Short Circuit

Networks

Computer Networking: A Top-Down Approach
by James Kurose & Keith Ross
Eighth Global Edition
Up until Chapter 4

Summary by Emiel Bos

1 Introduction

1.1 Internet, Part I

The *internet* is a network of *hosts/end systems* (PCs, phones, smart devices, etc.) that are connected by *communication links* (coaxial cable, copper wire, optical fiber, radio spectrum, etc.) and *packet switches* (routers, link-layer switches, etc.). Every end system on the internet has an *IP address*, which, similarly to postal addresses, is hierarchical.

1.1.1 ISPs

End systems access the internet through *Internet Service Providers* (ISPs), be it residential, corporate, university, public, or cellular data ISPs. Each ISP is in itself a network of links and switches. It is independently managed, runs the IP protocol (specifies the format of the packets), and conforms to certain naming and address conventions. ISPs are also interconnected in hierarchical fashion, with lower-tier *access ISPs* interconnected through *regional ISPs*, which are interconnected through *tier-1 ISPs*, which are interconnected directly to each other. Even tier-1 ISPs do not have a presence in every city in the world. There are approximately a dozen tier-1 ISPs (e.g. Level 3 Communications, AT&T, Sprint, and NTT). There isn't an official tier-1 designation, however. Much of the evolution of this network of networks is driven by economics and national policy rather than performance considerations. There may be multiple competing regional ISPs in one region. What's more, in some regions, there may be a larger regional ISP to which the smaller regional ISPs in that region connect.

1.1.2 PoPs

A *point of presence* (PoP) is a group of one or more routers (at the same location) in a provider's network where customer ISPs can connect into the provider ISP. For a customer network to connect to a provider's PoP, it can lease a high-speed link from a third-party telecommunications provider to directly connect one of its routers to a router at the PoP.
Any (non-tier-1) ISP may choose to *multi-home*, i.e. to connect to two or more provider ISPs, to ensure connectivity to the internet even if one of its providers has a failure.

1.1.3 IXPs

Each ISP pays the ISP above it, with the amount proportional to the traffic it exchanges with the provider. To reduce these costs, a pair of nearby ISPs at the same level of the hierarchy can *peer*, i.e. directly connect their networks together so that all the traffic between them passes over the direct connection rather than through upstream intermediaries. This is typically settlement-free; neither party pays the other. To facilitate this, a third-party company can create an *Internet Exchange Point* (IXP), which is a meeting point where multiple ISPs can peer together. There are over 600 IXPs on the internet as of 2020.

1.1.4 RFCs

Internet standards are developed by the internet Engineering Task Force (IETF). The IETF standards documents are called *requests for comments* (RFCs). RFCs started out as general requests for comments (hence the name) to resolve network and protocol design problems that faced the precursor to the internet. RFCs tend to be quite technical and detailed. They define all protocols. There are currently nearly 9000 RFCs. Other bodies also specify standards for network components, most notably for network links. The IEEE 802 LAN Standards Committee specifies the Ethernet and wireless WiFi standards, for example.

1.2 Transmission

There are two fundamental approaches to moving data through a network of links and switches:

- In a *packet-switched network*, information is sent in *packets*, which are packages of segmented data with header bytes. The header contains the destination IP address. A packet switch forwards packets from on one of its incoming communication links to one of its outgoing communication links. The sequence of communication links and packet switches traversed by a packet is called a *route* or *path*. Most packet switches use *store-and-forward transmission*, which means the entire packet needs to be received before it can begin to transmit it. Packets are transmitted over each communication link at the full transmission rate of the link, measured in bits/second (R, for "rate"). If 3 packets, each of L bits (for "length"), are transmitted across N links, the transmission rate is $(N + 3 - 1)\frac{L}{R}$, whereas it would just be $3\frac{L}{R}$ without store-and-forward transmission.[1]
  A packet switch has an output buffer/queue for each output link, where packets are queued if that link is busy, causing additional queuing delays. These delays are variable and depend on the level of congestion in the network. Packet loss will occur if a buffer is full and cannot house incoming packets anymore.

  Packet forwarding is done in different ways in different types of computer networks, but in the internet, each router has a *forwarding table* that maps (portions of) destination addresses to that router's outbound links, which it uses to find the appropriate outbound link for an incoming packet.

- In a *circuit-switched network*, the resources needed along a path (buffers, link transmission rate) are reserved for the duration of the communication session between the end systems. When the network establishes the connection (the circuit, in the jargon of telephony), it reserves a fraction of each link's transmission capacity in the network's links, which guarantees a constant transmission rate. This distribution of a link's capacity can be done in two ways:
  - Frequency-division multiplexing (FDM) partitions the frequency spectrum of a link into frequency bands, with the width of each band called th *bandwidth*.
  - Time-division multiplexing (TDM) partitions time into frames of fixed duration, and each frame is divided into a fixed number of time slots, with one time slot dedicated to one connection.

Packet switching is not suitable for real-time services because of its variable and unpredictable end-to-end delays (due mostly to unpredictable queuing delays). Packet switching offers better sharing of transmission capacity than circuit switching and is simpler, more efficient, and less costly to implement than circuit switching, while circuit switching is wasteful because the dedicated circuits are idle during silent periods (inactivity), and establishing circuits and reserving capacity is complicated to coordinate. Time has favoured packet switching, with even many of today's circuit-switched telephone networks are migrating toward packet switching.

1.2.1 Link medium types

Data is transmitted across a network via (a combination of) any of the following physical media:

- Twisted pair copper wire; two insulated, 1mm thick copper wires, arranged in a regular spiral pattern to reduce the electrical interference from similar pairs close by. One pair is a single communication link. Typically, a number of pairs are bundled together in a protective wrapping. Unshielded twisted pair (UTP) is commonly used for computer networks within a building, i.e. LANs. Historically used as telephone lines. 10 Mbps to 10 Gbps, depending on wire thickness and distance.
- Coaxial cable; similar to twisted pair but with concentric copper wires rather than parallel. Historically used for cable television. Hundreds of Mbps.
- Fiber optics; thin, flexible medium that conducts pulses of light, with each pulse representing a bit. They are immune to electromagnetic interference, have very low signal attenuation up to 100 kilometers, and are very hard

[1] This ignores propagation delay.

to tap. However, the high cost of optical devices (e.g. transmitters, receivers, and switches) has hindered their adoption in LANs. Tens to hundreds of Gbps.

- Terrestrial radio channels; carry signals in the electromagnetic spectrum. Can be grouped:
  - Short distance; one or two meters. Wireless headsets, keyboards, and medical devices, etc.
  - Medium distance; tens too hundreds of meters. WiFi
  - Long distance; tens of kilometers. Cellular access technologies
- Satellite radio channels; links two or more Earth-based microwave transmitter/ receivers called ground stations. Regenerates received signals using a repeater (discussed and transmits the signal on another frequency. Two types:
  - Geostationary satellites; permanently remain above the same spot on Earth in orbit at 36,000 kilometers above earth, which incurs a substantial signal propagation delay of 280 milliseconds. Often used in areas without access to DSL or cable-based internet access. Hundreds of Mbps.
  - Low-earth orbiting (LEO) satellites; rotate around earth at a much closer distance. May be used for internet access sometime in the future.

A *local area network* (LAN) is a computer network within a limited area such as a house, university or office building. Ethernet (twisted-pair copper wire) and Wi-Fi (technically IEEE 802.11) are the two most common technologies to connect end systems to the (edge) router in LANs. A few ways in which this router in turn connects to the internet are:

- Digital Subscriber Line (DSL), in which case the user's telephone company is its ISP and connects it across its existing local telephone infrastructure (copper wiring) to its digital subscriber line access multiplexer (DSLAM), which separates the data from the phone signals.
- Cable, in which case the user's cable television company is its ISP and connects it across its existing cable television infrastructure to its cable modem termination system (CMTS), which turns the analog signal sent from the cable modems back into digital format (similar to a DSLAM). Fiber optics connect the cable head end to neighborhood-level junctions, from where traditional coaxial copper cable is used to reach individual houses and apartments, hence called hybrid fiber coax (HFC).
- Fiber-to-the-home (FTTH), which provide an optical fiber path from the central office (CO) directly to the home. Up and coming technology that is much faster.
- 5G fixed wireless, a provider's base station is connected wirelessly to a modem in the home across a 5G cellular network (radio spectrum).

