesigning the Internet architecture to offer better accountability:

Given the fact that IP network layer provides little to no protection against misconfiguration or malicious actions which occur frequently, Researchers proposed network "accountability" in order to establish the foundation for defenses against those behaviors. Accountability is an ability to associate each action with the responsible entity. The proposed work addressed two accountabilities, i.e., source accountability and control-plane accountability, and illustrated that both accountabilities could be improved by a network layer called Accountable Internet Protocol (AIP).

Source accountability is the ability to trace actions to a particular end host and stop that host from misbehaving. It is necessary to describe the address format of AIP first and then illustrate how AIP improved source accountability. AIP addresses are of the form AD:EID, where AD is the identifier for the network that the host belongs to, and EID is a globally unique host identifier. For source accountability, AIP makes use of this self-certifying addressing to develop simple mechanisms that verify the source of packets and drop the packets if the sources are spoofed. In addition, AIP can throttle certain forms of unwanted traffic using a simple "shut-off message".

Control-plane accountability is the ability to pinpoint and prevent attacks on routing. The proposed work suggested ways to improve control-plane accountability by providing origin authentication (ensuring that the network that appears to originate paths is indeed the correct network) and path authentication (checking the integrity of the network path) to detect misleading route advertisements.

Interconnecting Hosts and Networks

We have different types of devices that help to provide connectivity between hosts that are in the same network, or help interconnect networks. These devices offer different services and they operate over different layers.

***Repeaters and Hubs**:* They operate on the physical layer (L1), as they receive and forward digital signals to connect different Ethernet segments. They provide connectivity between hosts that are directly connected (in the same network). The advantage is that they are simple and inexpensive devices, and they can be arranged in a hierarchy. Unfortunately, hosts that are connected through these devices belong to the same collision domain, meaning that they compete for access to the same link.

Bridges and Layer2-Switches: These devices can enable communication between hosts that are not directly connected. They operate on the data link layer (L2) based on MAC addresses. They receive packets and they forward them to reach the appropriate destination. A limitation is the finite bandwidth of the outputs. If the arrival rate of the traffic is higher than the capacity of the outputs then packets are temporarily stored in buffers. But if the buffer space gets full, then this can lead to packet drops.

Routers and Layer3-Switches: These are devices that operate on Layer 3. We will talk more about these devices and the routing protocols on the following lectures.

Learning Bridges

A bridge is a device with multiple inputs/outputs. A bridge transfers frames from an input to one (or multiple) outputs. Though it doesn't need to forward all the frames it receives. In this topic we will talk about how a bridge learns how to perform that task.

A learnings bridge learns, populates and maintains, a forwarding table. The bridge consults that table so that it only forwards frames on specific ports, rather than over all ports.

For example, let's consider the topology on the following figure. When the bridge receives a frame on port 1, with source Host A and destination Host B, the bridge does not have to forward it to port 2.

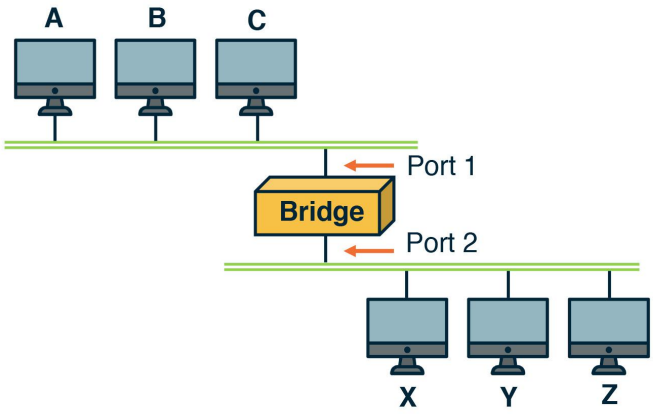


Illustration of a Learning Bridge

So how does the bridge learn? When the bridge receives any frame this is a “learning opportunity” to know which hosts are reachable through which ports. This is because the bridge can view the port over which a frame arrives and the source host. Going back to our example topology, eventually the bridge builds the following forwarding table.

Host	Port
A	1
B	1
C	1
X	2
Y	2
Z	2

Forwarding Table Maintained by a Bridge

Looping Problem in Bridges and the Spanning Tree Algorithm

Unfortunately using bridges to connect LAN's fails, if the network topology results in loops (cycles). In that case, the bridges loop through packets forever!

The answer to this problem is excluding links that lead to loops by running the spanning tree algorithm. Let's represent the topology of the network as a graph. The bridges are represented as nodes and the links between the bridges are represented as edges. The goal of the spanning tree algorithm is to have the bridges select which links (ports) to use for forwarding eliminating loops.

Let's take a look at how bridges run this distributed algorithm.

Every node (bridge) in the graph has an ID. The bridges eventually select one bridge as the root of the topology. Let's see how this selection happens.

The algorithm runs in "rounds" and at every round each node sends to each neighbor node a configuration message with three fields: a) the sending node's ID, b) the ID of the roots as perceived by the sending node, and c) the number of hops between that (perceived) root and the sending node.

At every round, each node keeps track of the best configuration message that it has received so far, and it compares that against the configuration messages it receives from neighboring nodes at that round.

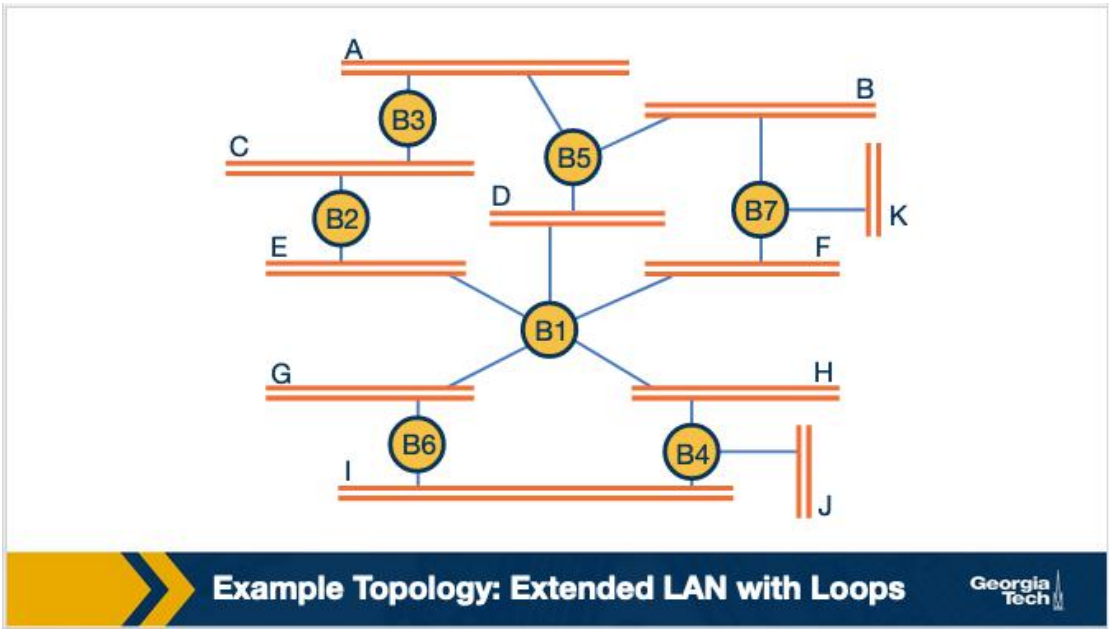
At the very first round of the algorithm, every node thinks that it is the root. So for a node with an ID 3 for example, the node sends a configuration message <3, 3, 0> to its neighbors. Note that the distance of the node from itself (perceived root) is 0.

So how does a node compare two configuration messages? Between two configurations, a node selects one configuration as better if: a) The root of the configuration has a smaller ID, or if b) The roots have equal IDs, but one configuration indicates smaller distance from the root, or if c) Both roots IDs are the same and the distances are the same, then the node breaks the tie by selecting the configuration of the sending node that has with the smallest ID. In addition, a node stops sending configuration messages over a link (port), when the node receives a configuration message that indicates that it is not the root, eg when it receives a configuration message from a neighbor that: a) either closer to the root, or b) it has the same distance from the root, but it has a smaller ID.

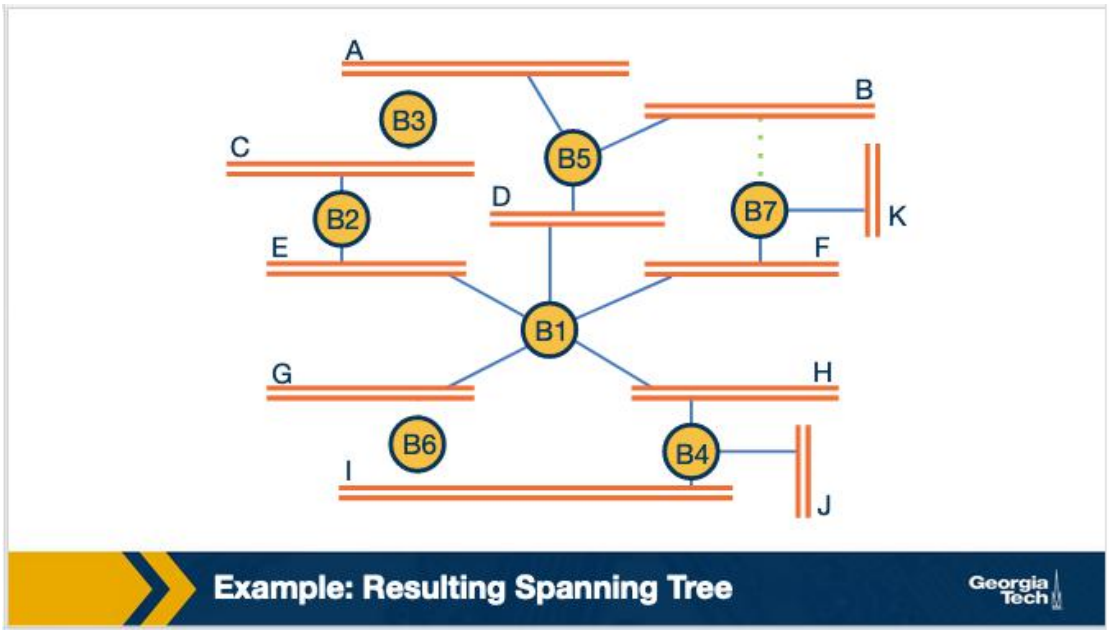
As an example, let's consider the topology below. By running the above steps on this topology, we note that in the first round B3 receives (B2,B2,0) and (B5,B5,0), so it accepts B2 as the root. So in the second round it sends (B3,B2,1) to its neighbors.