1.2.2 Delay

Each *node* (i.e. host or router) along the path of a packet contributes *nodal delay* to the transmission time of that packet, which is the sum of:

- *Processing delay*: time required to examine the packet's header and determine where to direct the packet, and to potentially check (and correct) for bit-level errors. Typically in the order of microseconds.
- *Queuing delay*: time spent waiting in the queue/buffer to be transmitted onto the respective link, which depends on the number of previously queues packets. If there are no other packets, the delay is zero. Typically in the order of microseconds to milliseconds.
  The *traffic intensity* is $\frac{La}{R}$, where L is the length of the packet in bits, R is the transmission rate of the link, and a is the average rate at which packets arrive at the queue in packets/sec. If $\frac{La}{R} > 1$, the queue will get clogged, so this needs to be avoided. If $\frac{La}{R} \leq 1$, the nature of the arriving traffic impacts the queuing delay: if packets arrive periodically (one packet every $\frac{L}{R}$ seconds), then there won't be queuing delay, but if packets arrive in burst (e.g. N packets every $\frac{L}{R}N$ seconds, then the nth packet has a queuing delay of $(n-1)\frac{L}{R}$ seconds. Generally, average queuing delay increases exponentially with $\frac{La}{R}$.
- *Transmission delay*: time required to push/transmit all of the packet's bits into the link. It is $\frac{L}{R}$. Typically in the order of microseconds to milliseconds.
- *Propagation delay*: time required for the bits to propagate from the beginning of the link to the next node. It is d/s, where d is the distance in meters and s is the propagation speed, which depends on the medium and is in the range of $2 \cdot 10^8$ meters/sec to $3 \cdot 10^8 meters/sec$; equal to, or a little less than, the speed of light. Typically in the order of milliseconds.

The end-to-end delay is sum of the nodal delays of the sending host and all intermediate nodes on the path.
Two other types of delay are the delays that end systems incur on purpose as part of a medium-sharing protocol, and media packetization delay, present in VoIP applications that need to still fill a packet with encoded digitized speech.

The *round-trip time* (RTT) is the time it takes for a small packet to travel from client to server and then back to the client; useful for measuring request-response time. It includes packet-propagation delays, packet-queuing delays in intermediate routers and switches, and packet-processing delays.

1.2.3 Throughput

Instantaneous throughput is the rate in bits/sec at which a destination host receives bits. (This is often what is displayed in GUIs during downloads.) If a file consists of F bits and the transfer takes T seconds for the destination host to receive all F bits, then the *average throughput* of is $\frac{F}{T}$ bits/sec. Throughput is equal to the transmission rate of the bottleneck link.[2] (Typically, bottleneck links are in the access networks, since the core of the internet is over-provisioned with high speed links that experience little congestion.) For something like file transfer, throughput is more important than delay, while for something like VoIP delay is more important than throughput.

1.2.4 traceroute

`traceroute` (on Unix-like systems) and `tracert` (on Windows) are simple programs that take a destination hostname and send a special packet to each router on the path toward that destination. Each packet is has the specified destination as the ultimate destination and carries a sequence number. When router n receives packet marked n, it doesn't forward it further, but it sends back a short message that contains its name and IP address. The destination host does the same. The source hosts records for each packet the elapsed round-trip time and prints this together with the name and IP of the corresponding router, and in this way reconstructs the path to the destination with delays. The program often does multiple (e.g. three) runs. Lost packets have an asterisk rather than a time. Because queuing delay varies with time, the round-trip delay of packet n can sometimes be longer than that of packet $n + 1$ sent one router further.

1.3 Protocol layers

A *protocol* is a defined set of rules and regulations that determine how data is transmitted. A human analogy is etiquette; agreed-upon agreements on how humans ought to communicate. The many protocols are organized and partitioned in a layered structure, where each *layers* implement and provide a service to the more abstract layer above it while making use of the service provided by the layer below it. The *protocol stack* is as follows:

Layer	Packets are called	Service	Example protocols	Typical implementation	Postal analogy
Application	Messages	Regulate communication between applications.	HTTP, FTP, DNS	Software	Person
Transport	Segments	Transport segments between app endpoints.	TCP, UDP	Software	Apartment
Network	Datagrams	Route datagrams between hosts.	IP	Hardware, software	Building
Link	Frames	Transfer frames between nodes.	Ethernet, WiFi	Hardware	Postal service
Physical	Bits	Transmit bits across links.	Medium-dependent	Hardware	Transport

Each layer *encapsulates* whatever it receives from the layer above (called the *payload*) with layer-specific header information, e.g. a transport-layer datagram is an application-layer message with transport headers prepended, and a link layer frame is a link layer header plus payload from the network layer.
The sending hosts encapsulates a message all the way down the stack, and the receiving host decapsulates all the way up the stack, until the message is retrieved, i.e. application endpoints implement all layers of the stack. Routers implement only the first three layers, meaning that at each router, frames are decapsulated until they are datagrams, at which point the IP protocol determines the next node to route the datagram to and prepends new network layer headers. Link-layer switches only implement the first two layers, such that only the link layer headers are removed and reprepended.

The following chapters will discuss each of these layers in turn.

[2] The bottleneck link is actually the link with the lowest transmission rate divided by the number of total simultaneous transmissions, i.e. if there are multiple other connections making use of a link, than the transmission rate has to be divided.

1.4 Principles of reliable data transfer

In this section, we treat principles of reliable data transfer, which are applied at multiple layers in the protocol stack. A *reliable data transfer protocol* ensures no transferred bits are corrupted (i.e. flipped) or lost, and all are delivered in the order in which they were sent, even when the layer below the reliable data transfer protocol may be unreliable. We only consider *unidirectional data transfer* (from sender to receiver), but *bidirectional data transfer* (full-duplex) is conceptually no more difficult. See Computer Networking, A Top-Down Approach (8E), section 3.4, for the finite-state machine (FSM) diagrams. We iteratively develop and add complexity to our reliable data transfer protocol:

1. When the layer below the transport protocol is already reliable, reliable transmission is trivial and simply consists of taking data from the layer above and transmitting those as packets to the other side, where the data is unpacked from the packet and forwarded above. Both the sending and the receiving side have only one state, in which they wait for calls from above (sender) or below (receiver).

2. If the underlying layer is physical, bit errors may occur when a packet is transmitted, propagated, or buffered. *Automatic Repeat reQuest* (ARQ) protocols take care of this using:

   - Error detection; extra checksum bits are sent in the packet checksum field.
   - Receiver feedback; the receiver provides explicit feedback to the sender in the form of positive (ACK) and negative (NAK) acknowledgment replies, which only need to take 1 bit.
   - Retransmission; in case a packet is received in error.

   After sending a packet (with checksum), the sender goes in the state in which it waits for an ACK or NAK. If a NAK is received, the protocol retransmits the last packet, and only once an ACK is received does it return to its original state of waiting for data to send. Because the sender will not send new packets until it is sure that the current packet has been correctly received, such a protocol is known as a *stop-and-wait protocol*. The receiver still only has one state, which is waiting for calls from below; if the packet is corrupted is returns a NAK and discards the packet, and if it is fine, it forwards it to the layer above and returns an ACK.

   2.1 Unfortunately, ACKs or NAK replies can also get corrupted. The solution is to add checksum bits for the ACKs and NAKs as well, and to let the sender (also) retransmit packets whenever an acknowledgement is corrupt. Because the receiver doesn't know whether its acknowledgement was transmitted correctly and therefore doesn't know whether a received packet is a retransmission or new data, sequence numbers are added in packet headers (which for stop-and-wait protocols can be one bit indicating a retransmission or new data). The sender and receiver FSMs each now have twice as many states as before, half of them for packets with sequence number 0 and the other half for sequence number 1 (and apart from that they're identical).

   2.2 Instead of sending a NAK on receipt of a corrupt packet, the receiver can return an ACK for the last correctly received packet if we add sequence numbers to the acknowledgements as well. A sender that receives two ACKs for the same packet knows that the receiver did not correctly receive the packet following the packet that is being ACKed twice. This protocol is now NAK-free.

3. The underlying channel may lose packets as well. There are multiple solutions. In practice, timeouts are often used; the sender retransmits a packet when it hasn't received an acknowledgement in a span of time within which packet loss is likely but not guaranteed. (The sender shouldn't wait too long because of delay, but it also shouldn't retransmit too quickly in case of duplicate transmission, which – even though the previous protocol handles that – does encumber the network needlessly.)

4. Even though the protocol 3 is a functionally correct reliable data transfer protocol, the stop-and-wait nature makes it very slow; the sender is twiddling its thumbs waiting for acknowledgements significantly longer than it is busy sending bits into the channel. The solution is to *pipeline*; have the sender send multiple packets without waiting for acknowledgments. This entails increasing the range of sequence numbers (ranging from 0 to 2^k, where k is the number of bits in the sequence number header field), buffers on the sender side (for packets that have been transmitted but not yet acknowledged) and receiver side (for correctly received packets (only for selective repeat)), and one of two basic approaches toward pipelined error recovery:

   - *Go-Back-N* (GBN); a *sliding-window* protocol that only allows to have up to N (called the *window size*) unacknowledged packets in the pipeline. The sender keeps track of two variables: `base`, which is the sequence number of the oldest unacknowledged sequence number, and `nextseqnum`, wich is the smallest unused sequence number. Within the window, numbers in the interval [`base`, `nextseqnum`-1] corresponds to packets that have been sent but not acknowledged, and numbers in the interval [`nextseqnum`, `base`$+N-1$] can be used for packets that can be sent immediately. When called from above, a GBN sender sends the packet(s) and updates its variables if its window isn't full (`nextseqnum` < `base` $+N$), and else it – depending on the implementation – either returns the data back to the upper layer as an indication that the window is full, or buffers the data, or has a synchronization mechanism (semaphore, flag, etc.). An acknowledgment for a packet with sequence number n will be taken to be a cumulative acknowledgment for all packets with sequence numbers up to and including n. On timeout, the sender resends all packets that have been previously sent but not acknowledged, hence the name. If a single timer is used, the timer is restarted when an ACK is received but there are still transmitted but unacknowledged packets out there. The receiver only maintains the sequence number of the next expected in-order packet in variable `expectedseqnum`, and only forwards and acknowledges correct and in-order-received packets; any other kind of packet is discarded[3], after which an ACK for the most recently received in-order packet is resend.

- *Selective repeat*; a protocol similar to GBN, but that avoids the potentially large number of retransmissions (if there are many packets in the GBN pipeline) by having the sender retransmit only those packets that it suspects were received in error, which requires that the receiver individually acknowledges correctly received packets. So receivers will now also acknowledge correct packets when they're out of order, and buffer those. The receiver now maintains its own `base` variable for the next in-order packet it expects:

  – Correct packets with sequence number smaller than `base` are ACKed again.

  – Correct packets with sequence number larger than or equal to `base` are ACKed and forwarded together with any consecutively numbered buffered packets (and `base` is moved).

  – Correct packets with sequence number larger than the sequence number of any buffered packet and within the window are buffered and ACKed.

  – Correct packets with sequence number larger than the window are discarded.

  The sender now maintains a separate logical timer for each packet.[4] Upon receipt of an ACK, the sender marks that packet as acknowledged, and if the packet's sequence number is equal to `base`, the window is moved forward to the smallest unacknowledged packet number, and newly available sequence numbers are used up by transmitting packets.
  It is important that the window size must be less than or equal to half the size of the sequence number space, or else a scenario may occur in which the receiver cannot distinguish a new packet from a retransmission. More precisely, the receiver doesn't know whether the sender has received a bunch of ACKs and has looped around the sequence number space, or that it hasn't received any ACKs and timed out on an old packet and is retransmitting that.

5. Lastly, packet reordering can occur, which can be thought of as the network essentially buffering packets and spontaneously emitting these packets at any point in the future. The approach taken in practice is to ensure that a sequence number is not reused until the sender is sure that any previously sent packets with the same number are no longer in the network. This is done by giving each packet a maximum packet lifetime (approximately three minutes is assumed in the TCP extensions for high-speed networks), after which it is deleted by the network.

1.5 Principles of congestion control

Packet loss typically results from the overflowing of router buffers as the network becomes congested. The principles of reliable data transfer just discussed treat a symptom of network congestion. We'll now discuss the principles of *congestion control*; mechanism to treat the cause of network congestion by throttling senders. The costs of a congested network (in which packet-arrival rate nears bottleneck link capacity) are manifold: large queuing delays, retransmissions in order to compensate for dropped packets due to buffer overflow, a router wasting its link bandwidth to forward unneeded copies of a prematurely retransmitted packet, and wasted transmission capacity up to the point along the path at which a packet is dropped. Contrary to the principles of reliable data transfer, which are applicable in multiple layers, congestion control is [exclusively?] done in the transport layer, and more specifically, only by TCP. In practice, we can distinguish between two broad approaches to congestion control that are taken in practice:

- *end-to-end congestion control*, in which the network layer provides no explicit support to the transport layer.
- *network-assisted congestion control*, which may be as simple as a single bit indicating congestion at a link, or more complex, such as *ATM Available Bite Rate* (ABR) congestion control, in which a router informs the sender of the maximum host sending rate it can support on an outgoing link. There are two feedback pathways a router may send

[3] Discarding correct but out-of-order packets isn't wasteful, because the sender will resend it and the previous missing packet(s) because those were not acknowledged. Also, now the receiver need not buffer any out-of-order packets.

[4] A single hardware timer can be used to mimic the operation of multiple logical timers.

congestion information back to a sender:

- Direct feedback may be sent from a network router to the sender, typically in the form of a choke packet indicating congestion.
- A router marks a field in a packet flowing from sender to receiver to indicate congestion, and then the receiver notifies the sender of the congestion. This is the more common approach, and takes a full RTT.

2 Link layer

3 Network layer

Each host can be identified by a network-layer IP address.

3.1 IP

Internet Protocol (IP) is a *best-effort delivery service*; it does it best to deliver segments between communicating hosts, but it is unreliable and hence makes no guarantees, i.e. no guaranteed deliveries, no guaranteed order, and no guaranteed data integrity.

4 Transport layer

The primary task of the transport layer is to extend the network layer's delivery service between two end systems to a delivery service between two processes running on those end systems. It provides for *logical communication*, which should make it look from an application's perspective as if the two processes were communicating directly without an error-prone, multi-link, globe-spanning network path in between. Transport-layer protocols are implemented only in the end systems, not in network routers in between. They take application-layer messages, (possibly) break them into smaller chunks, and encapsulate those into transport-layer *segments* by adding a transport-layer header to each chunk. The transport layer passes the segment to the network layer, where the segment is encapsulated within a network-layer datagram and sent to the destination. The reverse happens on the receiving side.

In other words, the transport layer takes data from the network layer (which only delivers between hosts) and forwards it to the correct process running on the host. Actually, it forwards segments to sockets. A *socket* is a software API through which application-level protocols send and receive messages. Each socket is identified by a 16-bit *port number*, ranging from 0 to 65535. Ports 0 through 1083 are reserved for open application-level protocols. Each transport-layer segment has a set of fields – a source port number field and a destination port number field – that allows the transport protocol to direct it to the appropriate socket, which is called *demultiplexing*. The job of gathering data chunks from (different) sockets, encapsulating each with header information (for demultiplexing) to create segments, and passing those to the network layer is called *multiplexing*.[5] At the very least, a transport layer has to provide a multiplexing/demultiplexing service.

There are four dimensions along which to classify the services offered by transport protocols:

- Reliable data transfer; data sent by one end of the application/process is delivered correctly and completely to the other end of the application/process. *Loss-tolerant applications* do not require this. See Section 1.4.
- Throughput guarantees; guaranteed available throughput at some specified rate in bits/sec. If the transport protocol cannot provide the requested throughput (but could provide a lower guaranteed throughput) the application should either encode at a lower rate or give up. *Bandwidth-sensitive applications* require this, while *elastic applications* do not.
- Timing guarantees; guaranteed maximum travel time for bits. *Real-time applications* require this.
- Security; such as confidentiality (achieved with cryptography), data integrity and end-point authentication.

Port scanning is the process of sequentially looping through port numbers and checking which ports on some remote host either respond to transmitted UDP segments or accept TCP connection requests. A popular open source port scanning application is `nmap`, included in most Linux distributions. It returns a list of open, closed, or unreachable ports. This is useful for system administrators, but can also be used by attackers to exploit known security flaws of applications using conventionally known port numbers. When a user specifies a host and port, `nmap` sends a TCP SYN segment. If it receives

[5] The terms multiplexing and demultiplexing can be used in any situation where a singular protocol passes data to multiple different protocols above it.

a SYNACK segment, `nmap` deems the port open; if it receives a RST segment, it means the target host is reachable but doesn't run an application on that port; if it receives nothing, it likely means that the SYN segment was blocked by an intervening firewall and never reached the target host.

The two transport layer protocols are TCP and UDP.

4.1 UDP

User Datagram Protocol (UDP)[6] is connectionless and provides no reliable data transfer[7] nor congestion control. It is a lightweight protocol which applications can use to just pump data into and see what comes out at the other side.