Similarly for B2; In the first round, B2 receives (B3,B3,0) and (B1,B1,0), it accepts B1 as the root. So in the second round B2 sends (B2,B1,1).

Finally, B5 receives configuration messages from B3, B7 and B1. B5 accepts B1 as root and sends (B5, B1, 1) to B3. This results to B3 also accepting B1 as root. In addition, B3 realizes that both its neighbors, namely B2 and B5 are closer to the root (B1) than itself. This causes B3 to not select any of its links (ports). So B3 stops participating in forwarding traffic.



The resulting spanning tree is:



Lesson 2

Introduction to Transport Layer and the Relationship between Transport and Network Layer

The transport layer provides an end-to-end connection between two applications that are running on different hosts. Of course the transport layer provides this logical connection regardless if the hosts are in the same network.

Here is how it works; The transport layer on the sender host receives a message from the application layer and it appends its own header. We refer to this combined message as a segment. This transport layer segment is then sent to the network layer which will further append (encapsulate) this segment with its header information. Then it will send it to the receiving host via routers, bridges, switches etc.

One might ask, why do we need an additional layer between the application and the network layer? Recall, that the network-layer is based on a best effort delivery service model. According to this model, the network layer makes a best effort to deliver data packets. Thus, it doesn't guarantee the delivery of packets, nor it guarantees integrity in data. So, here is where the transport layer comes to offer some of these functionalities. This allows application programmers to develop applications assuming a standard set of functionalities that are provided by the transport layer. So the applications can run over diverse networks without having to worry about different network interfaces or possible unreliability of the network.

Within the transport layer, there are two main protocols: User datagram protocol (UDP) and the Transmission Control Protocol (TCP). These protocols differ based on the functionality they offer to the application developers; UDP provides very basic functionality and relies on the application-layer to implement the remaining. On the other hand, TCP provides some strong primitives with a goal to make end-to-end communication more reliable and cost-effective. In fact, because of these primitives, TCP has become quite ubiquitous and is used for most of the applications today. We will now look at these functionalities in detail.

Multiplexing: why we need it?

One of the main desired functionalities of the transport layer is the ability for a host to run multiple applications to use the network simultaneously; which we refer to as multiplexing.

Let us consider a simple example to further illustrate why we need transport-layer multiplexing. Consider a user who is using Facebook while also listening to music on Spotify. Clearly, both of these processes involve communication to two different servers. How do we make sure that the incoming packets are delivered to the correct application? Note that, the network layer uses only the IP address and an IP address alone does not say anything about which processes on the host should get the packets. Thus, we need an addressing mechanism to distinguish the many processes sharing the same IP address on the same host.

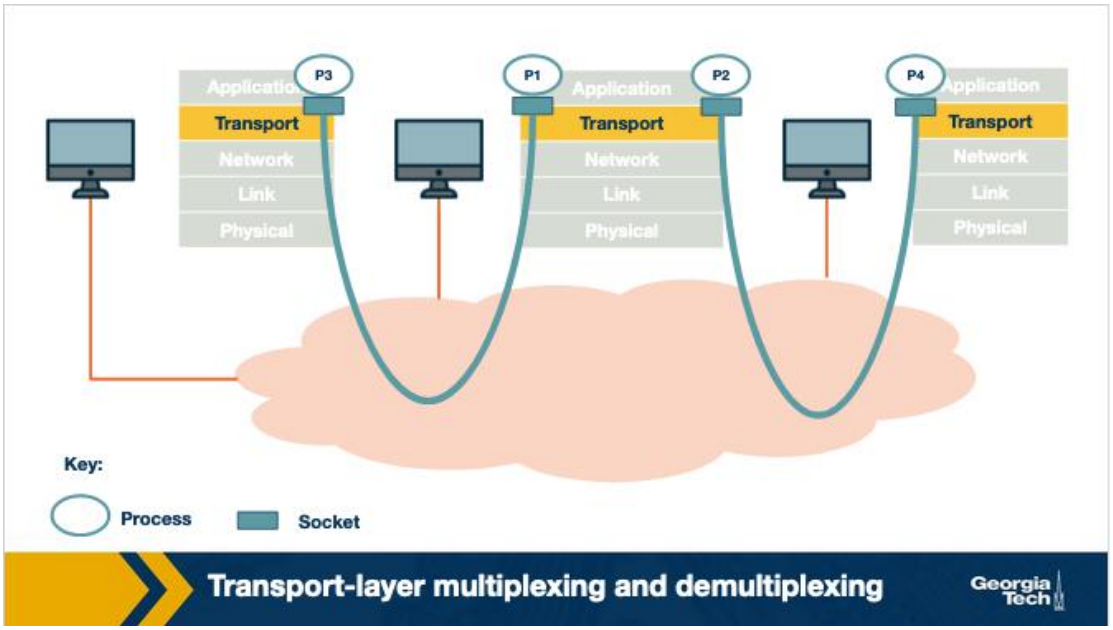
The transport layer solves this problem, by using additional identifiers known as ports. Each application binds itself to a unique port number by opening sockets and listening for any data from a remote application. Thus, the transport layer can do multiplexing by using ports.

There are two ways in which we can use multiplexing. Connectionless and connection oriented multiplexing. As the name suggests, it depends if we have a connection established between the sender and the receiver or not.

In the next topic we are looking into multiplexing and demultiplexing.

Connection Oriented and Connectionless Multiplexing and Demultiplexing

In this topic, we will talk about multiplexing and demultiplexing.



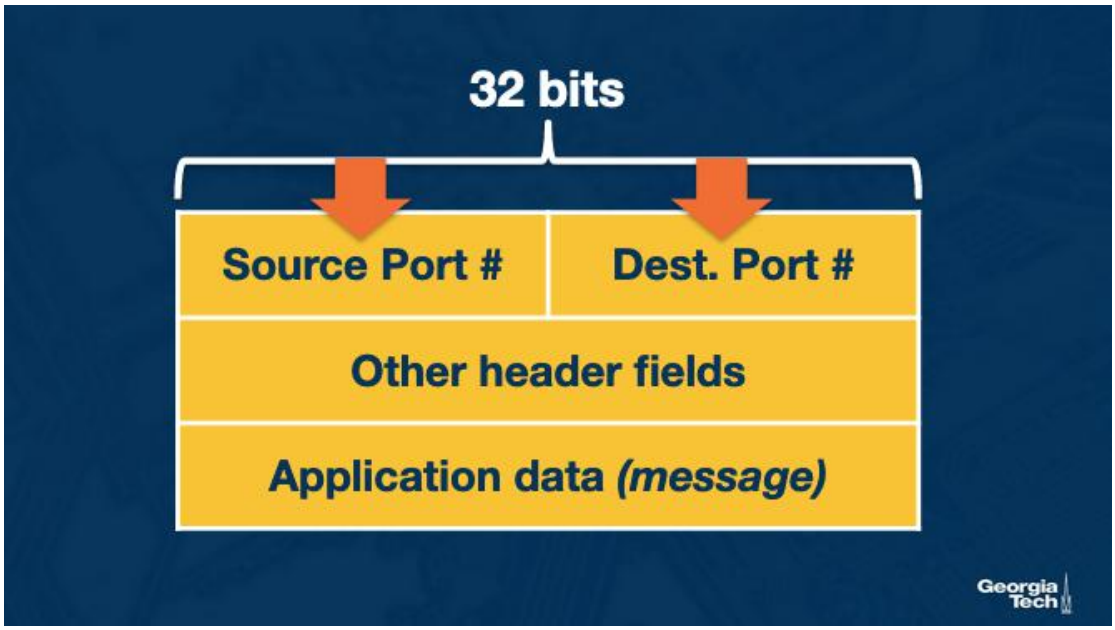
Let's consider the scenario shown in the figure above which includes three hosts running an application. A receiving host that receives an incoming transport-layer segment will forward it to the appropriate socket. The receiving host identifies the appropriate socket by examining a set of fields in the segment.

The job of delivering the data that are included in a transport-layer segment to the appropriate socket, as defined in the segment fields, is called **demultiplexing**.

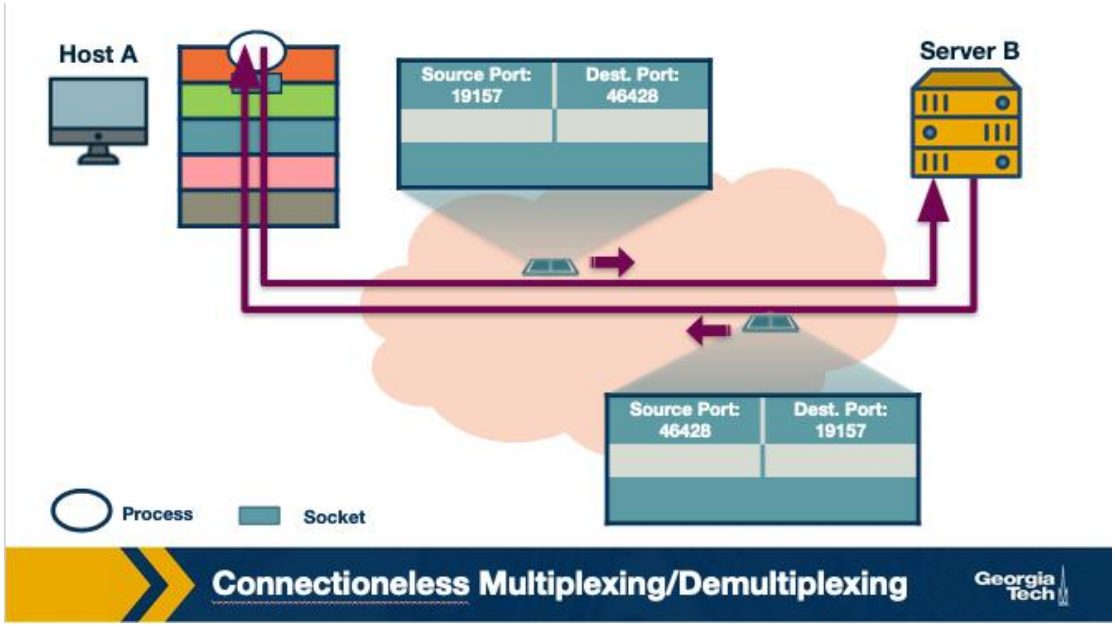
Similarly, the sending host will need to gather data from different sockets, and encapsulate each data chunk with header information (that will later be used in demultiplexing) to create segments, and then forward the segments to the network layer. We refer to this job as **multiplexing**.