The following is code for a simple UDP client program in Python 3:

```
from socket import * # This module forms the basis of all network communications in Python
serverName = "cis.poly.edu" # Or an IP addres like "128.138.32.126"
serverPort = 12345
clientSocket = socket(AF_INET, SOCK_DGRAM) # Creates a socket; AF_INET indicates IPv4 and
    SOCK_DGRAM indicates a UDP port. The OS automatically binds a port to the socket
clientMessage = "Hi!"
clientSocket.sendto(clientMessage.encode(),(serverName, serverPort)) # Sends data with destination
    attached to the server. Sockets only accept bytes, so the messages needs to be encoded. Source
    address is attached automatically by the OS.
serverMessage, _= clientSocket.recvfrom(2048) # Wait for data, returned as first return value. The
    server's address (a tuple of (IP address, port)) is the second return value (but we already
    have those). Buffer size is 2048
print(serverMessage.decode()) # Decode the server's message and print
clientSocket.close()
```

The following is code for a simple UDP server program in Python 3:

```
from socket import *
serverPort = 12345
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(("", serverPort)) # Assigns port 12345 to the server's socket

# This while-loop must be running in order to receive client messages
while True:
   clientMessage, clientAddress = serverSocket.recvfrom(2048) # Wait for data to arrive
   serverMessage = "Hello!"
   serverSocket.sendto(serverMessage.encode(), clientAddress) # clientAddress is, again, a (IP
       address, port)-tuple
```

A UDP socket (or rather, its address) is identified by a two-tuple (IP address, port number), i.e. two distinct segments with different source addresses but the same destination address will be forwarded to the same socket (this is different for TCP). As shown, a destination address needs to be attached to every packet before outputting it through a UDP socket (which is not the case for TCP). UDP simply uses the source port number and destination port number fields in a segment's header for (de)multiplexing.

UDP barely adds anything to IP: the only two services UDP provides are process-to-process data delivery and data integrity checking. UDP headers have only four fields, each consisting of two bytes: source port number, destination port number, length (number of bytes in the UDP segment, i.e. header plus application data), and checksum (to check for errors[8][9]). The latter is calculated by taking the 1s complement (i.e. flipping all `0`s to `1`s and vice versa) of the sum (with overflow wrapped around to the least significant bit) of all the 16-bit words in the segment. At the receiver, all four 16-bit words are added, including the checksum. This sum will be all-`1`s if and only if no errors are introduced. While UDP provides error checking, it does not do anything to recover from them. Some implementations simply discard damaged segments, others pass the damaged segment to the application with a warning.

The benefits of UDP are finer application-level control over what data is sent, and when, since UDP sents stuff immediately, with no regards for congestion control. This is useful for real-time applications, which require a minimum sending

[6] Internet literature (such as RFCs) often refer to transport-layer packets as datagrams in the context of UDP, but as segments in the context of TCP. In this work, I refer to both TCP and UDP packets as segments, and reserve the term datagram for network-layer packets.

[7] Of course, developers can built reliability into the application itself, but this is nontrivial.

[8] In truth, the checksum is also calculated over a few of the fields in the IP header in addition to the UDP segment.

[9] While many link-layer protocols (including the popular Ethernet protocol) also provide error checking, there is no guarantee that all the inbetween links provide error checking, and bit errors could be introduced when a segment is stored in a router's memory, hence the need for end-to-end error checking.

rate, do not want to overly delay segment transmission, and can tolerate some data loss. Furthermore, it avoids delay due to connection-establishment, does not need to maintain connection state – as TCP does – in a bunch of buffers and parameters (meaning servers can support many more active clients), and has a small packet header overhead. Downsides are the lack of congestion control that may result in packet loss, and the crowding out of TCP sessions.

When a host receives a UDP packet whose destination port number doesn't match with an ongoing UDP socket, the host sends a special ICMP datagram.

4.2 TCP

4.2.1 Handshaking

Transmission Control Protocol (TCP) is connection-oriented. This means the client and server exchange transport-layer control information with each other before the application-level messages begin to flow, called *handshaking*. Specifically, a TCP connection is set up with a *three-way handshake*: the client sends a small TCP segment to the server, the server acknowledges and responds with a small TCP segment, and the client acknowledges the acknowledgement. This last acknowledgement is usually paired with the first request. (This process is detailed below.) The resulting connection is a *full-duplex* connection in that the two processes[10] can send messages at the same time. Messages do not need to attach destination addresses (which is the case for UDP), because of the connection. When the application finishes sending messages, it must tear down the connection. Connections are also strictly *point-to-point*, so multicasting is not possible with TCP. During handshaking, both hosts initialize connection state variables.
A server expecting TCP connection requests from clients must maintain a special "welcoming" socket used to receive connection requests through. After a client reaches out to the server via this socket and during the three-way handshake, the server creates a new "connection" socket dedicated to that particular client through which the server communicates with the client.

A connection is set up with a three-way handshake follows, with the various *TCP states* that both hosts go through inbetween:

- Client state: `CLOSED` | Server state: `CLOSED`

0 The server creates a listen socket.

- Client state: `CLOSED` | Server state: `LISTEN`

1. The client (the host that wants to initiate the connection) sends a *SYN segment* with the SYN bit set, with its randomly chosen initial sequence number in the sequence number field and without data.

- Client state: `SYN_SENT` | Server state: `LISTEN`

2. Upon receipt, the server allocates buffers and variables to the connection, and sends a *SYNACK segment*, with the SYN bit set, with its own randomly chosen initial sequence number in the sequence number field, the client's sequence number that was received +1'ed in the acknowledgement field, and without data.

- Client state: `SYN_SENT` | Server state: `SYN_RCVD`

3. Upon receipt, the client allocates buffers and variables to the connection, and acknowledges the server's acknowledgement (without the SYN bit) by putting the server's initial sequence numbr +1'ed in the acknowledgement field and optionally with the first client-to-server data in the segment payload.

- Client state: `ESTABLISHED` | Server state: `SYN_RCVD`

4. The server receives the acknowledgement.

- Client state: `ESTABLISHED` | Server state: `ESTABLISHED`

The fact that a server allocates and initializes connection variables and buffers when receiving a SYN segment sets the stage for a classic DoS attack known as the SYN flood attack, in which an attacker sends a deluge of SYN segments without completing the third stage of the handshake. Most major operating systems therefore deploy *SYN cookies*. Instead of creating a half-open TCP connection, the server returns a SYNACK packet with a cookie: an initial TCP sequence number that is a complicated hash function of source and destination IP addresses and port numbers of the SYN segment as well as a secret number only known to the server. The server only remembers the secret number, so that, if the client does not return an ACK, little harm has been done. A legitimate client will return an ACK, and the server can verify that it corresponds to some SYN sent earlier by rehashing the IP addresses and port numbers in the SYNACK and the secret

[10] There is not always a one-to-one correspondence between connection sockets and processes. Web servers often use only one process, and create a new thread with a new connection socket for each new client connection. We'll use the term "process", however.

number. If this hash plus one is the same as the acknowledgment (cookie) value in the client's SYNACK, it is valid. Both client and server can close the connection, but supposing the client initiates this:

- Client state: ESTABLISHED | Server state: ESTABLISHED

1. The client sends a segment with the FIN bit set.

- Client state: FIN_WAIT_1 | Server state: ESTABLISHED

2. Upon receipt, the server acknowledges the shutdown segment.

- Client state: FIN_WAIT_1 | Server state: CLOSE_WAIT

3. The client receives the acknowledgement, sends nothing.

- Client state: FIN_WAIT_2 | Server state: CLOSE_WAIT

4. The server then sends its own separate shutdown segment with the FIN bit set.

- Client state: FIN_WAIT_2 | Server state: LAST_ACK

5. Upon receipt, the client acknowledges the server's shutdown segment, and then waits an implementation-defined amount of time (typically 30 seconds to 2 minutes) before going CLOSED (this allows the client to resend the final acknowledgment in case it gets lost).

- Client state: TIME_WAIT | Server state: LAST_ACK

6. The server receives the ACK.

- Client state: CLOSED | Server state: CLOSED

When either host enters the CLOSED state, all buffers and variables are deallocated.

TCP provides reliable data transfer – meaning bytes are guaranteed to arrive, and in order – and includes a congestion-control mechanism, a service for the general welfare of the internet rather than for the direct benefit of the communicating processes, which throttles a sending process (client or server) when the network is congested between sender and receiver.

TCP connections can be *persistent*, in which case one connection is used for multiple request-response messages, or *non-persistent*, in which case a separate TCP connection is made for every request-response pair. For each connection, TCP sockets must be created, TCP buffers must be allocated and TCP variables must be kept in both the client and server, which, if many requests are made non-persistently, can place a burden on the servers and requires two RTTs for each request.

The following is code for a simple TCP client program in Python 3:

```
from socket import *
serverName = "cis.poly.edu" # Or an IP addres like "128.138.32.126"
serverPort = 12345
clientSocket = socket(AF_INET, SOCK_STREAM) # SOCK_STREAM indicates TCP
clientSocket.connect((serverName,serverPort)) # Initiates connection with a three-way handshake
    behind the scenes. This sends a connection establishment request segment (which is simply a TCP
    segment with a special connection-establishment bit set)
clientMessage = "Hi!"
clientSocket.send(clientMessage.encode()) # Note how, contrary to UDP, a destination address is not
    needed
serverMessage = clientSocket.recv(1024) # Continues to accumulate until the line ends with a
    carriage return
print(serverMessage.decode())
clientSocket.close() # Causes TCP in the client to send a TCP message to TCP in the server
```

The following is code for a simple TCP server program in Python 3:

```
from socket import *
serverPort = 12345
serverSocketWelc = socket(AF_INET, SOCK_STREAM) # Create welcoming socket
serverSocketWelc.bind(("", serverPort))
serverSocketWelc.listen(1) # Listen for TCP connection requests from clients, with the maximum
    number of queued connections being 1

# This while-loop must be running in order to receive client messages
while True:
   serverSocketConn, addr = serverSocketWelc.accept() # Creates a new, client-specific socket when
```

```
        a client requests a connection
    clientMessage = serverSocketConn.recv(1024).decode()
    serverMessage = "Hello!"
    serverSocketConn.send(serverMessage.encode())
    serverSocketConn.close()
```

A TCP socket (or rather, its address) is identified by a four-tuple (source IP address, source port number, destination IP address, destination port number), i.e. two distinct segments with different source addresses but the same destination address will be forwarded to different sockets (with the exception of the original connection-establishment request). TCP uses all four fields for (de)multiplexing. This allows Web servers to use port 80 both for the welcoming socket as well as client-specific sockets, so that both the initial connection-establishment segments and the segments carrying HTTP requests are sent to port 80.

Once a sending process passes data through its socket, TCP directs this data to the connection's *send buffer*, reserved during handshaking. From time to time, TCP will grab chunks of data from the send buffer and pass them to the network layer.[11] The maximum amount of (application-layer) data that can be grabbed and placed in a segment is the *maximum segment size* (MSS)[12], typically set by first determining the *maximum transmission unit* (MTU) – the length of the largest link-layer frame that can be sent by the local sending host – and setting the MSS to ensure that a TCP segment (encapsulated in an IP datagram) plus the TCP/IP header length (typically 40 bytes) are less than the MTU. Both Ethernet and PPP link-layer protocols have an MTU of 1,500 bytes, and therefore a typical value of MSS is 1460 bytes.

A segment typically has the following structure (note that the first 8 bytes are the same as a UDP header):

- Header (typically 20 bytes, but can be of variable length due to the options field):
  - Source port number field (2 bytes)
  - Destination port number field (2 bytes)
  - Sequence number field (4 bytes)
  - Acknowledgement number field (4 bytes)
  - Header length field; the number of 32-bit words of the header (4 bits)
  - (4 unused bits)
  - Flag field (8 bits):
    - CWR, used in explicit congestion notification
    - ECE, used in explicit congestion notification
    - URG, indicates that there is data in this segment that the sending-side upper-layer entity has marked as "urgent". The location of the last byte of this urgent data is indicated by the 16-bit urgent data pointer field. TCP must inform the receiving-side upper-layer entity when urgent data exists and pass it a pointer to the end of the urgent data. Not actually used in practice.
    - ACK, used to indicate that the segment contains an acknowledgment for a segment that has been successfully received
    - PSH, indicates that the receiver should pass the data to the upper layer immediately. Not actually used in practice.
    - RST, used for connection setup and teardown
    - SYN, used for connection setup and teardown
    - FIN, used for connection setup and teardown
  - Receive window field (2 bytes)
  - Checksum field (2 bytes)
  - Urgent data pointer (2 bytes)
  - Options field, used when a sender and receiver negotiate the MSS or as a window scaling factor for use in

[11] The TCP specification is very laid back about specifying when TCP should actually send buffered data, merely stating that TCP should "send that data in segments at its own convenience.”

[12] Which is a misnomer, as it limits the size of the segment's data field rather than the size of the segment itself.

high-speed networks (variable length and optional)

- Data (MSS bytes)

4.2.2 Reliable data transfer

TCP builts heavily on the principles of reliable data transfer, but does some things differently. TCP views data as an unstructured, ordered stream of bytes, and as such a segment's sequence number (32-bits) is the byte-stream number of the first byte in the segment. Also, the acknowledgment number in a segment is the sequence number of the next expected byte. TCP uses *cumulative acknowledgements*; only bytes up to the first missing byte in the stream are acknowledged. The TCP RFCs do not specify what to do with out-of-order segments, but in practice, receivers buffer any out-of-order bytes and wait for the missing bytes to fill in the gaps. Because TCP connections are full-duplex, acknowledgements can be *piggybacked* with data segments, so that such segments serve a dual purpose. In particular, segments always have both a sequence number indicating the next byte to-be-sent (even if the segment is a pure acknowledgement and doesn't actually yet contain the indicated byte) and an acknowledgement number indicating the next expected byte.
Rather than sequence numbers starting at 0, both sides of a TCP connection randomly choose their initial sequence number, in order to minimize the possibility that a segment that is still present in the network from an earlier, already-terminated connection using the same port numbers between the same two hosts is mistaken for a valid segment in a later connection. These initial sequence numbers are exchanged during the three-way handshake.

In order to determine the timeout interval, at any point in time, TCP samples the RTT of one of the transmitted but currently unacknowledged segments, leading to a new sample approximately once every RTT. A sample is used to update an exponential weighted moving average (EWMA) of the estimated RTT:

$$\text{RTT}_{\text{est}} = (1-\alpha) * \text{RTT}_{\text{est}} + \alpha * \text{RTT}_{\text{smp}} \tag{1}$$

and an EWMA of the difference between sample and estimation; the estimation's variability:

$$\text{RTT}_{\text{dev}} = (1-\beta) * \text{RTT}_{\text{dev}} + \beta * |\text{RTT}_{\text{smp}} - \text{RTT}_{\text{est}}| \tag{2}$$

where $\alpha = 0.125$ and $\beta = 0.25$ are recommended. TCP's method for determining the retransmission timeout interval is then:

$$\text{RTT}_{\text{est}} + 4 * \text{RTT}_{\text{dev}} \tag{3}$$

while an initial value of 1 second is recommended. The recommended TCP timer management procedures use only a single retransmission timer, even when pipelining. Disregarding TCP flow or congestion control and assuming unidirectional transfer, a TCP sender restarts the timer in the following cases (it is helpful to think of the timer as being associated with the oldest unacknowledged segment):

- In case of data received from the application above, (data is transmitted in segment(s) over IP and) the timer is started if it isn't already running.
- In case of timeout, (the not-yet-acknowledged segment with smallest sequence number is retransmitted and) the timer is restarted with double the previous interval. This provides a limited form of congestion control, because the timeout is most likely caused by congestion in the network.
- In case an ACK is received and `ackNum > sendBase`[13], (the sender sets `sendBase = ackNum` and) the timer is restarted if there are currently any not-yet-acknowledged segments.
  If the sender receives three *duplicate ACKs* (for which `ackNum <= sendBase`), the sender does a *fast retransmit*; it retransmits the duplicately reacknowledged segment before the timer runs out, avoiding long end-to-end delay. To explain: when a TCP receiver detects a gap in the data stream (i.e. a missing segment), it simply reacknowledges the last in-order byte of data it has received (because TCP doesn't do NAKs). Because a sender often sends/pipelines multiple segments back-to-back, if one segment is lost, there will likely be many back-to-back duplicate ACKs.

TCP's error recovery can be classified as a hybrid between Go-Back-N (GBN) and Selective Repeat (SR): acknowledgments are cumulative and not individual and the TCP sender only maintains the smallest transmitted-but-unacknowledged byte's sequence number, but TCP receivers buffer correctly received but out-of-order segments, and a TCP sender only

[13] Note that the TCP state variable `sendBase` is the sequence number of the oldest unacknowledged byte (and `sendBase - 1` is therefore the sequence number of the last byte known to have been received correctly and in order at the receiver).

retransmits one segment if that is the only segment for which no ACK is received before timeout (and it doesn't even do that if an ACK for a subsequent segment is received before timeout). Furthermore, a proposed modification to TCP, the so-called selective acknowledgment, allows a TCP receiver to acknowledge out-of-order segments individually in favor of cumulative acknowledgmenet.

Both participants in a TCP connection reserve a receive buffer, from which an application can read data at its own leisure. If the sender sends data quicker than the receiving application reads it, the receive buffer can overflow. TCP provides *flow control* to match the rate of sending against the rate of reading. We assume for simplicity unidirectional transfer from a sender to a receiver, and that the receiver discards out-of-order segments. The receiver maintains variables `bufferSize`, the size of the receive buffer in bytes; `lastByteRead`; `lastByteRcvd`; and `rcwd`, the *receive window*: the remaining space in bytes left in the buffer, calculated as `rcwd = bufferSize - (lastByteRcvd - lastByteRead)`. The receiver tells the sender how much spare room it has left by putting `rcwd` in the receive window field of every segment it sends. The sender keeps track of `lastByteSent` and `lastByteAckd`, and only sends new segments if `LastByteSent - LastByteAcked <= rwnd` still holds afterwards. In order to avoid the situation in which the sender thinks `rwnd = 0` but the receiver cannot let the sender know that its buffer is emptying again because it has no segments to send, the TCP specification requires the sender to still continue to send segments with one data byte when it thinks `rwnd = 0`, which can then be acknowledged by the receiver.