As an example, let's take a closer look at the host in the middle. The transport layer in the middle host, will need to demultiplex the data arriving from the network layer to the correct socket (P1 or P2). Also, the transport layer in the middle host, will need to perform multiplexing, by collecting the data from sockets P1 or P2, then by generating transport-layer segments, and then finally by forwarding these segments to the network layer below.

Now, let's focus at the socket identifiers: The sockets are identified based on special fields (shown below) in the segment such as the **source port number field** and the **destination port number field.**



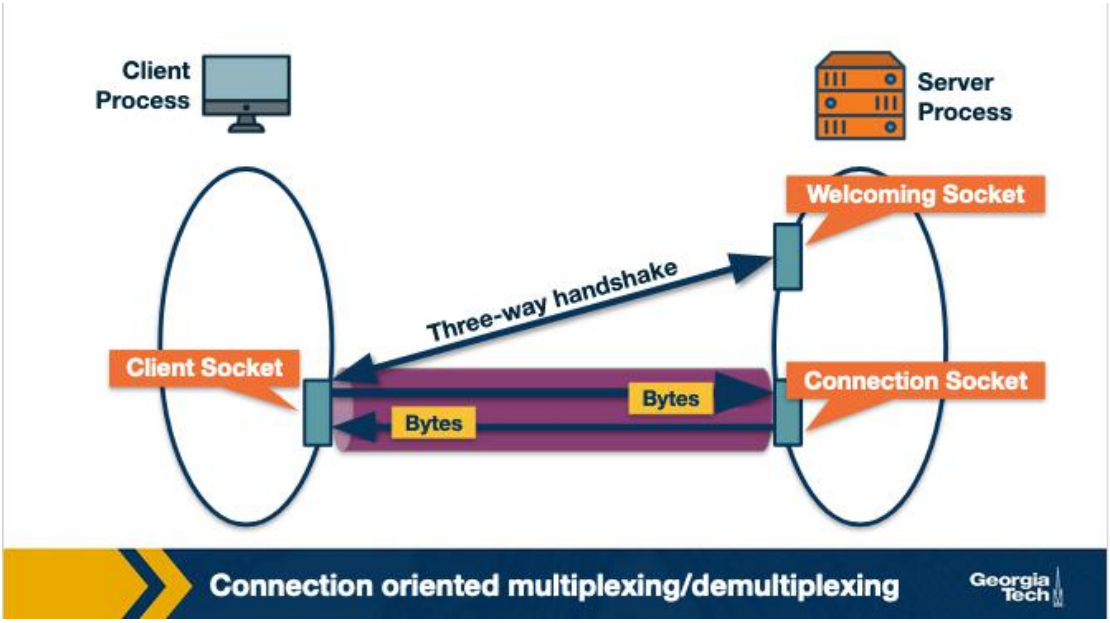
We have two flavors of multiplexing/demultiplexing: the connectionless and connection oriented.



First, we will talk about the connectionless multiplexing and demultiplexing. The identifier of a UDP socket is a two-tuple that is consisted of a destination IP address and a destination port number. Consider two hosts, A and B, which are running two processes at UDP ports a and b respectively. Let's suppose that host A sends data to host B. The transport layer in host A creates transport layer segment by the application data, the source port and the destination port, and forwards the segment to the network layer. In turn the network layer encapsulates the segment into a network-layer datagram and sends it to host B with best effort delivery. Let's suppose that the datagram is successfully received by host B. Then the transport layer at host B, identifies the correct socket by looking at the field of the destination port. In case that host B runs multiple processes, each process will have its own own UDP socket and therefore a distinct associated port number. Host B will use this information to demultiplex receiving data to the correct socket. If Host B receives UDP segments with *destination* port number, it will forward the segments to the same destination process via the same

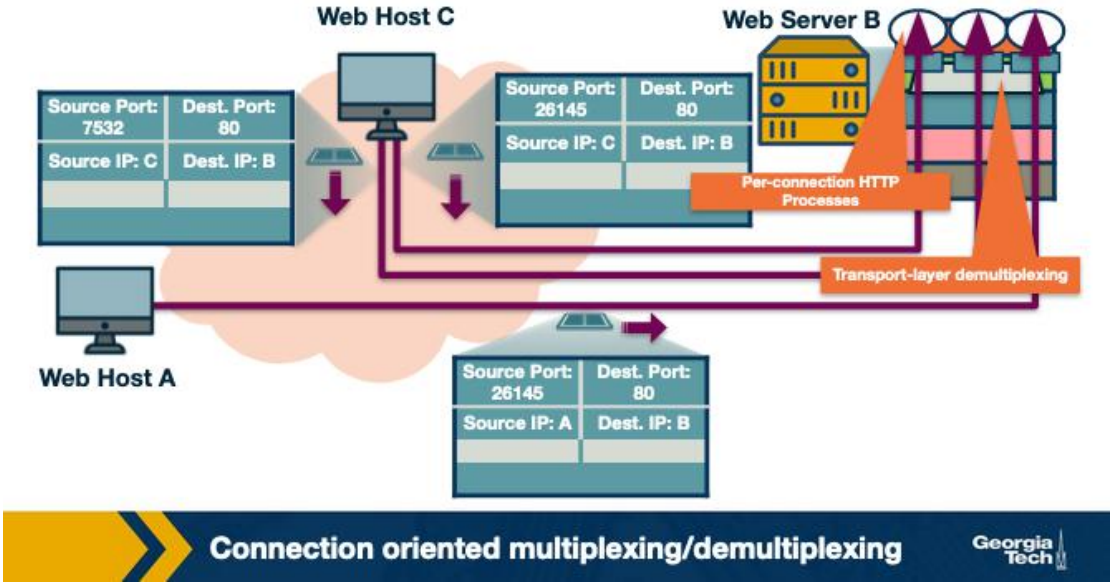
destination socket, even if the segments are coming from different source hosts and/or different source port numbers.

Now let’s consider the connection oriented multiplexing and demultiplexing.



The identifier for a TCP socket is a four tuple that is consisted by the source IP, source port, destination IP and destination port. Let’s consider the example of a TCP client server as shown in the figure 2.29. The TCP server has a listening socket that waits for connections requests coming from TCP clients. A TCP client creates a socket and sends a connection request, which is a TCP segment that has a source port number chosen by the client, a destination port number 12000 and a special connection-establishment bit set in the TCP header. Finally, the TCP server receives the connection request, and the server creates a socket that is identified by the four-tuple source IP, source port, destination IP and destination port. The server uses this socket identifier to demultiplex incoming data and forward them to this socket. Now, the TCP connection is established and the client and server can send and receive data between one another.

Example: Let's look at an example connection establishment.



In this example, we have three hosts A, B and C. Host C and A initiate two and one HTTP sessions to server B, respectively. Hosts C and A assign port numbers to their connections independently of one another. Host C assigns port numbers 26145 and 7532. In case Host A assigns the same port number as C, host B will still be able to demultiplex incoming data from the two connections because the connections are associated with different source IP addresses.

Let's add a final note about web servers and persistent HTTP. Let's assume, we have a webserver listening for connection requests at port 80. Clients send their initial connection requests and their subsequent data with destination port 80. The webserver is able to demultiplex incoming data based on their unique source IP addresses and source port numbers. The client and the server maybe persistent HTTP, in which case, they exchange HTTP messages via the same server socket. The client and the server maybe using non-persistent HTTP, where for every request and response, a new TCP connection and a new socket are created and closed for every response/request. In the second case, a busy webserver may experience severe performance impact.

A word about the UDP protocol

This lecture is primarily focused on TCP. Before exploring more topics on the TCP protocol let's briefly talk about UDP.

UDP is: a) an unreliable protocol as it lacks the mechanisms that TCP has in place and b) a connectionless protocol that does not require the establishment of a connection (eg threeway handshake) before sending packets.

The above description doesn't sound so promising... so why do we have UDP at the first place? Well, it turns out that it is exactly the lack of those mechanisms that make UDP more desirable in some cases.

Specifically UDP offers less delays and better control over sending data because with UDP we have:

1. **No congestion control or similar mechanisms.** With UDP, as soon as the application passes data to the transport layer, then UDP encapsulates it and sends it over to the network layer. In contrast TCP "intervenes" a lot

with sending the data e.g. with the congestion control mechanism or the retransmissions in case an ACK is not received. These TCP mechanisms cause further delays.

2. **No connection management overhead.** With UDP we have no connection establishment and no need to keep track of connection state (eg with buffers). Both mean even less delays.