When a host receives a TCP segment whose port numbers or source IP address do not match with any of its ongoing sockets, it returns a special reset segment to the source with the RST flag bit set, essentially telling the source there is no socket for that segment.

4.2.3 Congestion control

In the following discussion, we assume that the TCP receive buffer is so large that the receive-window constraint can be ignored, in order to focus on congestion control and not flow control.

- "Classic" TCP uses end-to-end congestion control, adjusting the rate at which it sends traffic into its connection as a function of perceived network congestion, such that TCP senders use all the available bandwidth without congest the network. A TCP sender receives no explicit congestion indications from the network layer, and instead infers congestion only through observed packet loss. The rate is limited by imposing another constraint on the flow control condition for when TCP sends segments, given above: TCP actually only sends new segments if `LastByteSent - LastByteAcked <= min{cwnd, rwnd}` still holds afterwards, where `cwnd` is an additional variable maintaining the *congestion window*. TCP takes the arrival rate of acknowledgments as indication of congestion on the path, and adjusts its congestion window accordingly. A lost segment (detected by a timeout or the receipt of four ACKs) for a given segment implies congestion, and therefore the TCP sender's rate is decreased when a segment is lost. An acknowledged segment indicates a congestion-free path, and therefore the sender's rate can be increased. The TCP sender thus increases its transmission rate to probe for the rate at which congestion begins, backs off from that rate, and then to begins probing again to see if the congestion onset rate has changed. Because TCP uses acknowledgments to trigger (or clock) its increase in congestion window size, TCP is said to be *self-clocking*.

  TCP's congestion-control algorithm defines three TCP connection states:

  - *Slow start mode* is the initial state of a TCP connection in which the value of `cwnd` is typically initialized to the small value of 1 MSS (resulting in an initial sending rate of roughly $\frac{\text{MSS}}{\text{RTT}}$) and is increased by 1 MSS for every received ACK, resulting in exponential rate increase – a doubling of the sending rate every RTT – contrary to what the name suggests.

    - ∗ Whenever a timeout occurs, the sender sets the value of a second state variable, `ssthresh` (for "slow start threshold") to $\frac{\texttt{cwnd}}{2}$, sets the value of `cwnd` to 1, and begins the slow start process anew (and retransmit the segment of course).

    - ∗ Whenever `cwnd` equals `ssthresh`, slow start ends and TCP transitions into congestion avoidance mode.

    - ∗ Whenever three duplicate ACKs are received, TCP performs a fast retransmit and transitions into the fast recovery state. Because the receipt of the triple ACKs suggests that the network is not terribly congested, TCP reacts less drastic than with a timeout-indicated loss and only halves the value of `cwnd` (adding in 3 MSS for good measure to account for the triple duplicate ACKs received). It still sets `ssthresh` to $\frac{\texttt{cwnd}}{2}$.

  - *Congestion avoidance mode* is the state in which TCP is more careful in approaching the congestion onset rate and increases the value of `cwnd` by just a single MSS every RTT. There are several ways to do this, but TCP typically increases `cwnd` by $\frac{\text{MSS}}{\texttt{cwnd}}$MSS bytes per received acknowledgement, since $\frac{\text{MSS}}{\texttt{cwnd}}$ ACKs are expected every RTT.

    - Whenever a timeout occurs, TCP does the exact same as in slow start mode and enters that mode again.
    - Whenever three duplicate ACKs are received, TCP does the exact same as in slow start mode and enters fast recovery mode.
  - *Fast recovery mode* is a recommended, but not required, state of the TCP specification. `cwnd` is increased by 1 MSS for every duplicate ACK received for the missing segment that caused TCP to enter the fast-recovery state.
    - Whenever a timeout occurs, TCP does the exact same as in slow start mode and enters that mode again.
    - When eventually an ACK arrives for the missing segment, TCP enters the congestion avoidance state after deflating `cwnd` by setting it to `ssthresh`.

  Figure 3.51 in Chapter 3.7 summarizes the above in a handy FSM diagram. An early version of TCP from the late 1980's, known as TCP Tahoe, unconditionally cut its congestion window to 1 MSS and entered the slow-start phase after either a (timeout-indicated or triple-duplicate-ACK-indicated) loss event. A newer version of TCP, TCP Reno, which nearly all web servers used in the 2000's, incorporated fast recovery. Ignoring the slow start state and timeout events, because `cwnd` increases linearly/additively (by 1 MSS per RTT) in congestion control mode and decreases multiplicative (by a factor of 0.5) on a triple duplicate-ACK event, the value of `cwnd` over time looks like a saw wave, and TCP Reno's congestion control is often referred to as *additive-increase, multiplicative-decrease* (AIMD). The average throughput of a TCP Reno connection is $\frac{0.75\texttt{cwnd}_{\max}}{RTT}$, where $\texttt{cwnd}_{\max}$ is the value of `cwnd` when a loss event occurs, ignoring slow start and assuming RTT and $\texttt{cwnd}_{\max}$ remain approximately constant. (More sophisticated and accurate models have been proposed.) However, if the state of a congested link doesn't change much, cutting the sending rate in half (or even worse, setting it to 1 as in TCP Tahoe) and then increasing linearly is overly cautious and wastes throughput.

  The property of *fairness* means that, if n TCP connections traverse the same bottleneck link with capacity R, the average transmission rate of each connection is approximately $\frac{R}{n}$, i.e. each connection gets an equal share of the link's bandwidth. TCP Reno's AIMD approach is fair. If two TCP connections share a single link, assume that the two connections have the same MSS and RTT and are exclusively operating in congestion avoidance/AIMD mode, whatever state of sending rates (values of `cwnd` of both connections) the two connections start with, they will over time slowly converge to a point where they share equal bandwidth (the two values of `cwnd` are equal). Figure 3.57 in Section 3.7 illustrates this. In practice, these assumptions are typically not met, and client-server applications can obtain very unequal portions of link bandwidth. In particular, it has been shown that sessions with a smaller RTT are able to grab newly available bandwidth at a bottleneck link more quickly. Of course, since UDP doesn't care about congestion control, it is possible for UDP sources to crowd out TCP traffic. Even if UDP traffic behaved fairly, the fairness problem would still not be completely solved: there is nothing to stop a TCP-based application from using multiple parallel connections, which is often what Web browsers do to transfer multiple objects.[14]

  TCP Cubic, introduced in 2008, offers a better way of probing for a packet sending rate that is just below the threshold of triggering packet loss. It only changes the congestion avoidance mode and the congestion window is still increased only on ACK receipt. The increase is determined by a function that takes as input the time remaining until `cwnd` matches the previous congestion threshold; the value of `cwnd` at which the previous packet loss was triggered (assuming no losses until that time). The function quickly ramps up TCP's sending rate to get close to this congestion threshold rate, and only then decreases the increase as it approaches it. In other words, rather than increasing linearly, it increases logarithmically until the time of (expected) congestion. If the congestion threshold has disappeared and TCP is past that point in time, it increases exponentially, i.e. it is still cautious right after, but increases faster after that. See Figure 3.54 in Chapter 3.7. In 2014, nearly 50% of the 5000 most popular Web servers shows that were running TCP Cubic.

- "Modern" TCP are
  - extensions that allow for a form of network-assisted congestion control called *Explicit Congestion Notification* (ECN).[15] At the network layer, two bits (four possible values) in the Type of Service field of the IP datagram header are used for ECN. One setting of this field is used by routers to indicate congestion, which is then carried in the marked datagram to the receiver. The receiver in turn informs the sender by setting the ENE (Explicit Congestion Notification Echo) bit in the header of the next ACK segment (so not in the Type of Service field of IP datagrams). The TCP sender reacts to such an ACK by halving the congestion window (as

[14] The exact number of multiple connections is configurable in most browsers.

[15] Other transport-layer protocols besides TCP may also make use of network-layer-signaled ECN. The Datagram Congestion Control Protocol (DCCP) provides a low-overhead, congestion-controlled UDP-like unreliable service that utilizes ECN. DCTCP (Data Center TCP) and DCQCN (Data Center Quantized Congestion Notification), designed specifically for data center networks, also makes use of ECN. ECN capabilities are increasingly deployed in popular servers as well as in routers along paths to those servers.

it would react to a lost segment using fast retransmit) and sets the CWR (Congestion Window Reduced) bit in the next transmitted sender-to-receiver segment. There is no standardized definition of congestion (which is left as a configuration choice to the router vendor), but the intuition is that the congestion indication bit is set to signal the onset of congestion but before loss actually occurs. A second setting of the ECN bits is used by the sending host to inform routers that the sender and receiver are ECN-capable and thus capable of taking action.

- variations of TCP that take a delay-based congestion-avoidance approach to proactively detect congestion onset before packet loss occurs, similarly to routers signalling congestion before it actually occurs. In TCP Vegas, the sender measures the RTT of all acknowledged packets and maintains RTT_{min} as the minimum of these. TCP Vegas assumes this RTT_{min} occurs when the path is uncongested, at which point the throughput rate would be $\frac{\texttt{cwnd}}{\text{RTT}_{\text{min}}}$. The intuition behind is that if the actual sender-measured throughput is close to this value, the sending rate can be increased since (by definition and by measurement) the path is not yet congested. However, if the actual sender-measured throughput is significantly less than the uncongested throughput rate, the path is congested and the Vegas TCP sender will decrease its sending rate.
  The BBR congestion control protocol builds on ideas in TCP Vegas, and incorporates mechanisms that allows it compete fairly with TCP non-BBR senders. Google has replaced TCP Cubic with BBR.
  Other delay-based TCP congestion control protocols include TIMELY for data center networks, and Compound TCP (CTPC) and FAST for high-speed and long distance networks.

TCP splitting is a method of reducing perceived latency of cloud services (e.g. social networking, search, etc.) on a user's end system that is effective in scenarios when the user is far away from the datacenters that serve the dynamic content. Assuming each requests takes multiple RTTs per request[16], TCP splitting is done by deploying front-end servers close to the users, and then splitting the TCP connection from user to datacenter up into a connection from user to front-end server and a connection from front-end server to datacenter. The latter connection is persistent with a very large TCP congestion window. The multiple RTTs from user to front-end server are now negligibly small, such that total delay becomes one RTT from front-end server to datacenter plus processing time. TCP splitting also helps reduce TCP retransmission delays caused by losses in access network.

It's probably not even correct anymore to refer to "the" TCP protocol. There are many more versions of TCP: designed for use over wireless links, over high-bandwidth paths with large RTTs, for paths with packet re-ordering, and for short paths strictly within data centers. There are versions of TCP that implement different priorities among TCP connections competing for bandwidth at a bottleneck link, and for TCP connections whose segments are being sent over different source-destination paths in parallel. There are also variations of TCP that deal with packet acknowledgment and TCP session establishment/closure differently. Perhaps the only common features of these protocols is that they use the same segment format.

4.3 TLS

Neither TCP nor UDP provide throughput or timing guarantees. Also, neither TCP nor UDP provide security. However, *Transport Layer Security* (TLS) was developed as an enhancement of TCP to add security, that is implemented in the application layer. TLS does everything that traditional TCP and on top of that provides encryption, data integrity, and end-point authentication. TLS has its own socket API that first encrypts data before it passes it into the TCP socket (and decrypts data before it takes it from the TCP socket). It is implemented in highly optimized libraries and classes. TLS builds on the now-deprecated SSL (Secure Sockets Layer), and it's most prominent use is in HTTPS.

4.4 QUIC

If an application needs more services than those provided by UDP but does not want all of TCP's functionality, applications can simply "roll their own" transport protocol at the application layer by building on top of UDP. Quick UDP Internet Connections (QUIC) was designed by Google to improve the performance of transport-layer services for secure HTTP. So while it technically is an application-layer protocol, in practice it serves more as a transport-layer protocol. QUIC being application-layer also means that changes can be made at "application-update timescales", that is, much faster than TCP or UDP update timescales. QUIC's major features include:

- Connection-oriented and secure. QUIC combines the handshakes needed to establish connection state with those needed for authentication and encryption, which is an improvement over first needing to establish a TCP connection and then needing to establish a TLS connection over the TCP connection.

[16] Typically, a search query requires three TCP windows during slow start to deliver the response.

- Streams. A *stream* is an abstraction for the reliable, in-order bi-directional delivery of data between two QUIC endpoints. Several different application-level streams can be multiplexed through a single QUIC connection, and new streams can be quickly added. In the context of HTTP/3, there would be a different stream for each object in a Web page. Each connection has a connection ID, and each stream has a stream ID; both of which are contained in a QUIC packet header. Data from multiple streams may be contained within a single QUIC segment. This notion was pioneered by the Stream Control Transmission Protocol (SCTP), an earlier reliable, message-oriented protocol used in control plane protocols in 4G/5G cellular wireless networks.
- Reliable, TCP-friendly congestion-controlled data transfer. QUIC guarantees reliable data transfer of each stream separately, using acknowledgment mechanisms similar to TCP's. TCP needs to deliver its entire bytestream in-order, so a lost segment halts the entire connection, whereas a lost QUIC segment only impacts its stream.
QUIC's congestion control is based on TCP NewReno, a slight modification of TCP Reno

Google has deployed QUIC on many of its public-facing Web servers, in its mobile video streaming YouTube app, in its Chrome browser, and in Android's Google Search app. In 2017, more than 7% of Internet traffic was QUIC.

5 Application layer

The two predominant application architecture designs are:

- Client-server, which consists of an always-on host, called the server, which services requests from many other hosts, called clients. Clients do not directly communicate with each other. The server has a fixed, well-known IP address. Actually, nowadays "the server" is a network of distributed datacenters housing a large number of hosts acting as a powerful virtual server.
- Peer-to-peer (P2P), which consists of intermittently connected, private hosts called *peers* that communicate in pairs, instead of relying on always-on, dedicated servers. The decentralized nature means such networks are self-scalable (which means that additional peers offset the workload they generate by the server capacity they add) and cost effective, but also face challenges of security, performance, and reliability.

In general, the process that initiates communication is labeled as the client, and the process that waits to be contacted to begin the session is the server. This also goes for P2P peers. The application developer has control over which transport protocol to use (TCP or UDP) and some transport-layer parameters, but apart from that, everything in between the sockets is handled by the chosen transport protocol. A process is identified by its host's IP address and the port number of its socket. The port numbers ranging from 0 to 1023 are called *well-known port numbers* and are restricted; they are reserved for well-known applications protocols, standardized in RFC 1700 and listed here. There are also many port number that are not well-known, but are still conventionally used by popular applications, e.g. Microsoft Windows SQL servers on UDP port 1434).

An application layer protocol dictates the types of messages exchanged (e.g. request, response), message syntax (e.g. message fields and delineation), fields semantics, and rules for when and how a process sends messages and responds to messages. We will discuss various application layer protocols in the following subsections. Some application-layer protocols are public standards specified in RFCs (such as HTTP), but many others are proprietary and intentionally not available in the public domain (such as whatever protocol Skype uses).

There are two types of network application:

- Open applications implement a specified protocol standard (such as an RFC), and because of this, applications from multiple developers are able to interoperate. These applications should use the conventional port number associated with the protocol.
- Proprietary applications implement an application-layer protocol that has not been openly published in an RFC or elsewhere. One developer or company creates both the client and server applications, and other developers will not be able to develop code that interoperates with the application. These applications must avoid using reserved port numbers.

5.1 HTTP (World Wide Web)

The World Wide Web, shortly referred to as the Web, was the application that elevated the internet above all other data networks and brought it to the mainstream, and still is the most well-known internet application. *Web pages*, also called *documents* consist of *objects*, which are simply files (HTML, JPG, etc.) addressable by a single URL. Most web pages consist of a base HTML file/object and several referenced objects. Web objects are made available to the internet on web servers and are accessed by web browsers (the clients). Each *URL* has two components: the hostname of the Web server (`www.site.com`) and the object's path (`/path/to/object.css`). A Web server is always on, with a fixed IP address, and it services potentially millions of different browsers. Popular Web server software include Apache and Microsoft Internet Information Server.

The HyperText Transfer Protocol (HTTP) is the Web's application-layer protocol. It dictates how Web clients/browsers request Web pages from Web servers and how servers transfer Web pages to clients. HTTP uses TCP as its underlying transport protocol.