So with some real time applications that are sensitive to delays UDP is a better option, despite possibly higher losses. Eg DNS is using UDP. Which other applications prefer UDP over TCP? The table below gives us an idea:

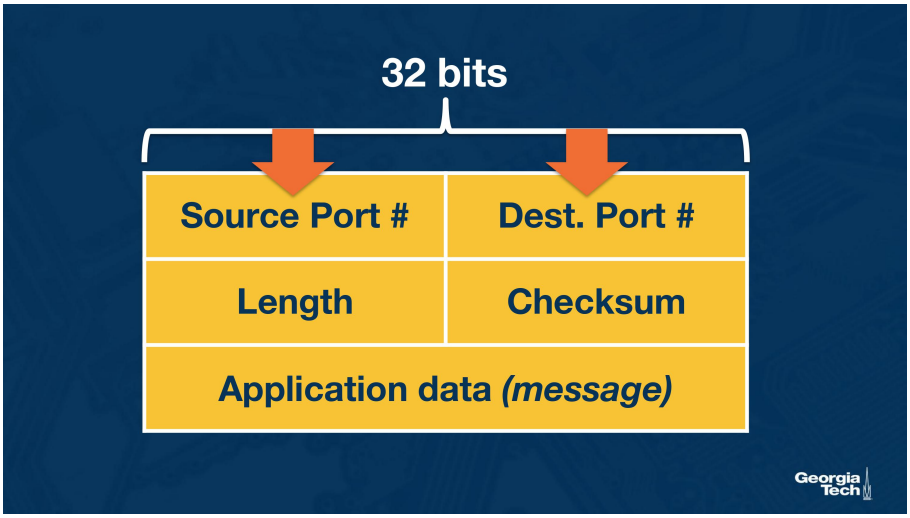
Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	typically proprietary	UDP or TCP
Internet telephony	typically proprietary	UDP or TCP
Network management	SNMP	Typically UDP
Routing protocol	RIP	Typically UDP
Name translation	DNS	Typically UDP

Popular Internet Applications and Transport Protocols

Georgia Tech

The UDP packet structure: UDP has a 64 bits header consisting of the following fields:

1. Source and destination ports.
2. Length of the UDP segment (header and data).
3. Checksum (an error checking mechanism). Since there is no guarantee for link-by-link reliability, we need a basis mechanism in place for error checking. The UDP sender adds the src port, the dest port and the packet length. Then it takes the sum and performs an 1s complement (all 0s are turned to 1 and all 1s are turned to 0s). If during the sum there is an overflow, its wrapped around. The receiver adds all the four 16-bit words (including the checksum). The result should be all 1s unless an error has occurred.



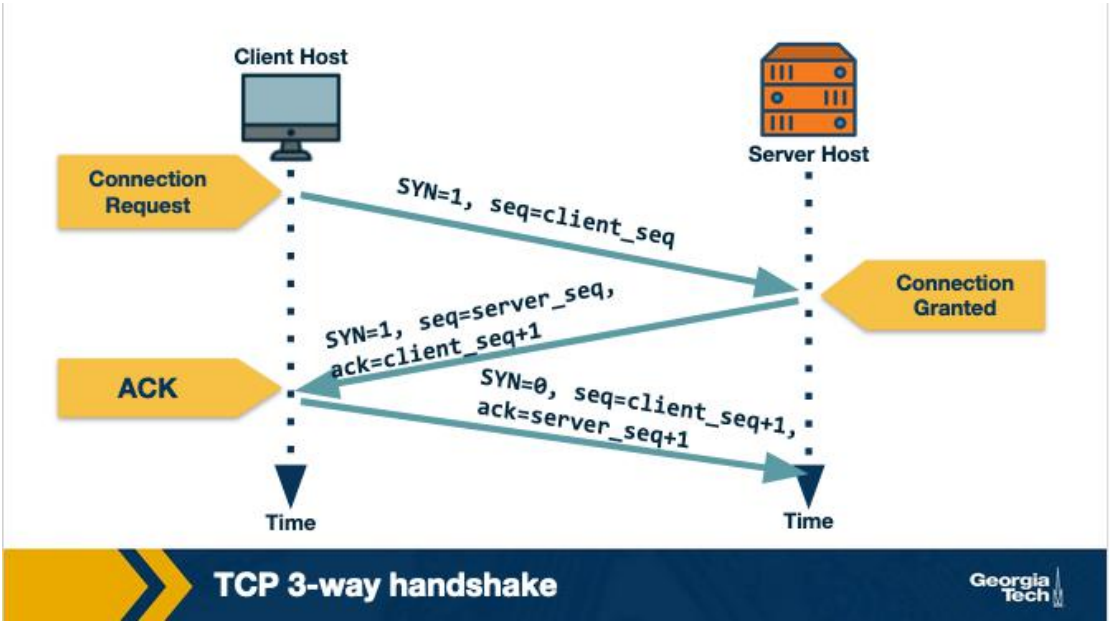
The TCP Three-Way Handshake

The TCP Three way Handshake:

Step 1: The TCP client sends a special segment, (containing no data) and with SYN bit set to 1. The Client also generates an initial sequence number (client_isn) and includes it in this special TCP SYN segment.

Step 2: The Server upon receiving this packet, allocates the required resources for the connection and sends back the special 'connection-granted' segment which we call SYNACK. This packet has SYN bit set to 1, ack field containing (client_isn+1) value and a randomly chosen initial sequence number in the sequence number field.

Step 3: When the client receives the SYNACK segment, it also allocates buffer and resources for the connection and sends an acknowledgment with SYN bit set to 0.



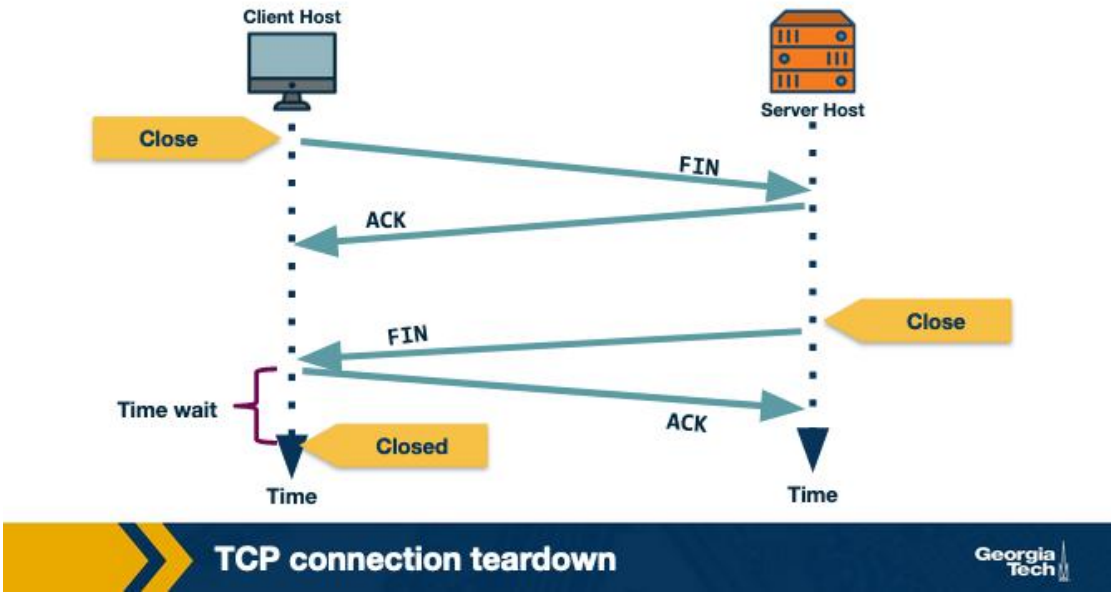
Connection Teardown:

Step 1: When client wants to end the connection, it sends a segment with FIN bit set to 1 to the server.

Step 2: Server acknowledges that it has received the connection closing request and is now working on closing the connection.

Step 3: The Server then sends a segment with FIN bit set to 1, indicating that connection is closed.

Step 4: The Client sends an ACK for it to the server. It also waits for sometime to resend this acknowledgment in case the first ACK segment is lost.



Reliable Transmission

What is reliable transmission? Recall that the network layer is unreliable and it may lead to packets getting lost or arriving out of order. This can clearly be an issue for a lot of applications. For example, a file downloaded over the Internet might be corrupted if some of the packets were lost during the transfer.

One option here is to allow the application developers take care of the network losses as is done in UDP. However, given that reliability is an important primitive desirable for a lot of applications, TCP developers decided to implement this primitive in the transport layer. **Thus, TCP guarantees an in-order delivery of the application-layer data without any loss or corruption.**

Now, let us look at how TCP implements reliability.

In order to have a reliable communication, the sender should be able to know which segments were received by the remote host and which were lost. Now, how can we achieve this? One way to do this is by having the receiver send acknowledgements indicating that it has successfully received the specific segment. In case the sender does not receive an acknowledgement within a given period of time, the sender can assume the packet is lost and resend it. This method of using acknowledgements and timeouts is also known as **Automatic Repeat Request or ARQ.**

There are various methods in which it can be implemented:

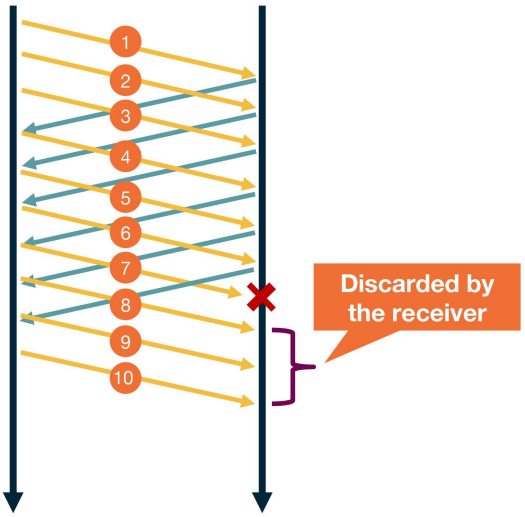
The simplest way would be for the sender to send a packet and wait for its acknowledgement from the receiver. This is known as **Stop and Wait ARQ**. Note that the algorithm typically needs to figure out the waiting time after which it resends the packet and this estimation can be tricky. A small value of timeout can lead to unnecessary re-transmissions and a large value can lead to unnecessary delays. In most cases, the timeout value is a function of the estimated round trip time of the connection.

Clearly, this kind of alternate sending and waiting for acknowledgement has a very low performance. In order to solve this problem, the sender can send multiple packets without waiting for acknowledgements. More specifically, the sender is allowed to send at most N unacknowledged packets typically referred to as the window size. As it receives acknowledgement from the receiver, it is allowed to send more packets based on the window size. In implementing this, we need to take care of the following concerns:

- The receiver needs to be able to identify and notify the sender of a missing packet. Thus, each packet is tagged with a unique byte sequence number which is increased for subsequent packets in the flow based on the size of the packet.
- Also, now both sender and receiver would need to buffer more than one packet. For instance, the sender would need to buffer packets that have been transmitted but not acknowledged. Similarly, the receiver may need to buffer the packets because the rate of consuming these packets (say writing to a disk) is slower than the rate at which packets arrive.

Now let's look at how does the receiver notify the sender of a missing segment.

One way is for the receiver to send an ACK for the most recently received in-order packet. The sender would then send all packets from the most recently received in-order packet, even if some of them had been sent before. The receiver can simply discard any out-of-order received packets. This is called **Go-back-N.** For instance, in the figure below if packet 7 was lost in the network, the receiver will discard any subsequent packets. The sender will send all the

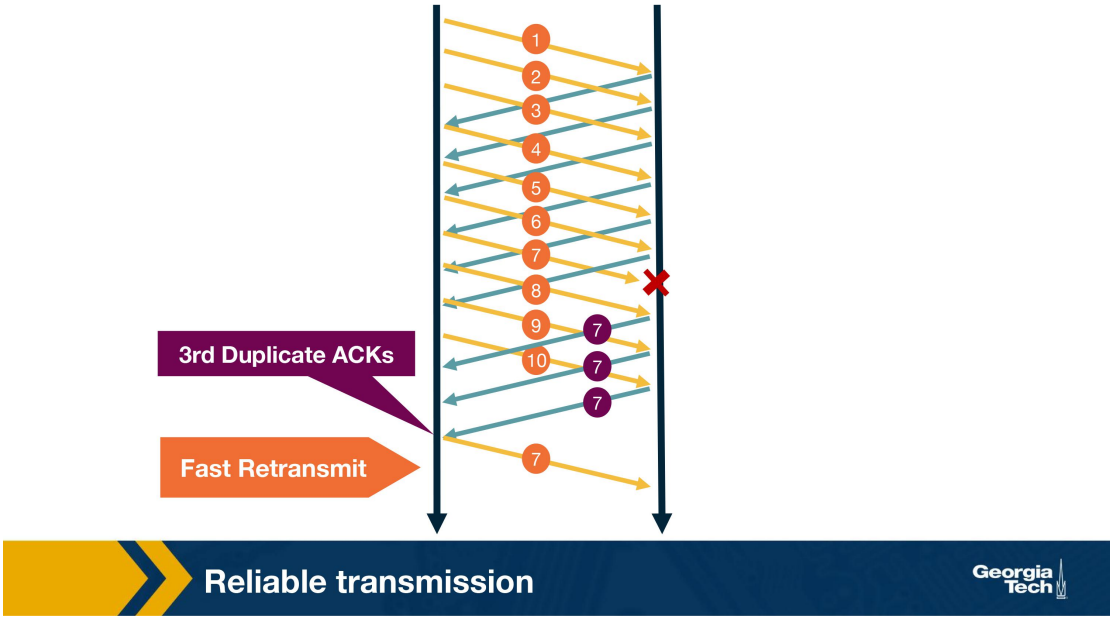


packets starting from 7 again.

Clearly, in the above case, a single packet error can cause a lot of unnecessary retransmissions. To solve this, TCP uses **selective ACK**ing. In this, the sender retransmits only those packets that it suspects were received in error. The receiver in this case would acknowledge a correctly received packet even if it is not in order. The out-of-order packets are buffered until any missing packets have been received at which point the batch of the packets can be delivered to the application layer.

Note that even in this case, TCP would need to use a timeout as there is a possibility of ACKs getting lost in the network.

In addition to using timeout to detect loss of packets, TCP also uses duplicate acknowledgements as a means to detect loss. A duplicate ACK is additional acknowledgement of a segment for which the sender has already received acknowledgment earlier. When the sender receives 3 duplicate ACKs for a packet, it considers the packet to be lost and will retransmit it instead of waiting for the timeout. This is known as **fast retransmit.** For example, in the figure below, once sender receives 3 duplicate ACKs, it will retransmit packet 7 without waiting for timeout.



Transmission Control

In this topic we will learn about the mechanisms provided in the transport-layer to control the transmission rate.

Why control the transmission rate? We will first illustrate why we need to know and adapt the transmission rate. Consider a scenario when user A needs to send 1 Gb of file to a remote host B on a 100 Mbps link. What rate should it send the file? One could say that it should be 100 Mbps. But how does user A determine that given it does not know the link capacity. Also, what about other users that also would be using the same link? What happens to the sending rate if the receiver B is also receiving files from a lot of other users? Finally, which layer in the network decides the data transmission rate? In this section, we will try to answer all these questions.

Where should the transmission control function reside in the network stack? One option is to let the application developers figure out and implement mechanisms for transmission control. This is what UDP does. However, it turns

out that transmission control is a fundamental function for most of the applications. Thus it will be easier if it is implemented in the transport layer. Moreover, it also has to deal with issues of fairness in using the network as we will see later, thus making it more convenient to handle it at the transport layer. Thus, TCP provides mechanisms for transmission control which have been a subject of interest to network researchers since the inception of computer networking. We will look at these in detail now.

Flow Control

Flow control: Controlling the transmission rate to protect the receiver's buffer

The first case where we need transmission control is to protect the buffer of the receiver from overflowing. Recall that TCP uses a buffer at the receiver end to buffer packets that have not been transmitted to the application. It could happen that the receiver is involved with multiple processes and does not read the data instantly. This can cause accumulation of huge amount of data and overflow the receive buffer.

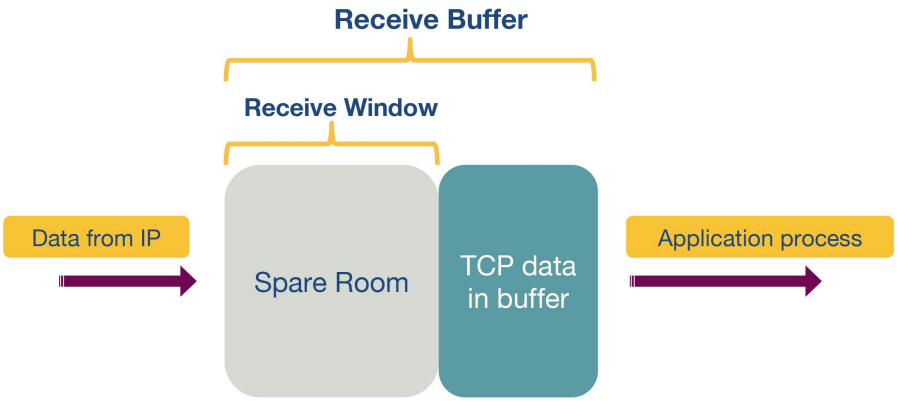
TCP provides a rate control mechanism also known as flow control that helps match the sender's rate against the receiver's rate of reading the data. Sender maintains a variable 'receive window'. It provides sender an idea of how much data the receiver can handle at the moment.

We will illustrate its working using an example. Consider two hosts, A and B, that are communicating with each other over a TCP connection. Host A wants to send a file to Host B. For this, Host B allocates a receive buffer of size **RcvBuffer** to this connection. The receiving host maintains two variables, **LastByteRead** (number of byte that was last read from the buffer) and **LastByteRcvd** (last byte number that has arrived from sender and placed in the buffer). Thus, in order to not overflow the buffer, TCP needs to make sure that

LastByteRcvd - LastByteRead <= RcvBuffer

The extra space that the receive buffer has, is specified using a parameter, termed as receive window.

rwnd =



Flow control

RcvBuffer - [LastByteRcvd - LastByteRead]

The receiver advertises this value of *rwnd* in every segment/ACK it sends back to the sender.

The sender also keeps track of two variables, **LastByteSent** and **LastByteAcked**.

UnAcked Data Sent = LastByteSent - LastByteAcked

To not overflow the receiver's buffer, the sender needs to make sure that the maximum number of unacknowledged bytes it sends are no more than the rwnd.

Thus we need:

LastByteSent – LastByteAcked <= rwnd

Caveat: However, there is one scenario where this scheme has a problem. Consider a scenario, if the receiver had informed the sender that rwnd = 0, and thus the sender stops sending data. Also, assume that B has nothing to send to A. Now, as the application processes the data at the receiver, the receiver buffer is cleared but the sender may never know that new buffer space is now available and will be blocked from sending data even when receiver buffer is empty.

TCP resolves this problem by making sender continue sending segments of size 1 byte even after when rwnd = 0. When the receiver acknowledges these segments, it will specify the rwnd value and the sender will know as soon as the receiver has some room in the buffer.

Congestion Control Introduction

Congestion control: Controlling the transmission rate to protect the network from congestion

The second and very important reason for transmission control is to avoid congestion in the network.

Let us look at an example to understand this. Consider a set of senders and receivers sharing a single link with capacity C. Assume, other links have capacity > C. How fast should each sender transmit data? Clearly, we do not want the combined transmission rate to be higher than the capacity of the link as it can cause issues in the network such as longer queues, packet drops etc. Thus, we want a mechanism to control the transmission rate at the sender in order to avoid congestion in the network. This is known as congestion control.

It is important to note that networks are quite dynamic with users joining and leaving the network, initiating data transmission and terminating existing flows. Thus the mechanisms for congestion control need to be dynamic enough to adapt to these changing network conditions.

What are the goals of congestion control?

Let us consider some of the desirable properties of a good congestion control algorithm:

- **Efficiency**. We should get high throughput or utilization of the network should be high.
- **Fairness.** Each user should its fair share of the network bandwidth. The notion of fairness is dependent on the network policy. For this context, we will assume that every flow under the same bottleneck link should get equal bandwidth.
- **Low delay**. In theory, it is possible to design protocols that have consistently high throughput assuming infinite buffer. Essentially, we could just keep sending the packets to the network and they will get stored in the buffer and will eventually get delivered. However, it will lead to long queues in the network leading to delays. Thus, applications that are sensitive to network delays such as video conferencing will suffer. Thus, we want the network delays to be small.
- **Fast convergence.** The idea here is that a flow should be able to converge to its fair allocation fast. This is important as a typical network's workload is composed a lot of short flows and few long flows. If the convergence to fair share is not fast enough, the network will still be unfair for these short flows.

Congestion control flavors: E2E vs Network-assisted

Broadly speaking, there can be two approaches to implement congestion control:

The first approach is network-assisted congestion control. In this we rely on the network layer to provide explicit feedback to the sender about congestion in the network. For instance, routers could use ICMP source quench to notify the source that the network is congested. However, under severe congestion, even the ICMP packets could be lost, rendering the network feedback ineffective.

The second approach is to implement end-to-end congestion control. As opposed to the previous approach, the network here does not provide any explicit feedback about congestion to the end hosts. Instead, the hosts infer congestion from the network behavior and adapt the transmission rate.

Eventually, TCP ended up using the end-to-end approach. This largely aligns with the end-to-end principle adopted in the design of the networks. Congestion control is a primitive provided in the transport layer, whereas routers operate at the network layer. Therefore, the feature resides in the end nodes with no support from the network. Note that this is no longer true as certain routers in the modern networks can provide explicit feedback to the end-host by using protocols such as ECN and QCN.

Let us now look at how TCP can infer congestion from the behavior of the network.

How a host infers congestion? Signs of congestion

There are mainly two signals of congestion.

First is the packet delay. As the network gets congested, the queues in the router buffers build up. This leads to increased packet delays. Thus, an increase in the round trip time, which can be estimated based on ACKs, can be an indicator of congestion in the network. However, it turns out that packet delay in a network tend to be variable, making delay-based congestion inference quite tricky.

Another signal for congestion is packet loss. As the network gets congested, routers start dropping packets. Note that packets can also be lost due to other reasons such as routing errors, hardware failure, TTL expiry, error in the links, or flow control problems, although it is rare.

The earliest implementation of TCP ended up using loss as a signal for congestion. This is mainly because TCP was already detecting and handling packet losses to provide reliability.

How does a TCP sender limit the sending rate?

The idea of TCP congestion control was introduced so that each source can determine the network's available capacity and know how many packets it can send without adding to the network's level of congestion. Each source uses ACKs as a pacing mechanism. Each source uses the ACK to determine if the packet released earlier to the network was received by the receiving host and it is now safe to release more packets into the network.

TCP uses a congestion window which is similar to the receive window used for flow control. It represents the maximum number of unacknowledged data that a sending host can have in transit (sent but not yet acknowledged).

TCP uses a probe-and-adapt approach in adapting the congestion window. Under regular conditions, TCP increases the congestion window trying to achieve the available throughput. Once it detects congestion then the congestion window is decreased.

In the end, the number of unacknowledged data that a sender can have is the minimum of the congestion window and the receive window.

LastByteSent – LastByteAcked <= min{cwnd, rwnd}

In a nutshell, a TCP sender cannot send faster than the slowest component, which is either the network or the receiving host.

Congestion control at TCP - AIMD

TCP decreases the window when the level of congestion goes up, and it increases the window when the level of congestion goes down. We refer to this combined mechanism as additive increase/multiplicative decrease (AIMD).

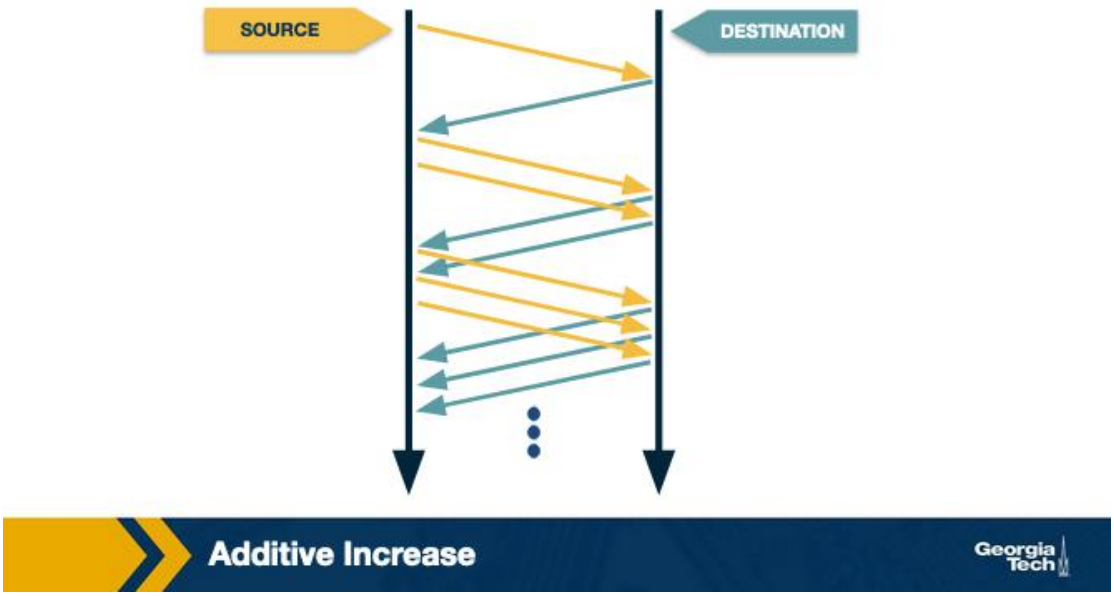
Additive Increase:

The connection starts with a constant initial window, typically 2 and increases it additively. The idea behind additive increase is to increase the window by one packet every RTT (Round Trip Time). So, in the additive increase part of the AIMD, every time the sending host successfully sends a cwnd number of packets it adds 1 packet to cwnd.

Also, in practice, this increase in AIMD happens incrementally. TCP doesn't wait for ACKs of all the packets from the previous RTT. Instead, it increases the congestion window size as soon as each ACK arrives. In bytes, this increment is a portion of the MSS (Maximum Segment Size).

Increment = MSS × (MSS / CongestionWindow)

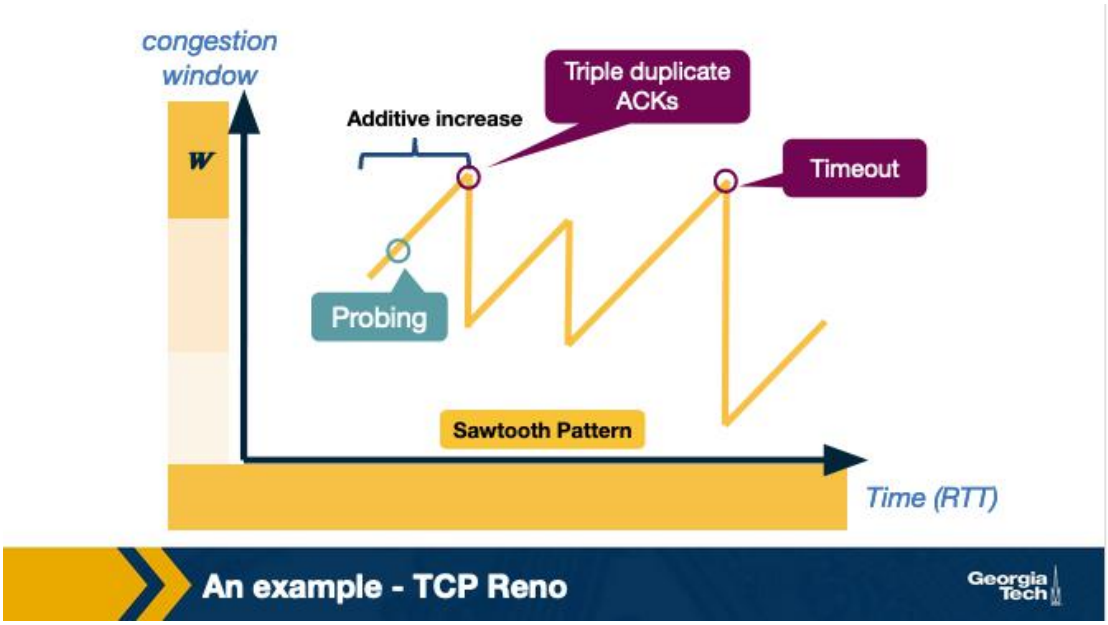
CongestionWindow + = Increment



Multiplicative Decrease:

Once TCP Reno detects congestion, it reduces the rate at which the sender transmits. So, when the TCP sender detects that a timeout occurred, then it sets the CongestionWindow (cwnd) to half of its previous value. This decrease of the cwnd for each timeout corresponds to the "multiplicative decrease" part of AIMD. For example, suppose the cwnd is currently set to 16 packets. If a loss is detected, then cwnd is set to 8. Further losses would result to the cwnd to be reduced to 4 and then to 2 and then to 1. The value of cwnd cannot be reduce further than 1 packet.

Figure below shows an example of how the congestion control window decreases when congestion is detected:



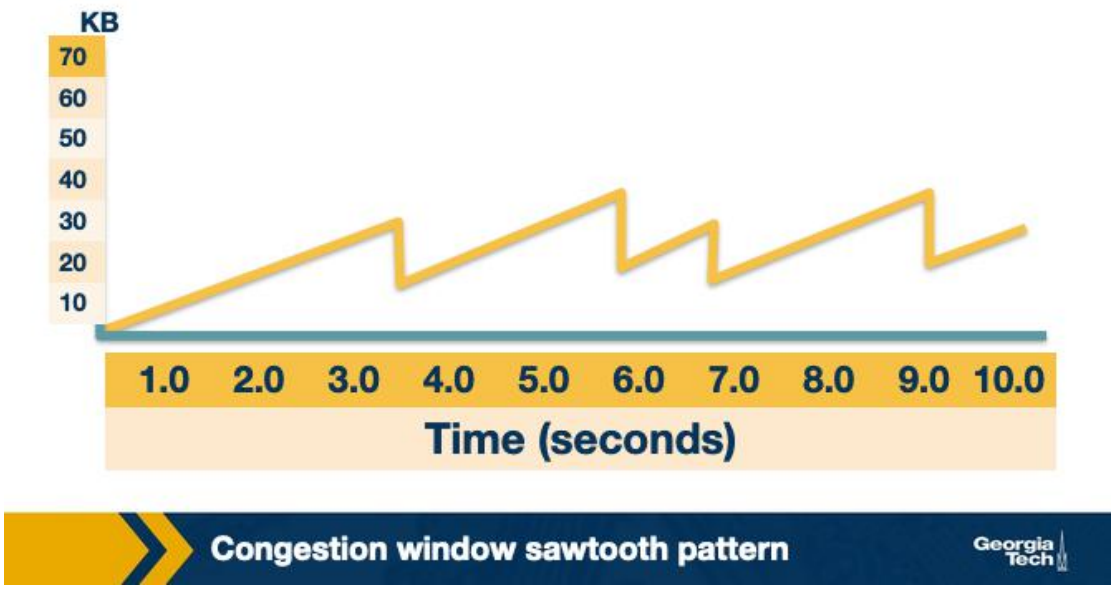
Signals of congestion:

TCP Reno uses two types of packet loss detection as a signal of congestion. First is the triple duplicate ACKs and is considered to be mild congestion. In this case, the congestion window is reduced to half of the original congestion window.

The second kind of congestion detection is timeout i.e. when no ACK is received within a specified amount of time. It is considered a more severe form of congestion, and the congestion window is reset to the Initial Window.

Congestion window sawtooth pattern:

TCP continually decreases and increases the congestion window throughout the lifetime of the connection. If we plot the cwnd with respect to time, we observe that it follows a sawtooth pattern as shown in the figure:



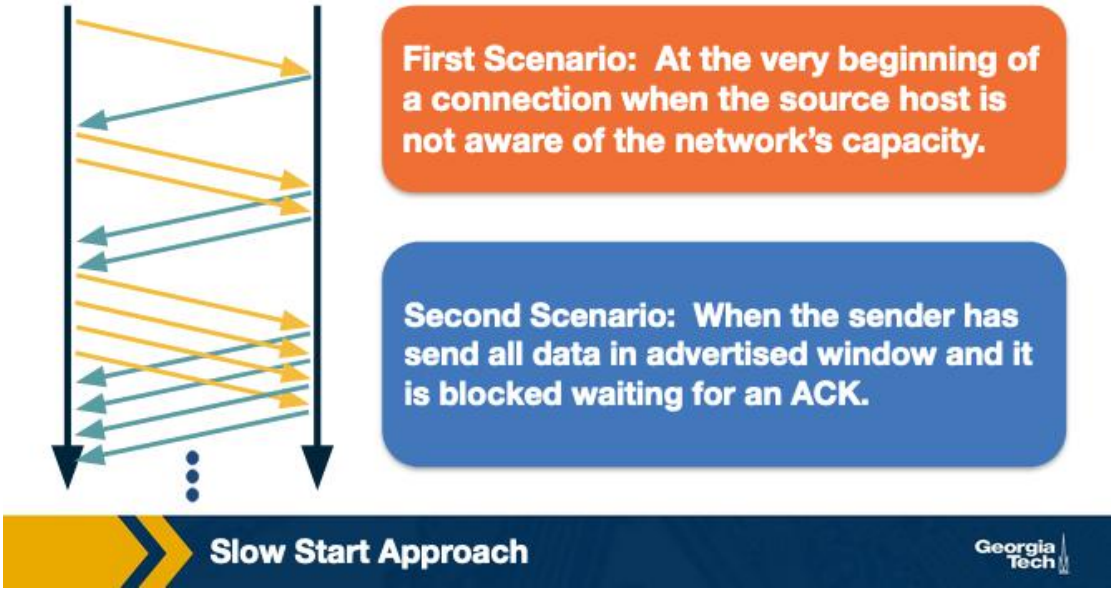
Slow start in TCP

The AIMD approach we saw in the previous topic is useful when the sending host is operating very close to the network capacity. AIMD approach reduces the congestion window at a much faster rate than it increases the congestion window. The main reason for this approach is that the consequences of having too large a window are much worse than those of it being too small. For example, when the window is too large, more packets will be dropped and retransmitted, making network congestion even worse; thus, it is important to reduce the number of packets being sent into the network as quickly as possible.

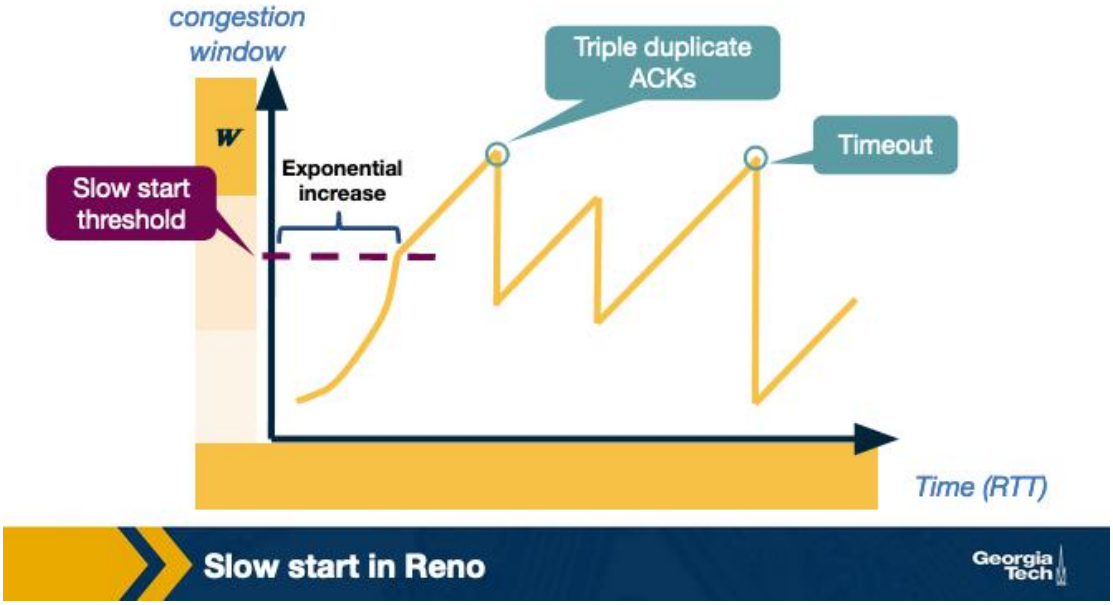
In contrast, when we have a new connection that starts from cold start, it can take much longer for the sending host to increase the congestion window by using AIMD. So for a new connection, we need a mechanism which can rapidly increase the congestion window from a cold start.

To handle this, TCP Reno has a **slow start phase** where the congestion window is increased exponentially instead of linearly as in the case of AIMD. The source host starts by setting cwnd to 1 packet. When it receives the ACK for this packet, it adds 1 to the current cwnd and sends 2 packets. Now when it receives the ACK for these two packets, it adds 1 to cwnd for each of the ACK it receives and sends 4 packets. Once the congestion window becomes more than a threshold, often referred to as **slow start threshold**, it starts using AIMD.

The figure below shows the sending host during slow start.



The figure below shows an example of the slow start phase.



Slow start is called "slow" start despite using an exponential increase because in the beginning it sends only one packet and starts doubling it after each RTT. Thus, it is slower than starting with a large window.

Finally, we note that there is one more scenario, where slow start kicks in. When a connection dies while waiting for a timeout to occur. This happens when the source has sent enough data as allowed by the flow control mechanism of TCP and times out while waiting for the ACK which will not arrive. Thus, the source will eventually receive a cumulative

ACK that will reopen the connection and then instead of sending the available window size worth of packets at once, it will use slow start mechanism.

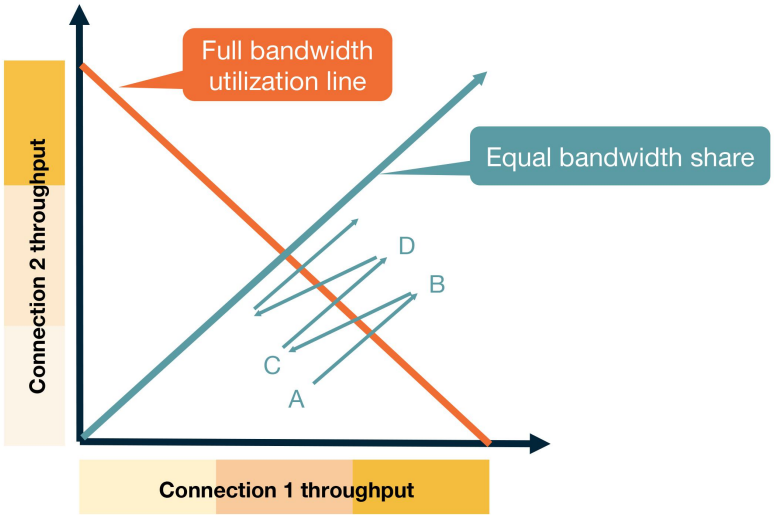
In this case, the source will have a fair idea about the congestion window from the last time it had a packet loss. It will now use this information as the “target” value to avoid packet loss in future. This target value is stored in a temporary variable “CongestionThreshold”. Now, source performs slow start by doubling the number of packets after each RTT until cwnd value reaches the congestion threshold (a knee point). After this point, it increases the window by 1 (additive increase) each RTT until it experiences packet loss (cliff point). After which it multiplicatively decreases the window.

TCP Fairness

Recall that we defined fairness as one of the desirable goals of congestion control. Note that fairness in this case means that for k-connections passing through one common link with capacity R bps, each connection gets an average throughput of R/k.

Let us understand if TCP is fair.

Consider a simple scenario where two TCP connections share a single link with bandwidth R. For simplicity, we assume that both connections have same RTT and there are only TCP segments passing through the link. If we plot a graph for throughput of these two connections, then the throughput for each should sum up to R. So, the goal is to get throughput achieved for each link fall somewhere near the intersection of the equal bandwidth share line and the full bandwidth utilization line, as shown in below graph:



Throughput realized by TCP connections 1 & 2

At point A in the above graph, total utilized bandwidth is less than R, so no loss can occur at this point. Therefore, both the connection will increase their window size, thus the sum of the utilized bandwidth will grow and graph will move towards B.

At point B, as the total transmission rate is more than R, both connection may start having packet loss. Now they will decrease their window size to half and come back to point C.

At point C, again the total throughput is less than R, so both connection will increase their window size to move towards point D and will again experience packet loss at D, and so on.

Thus, using AIMD leads to fairness in bandwidth sharing.

Caution about fairness

There can be cases when TCP is not fair.

One such case arises due to the difference in the RTT of different TCP connections. Recall that TCP Reno uses ACK-based adaptation of the congestion window. Thus, connections with smaller RTT values would increase their congestion window faster than the ones with longer RTT values. This leads to an unequal sharing of the bandwidth.

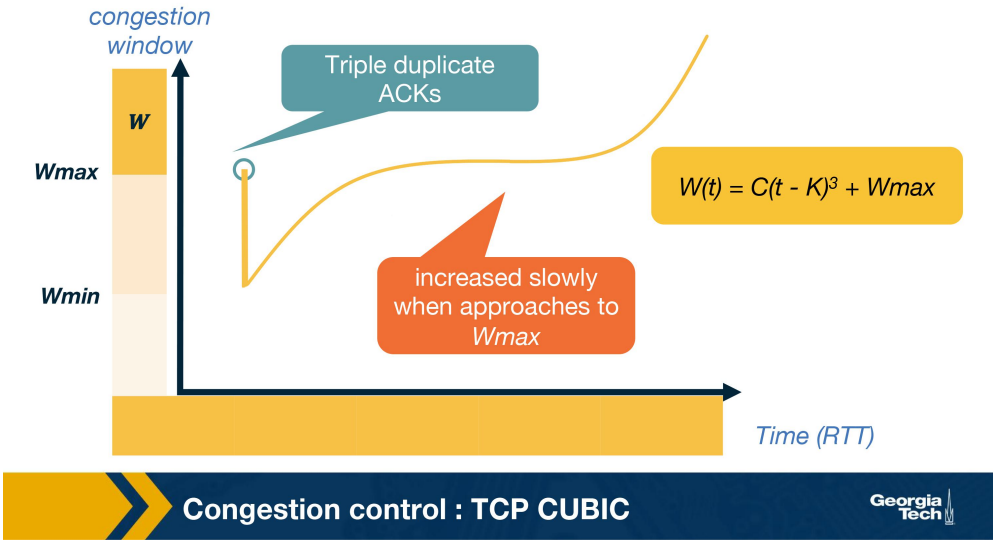
Another case of unfairness arises if a single application uses multiple parallel TCP connections. Consider, for example, nine applications using one TCP connection sharing a link of rate R. If a new application establishes connection on the same link and also uses one TCP connection, then each application gets fairly the same transmission rate of R/10. But if the new application had 11 parallel TCP connections, then it would get an unfair allocation of more than R/2.

Congestion Control in Modern Network Environments: TCP CUBIC

Over the years, networks have improved with link speeds increasing tremendously. This has called for changes in the TCP congestion control mechanisms mainly with a desire to improve link utilization.

We can see that TCP Reno has low network utilization, especially when the network bandwidth is high or the delay is large. Such networks are also known as high bandwidth delay product networks.

To make TCP more efficient under such networks, many improvements to TCP congestion control have been proposed. Now we will look at one such version, called TCP CUBIC, which was also implemented in the Linux kernel. It uses a CUBIC polynomial as the growth function.



Let us see what happens when TCP experiences a triple duplicate ACK, say at window=Wmax. This could be because of congestion in the network. To maintain TCP-fairness, it uses a multiplicative decrease and reduces the window to half. Let us call this Wmin.

Now, we know that the optimal window size would be in between Wmin and Wmax and closer to Wmax. So, instead of increasing the window size by 1, it is okay to increase the window size aggressively in the beginning. Once the W approaches closer to Wmax, it is wise to increase it slowly because that is where we detected a packet loss last time. Assuming no loss is detected this time around Wmax, we keep on increasing the window a little bit. If there is no loss still, it could be that the previous loss was due to a transient congestion or non-congestion related event. Therefore, it is okay to increase the window size with higher values now.

This window growth idea is approximated in TCP CUBIC using a cubic function. Here is the exact function it uses for the window growth:

$$W(t) = C(t-K)^3 + W_{max}$$

Here, Wmax is the window when the packet loss was detected. Here C is a scaling constant, and K is the time period that the above function takes to increase W to Wmax when there is no further loss event and is calculated by using the following equation:

$$K = \sqrt[3]{\frac{W_{max}\,\beta}{C}}$$

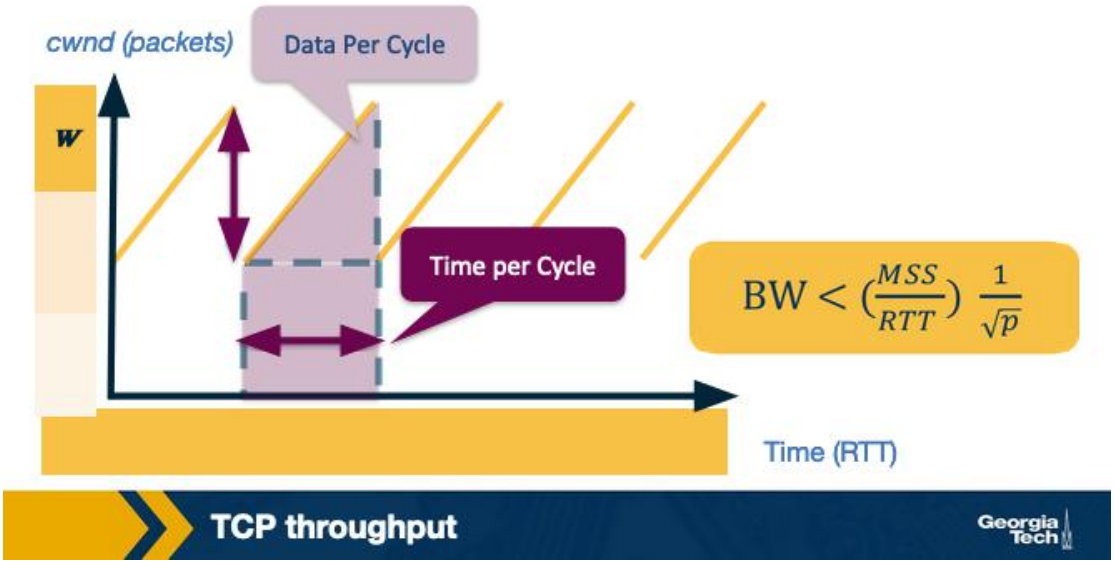
It is important to note that time here is the time elapsed since the last loss event instead of the usual ACK-based timer used in TCP Reno. This also makes TCP CUBIC RTT-fair.

The TCP Protocol: TCP Throughput

In a previous topic, we saw that the congestion window follows a sawtooth pattern. As shown in this Figure. The congestion window is increased by 1 packet every RTT, until it reaches the maximum value W, at which point a loss is detected and the cwnd is cut in half, W/2.

Given this behavior, we want to have a simple model that predicts the throughput for a TCP connection.

To make our model more realistic, let's also assume that we have p = the probability loss. So, we assume that the network delivers 1 out of every p consecutive packets followed by a single packet loss.



Because the congestion window (cwnd) size increases a constant rate of 1 packet for every RTT, the height of the sawtooth is W/2 and the width of the base is W/2, which corresponds to W/2 round trips, or **RTT* W/2**.

The number of packets sent in one cycle the area under the sawtooth. Therefore, the total number of packets sent:

The number of packets sent in one cycle the area under the sawtooth. Therefore, the total number of packets sent:

$(W/2)^2 + 1/2*(W/2)^2 = 3/8*W^2$

As stated in our assumptions about out lossy network, it delivers *1/p* packets followed by a loss. So:

$1/p = (w/2)^2 + 1/2*(w/2)^2 = 3/8*W^2$, solving for $W = \sqrt{\frac{8}{3p}}$

The rate that data that is transmitted is computed as:

BW = data per cycle / time per cycle

Substituting from above:

$$\frac{data\ per\ cycle}{time\ per\ cycle} = \frac{MSS \cdot \frac{3}{8}W^2}{RTT \cdot \frac{W}{2}} = \frac{\frac{MSS}{p}}{RTT\sqrt{\frac{2}{3p}}}$$

We can collect all of our **constants** into $C = \sqrt{\frac{3}{2}}$, compute the throughput:

$$BW = \frac{MSS}{RTT} \cdot \frac{C}{\sqrt{p}}$$

In practice, because of additional parameters, such as small receiver windows, extra bandwidth availability, and TCP timeouts, our constant term C is usually less than 1. This means that bandwidth is bounded by:

$$BW < \frac{MSS}{RTT} \cdot \frac{1}{\sqrt{p}}$$

Optional Reading: Datacenter TCP

At this point, it is important to know that a lot of research has been going in optimizing the congestion control mechanisms. These optimizations are called for because of evolution of the networks. We looked at TCP CUBIC as one such example for high bandwidth delay product networks.

Similarly, data center (DC) networks are other kinds of networks where new TCP congestion control algorithms have been proposed and implemented. There are mainly two differences that have led to this -- the flow characteristics of DC networks are different from the public Internet. There are many short flows that are sensitive to delay. Thus, the congestion control mechanisms are optimized for both delay and throughput and not just the latter alone. Another reason is that DC networks are often owned by a private entity making any changes in the transport layer easier as the new algorithms need not co-exists with the older ones.

DCTCP and TIMELY are two popular examples of TCP designed for DC environments. DCTCP is based on a hybrid approach of using both implicit feedback aka packet loss and explicit feedback from the network using ECN for congestion control. TIMELY uses the gradient of RTT to adjust its window.