5.1.1 Messages

Request messages have different *methods* of obtaining objects. By far the most prevalent such method is `GET`, which simply requests an object and which generally looks something like this:

`GET /path/to/page.html HTTP/1.1` *Request line*, specifies method, object and HTTP version
`Host: www.site.com` Required by Web proxy caches
`Connection: close` Tells the server to close the TCP connection after sending (i.e. non-persistent)
`User-agent: Mozilla/5.0` Allows the server to send a browser-specific version
`Accept-language: fr` Allows the server to send a language-specific version, if it exists

The lines other than the request line are called *header lines*. Besides request and header lines, HTTP request messages also have an *entity body*. This is empty for `GET` request, but is used by `POST` requests. `POST` requests also request an object, but the specific contents of the Web page depend on the input that the user entered into HTML form fields, carried in the entity body. However, forms may also be submitted with a `GET` request with the input data in the requested URL, e.g. `/path/to/page.html?name=kortom&age=28`.
The `HEAD` method is like `GET` but without specifying an object, useful for debugging. `PUT` asks the server to upload an object to a specific path, useful for Web publishing tools or uploading files. Lastly, `DELETE` asks the server to delete the specified object.

An HTTP repsonse message may looks like:

`HTTP/1.1 200 OK` *Status line*, specifies version and status code + phrase
`Connection: close` The server will close the connection after sending
`Date: Tue, 18 Aug 2015 15:44:04 GMT` When the HTTP response was sent by the server
`Server: Apache/2.2.3 (CentOS)` Analogous to the `User-agent` header line
`Last-Modified: Tue, 18 Aug 2015 15:11:03 GMT` When the object was created or last modified
`Content-Length: 6821` The number of bytes in the object being sent
`Content-Type: text/html` Indicates the object file type (takes precedence over file extension)
`<data...>` The actual requested object

Common status codes and phrases are `200 OK` (all is fine), `404 Not Found`, `400 Bad Request` (generic error), `301 Moved Permanently` (the object has been moved and the new URL is specified in the `Location:` header of the response message; the client will automatically retieve it from there), and `505 HTTP Version Not Supported`.

5.1.2 Cookies

HTTP itself is a *stateless protocol*, meaning it does not maintain client information. This simplifies server design allows for high-performance. However, often servers want to serve content as a function of the user identity, which is why HTTPS uses *cookies* that allow sites to keep track of users. This is done in the following way: when a browser visits a site without a `Cookie:` header line, the server creates an entry in its back-end database indexed by a unique identification number, includes this identification number in the `Set-cookie:` header line in its HTTP response. Upon receipt, the browser appends a line to the special cookie file that it manages, associating the hostname with the identification number. Subsequent visits will have the browser extract the identification number from its cookie file to include it in the `Cookie:` header line in the HTTP request. The site can use this number and associate it with whatever in their database; shopping carts, account information, etc. Naturally, this entails a host of privacy concerns.

5.1.3 Web caches

A *Web cache*, or *proxy server*, is an intermediate host that serves HTTP object requests on behalf of one or more servers, which can reduce both response times as well as traffic on an institution's access link to the internet. A client's browser can be configured to point to the Web cache rather than the origin server itself. If the Web cache doesn't have the requested object, it asks the origin server for the object with a `GET` request, which it caches together with the datetime

in the `Last-modified:` header of the origin server's response before it forwards the requested object to the client. If the Web cache does have the requested object, it still contacts the origin server to verify that a file is up-to-date using a *conditional `GET`* requests, which is a HTTP `GET` request that contains a `If-Modified-Since:` header line. This header line contains the datetime of the `Last-modified:` header of the original response. The origin server will only send the object if it has a newer version, else it will send status code `304 Not Modified`.
A Web cache can be regarded as both a client and a server. Typically, Web caches are owned and maintained by ISPs, e.g. a university that configured all campus browsers to point to their Web cache (which caches stuff from the outside internet for these campus browsers), or a major residential ISP that preconfigures browsers that it ships to customers to point to their Web cache. The cost of installing and maintaining a Web cache is often much lower than upgrading an access link.

5.1.4 Versions

HTTP/1.0 employed non-persistent TCP connections, i.e. each Web object was send over a separate TCP connection. As of 2020, the majority of HTTP transactions take place over HTTP/1.1, which by default uses persistent connection[17]; the server leaves the TCP connection open after sending a response, and only closes it after a configurable time-out interval of inactivity.

HTTP/2 was standardized in 2015, and as of 2020, 40% of the top 10 million websites and all major browsers support it. The idea of HTTP/1.1 was to have a single persistent connection per Web page. However, many HTTP/1.1 browsers open up to six parallel TCP connections, for two reasons:

- *Head of Line blocking* (HOL blocking), which is the phenomenon that large objects (in terms of file size, e.g. a large video file) near the top of a base HTML document may block smaller objects below it, because objects are requested in order of appearance. This causes user-perceived delay. Web browsers therefore open multiple parallel connections to load the smaller elements sooner.
- TCP congestion control aims to give each TCP connection sharing a bottleneck link an equal share of the available bandwidth, but browsers can cheat and grab a larger portion of bandwidth by opening multiple parallel connections.

HTTP/2 aims to reduce the number of parallel TCP connections for transporting a single Web page. This not only reduces the number of sockets that need to be open and maintained at servers, but also allows TCP congestion control to operate as intended. HTTP/2's solution is to partition each requested object into *frames* of a fixed size, and to interleave the transmission of these frames, i.e. one frame of each object is send in turn, such that smaller objects are completely received sooner than larger objects. When a server wants to send an HTTP response, the response is broken down into frames by the framing sub-layer. The header field of the response becomes one frame, and the body of the message is broken down into one for more additional frames. At the client, arrriving frames are first reassembled into the original response messages at the framing sub-layer and then processed by the browser as usual. Similarly, a client's HTTP requests are broken into frames and interleaved. The framing sublayer additionally binary encodes the frames, because binary protocols are more efficient to parse, lead to slightly smaller frames, and are less error-prone.
HTTP/2 also supports message prioritization, which allows a client to assign a one-byte weight to requests denoting their importance, based on which the server can prioritize the sending of frames. The client also states each message's dependency on other messages by specifying the ID of the message on which it depends.
Another feature of HTTP/2 is the ability for a server to send multiple responses for a single request, i.e. a server analyzes a requested HTML page and immediately *pushes* the other objects that are required by the base HTML object rather than wait for explicit requests for each of these objects.

HTTP/3 is designed to operate over QUIC, which subsumes many of the features that HTTP/2 introduced (such as message interleaving), allowing for a simpler, streamlined design for HTTP/3. To summarize, the traditional secure HTTP protocol stack consisted of HTTP/2, which uses TLS, which uses TCP. The modern secure QUIC-based HTTP/3 protocol stack consists of a simplified/trimmed version of HTTP/2, which uses QUIC (taking over the functionalities cut from HTTP/2), which uses UDP. HTTP/3 is the combination of the trimmed HTTP/2 and QUIC. It was standardized in 2022, with 75% of web browsers and 26% of the top 10 million websites supporting it in 2022.

5.2 SMTP (E-mail transmission)

Electronic mail consists of e-mail applications (e.g. Microsoft Outlook) that each have an associated mail server (the name of which is what you put behind the `@` in e-mail addresses, e.g. `username@mailservername.com`[18]). The mail server sends, receives and stores the e-mails on behalf of the e-mail application. If a mail server cannot deliver a message,

[17] Though HTTP clients and servers can be configured to use non-persistent connections.

[18] Actually, this name is often a DNS alias for a more complicated canonical hostname, e.g. `mailservername.com` instead of `relay1.west-coast.mailservername.com`.

it holds the message in a message queue and reattempts every 30 minutes or so; if there is no success after several days, the server removes the message and notifies the sender with an e-mail. SMTP uses persistent TCP connections; multiple messages between mail servers can be over the same TCP connection.
The reason for using mail servers as intermediaries rather than having the e-mail applications communicate directly, is that a host needs to be on in order to receive e-mail, which is inconvenient for personal computers and devices. Also, mail servers can retry sending mail every 30 minutes if the recipient's mail server is unreachable. Mail servers are often shared between multiple users.

Simple Mail Transfer Protocol (SMTP) is much older than HTTP, which is evident in certain archaic characteristics. For example, it restricts the body (not just the headers) of all mail messages to simple 7-bit ASCII, which made sense in the early 1980s, but which now requires binary multimedia data to be encoded and decoded to ASCII while being sent over SMTP.

When a mail server sends mail, it acts as an SMTP client, and when it receives mail, it acts as an SMTP server. An e-mail client places a message in its mail server's (outgoing) message queue over either SMTP or HTTP, after which the client side of SMTP (that runs on the mail server) opens a TCP connection to the server side of SMTP (running on the recipient's mail server), and sends the message (after some SMTP handshaking), where the recipient's mail server places it in its inbox. An SMTP correspondence may look like this:

Server: `220 recipientname.edu`
Client: `HELO sendername.fr`
Server: `250 Hello sendername.fr, pleased to meet you`
Client: `MAIL FROM: <alice@sendername.fr>`
(Indicates start of the message.) Server: `250 alice@sendername.fr ... Sender ok`
Client: `RCPT TO: <bob@recipientname.edu>`
Server: `250 bob@recipientname.edu ... Recipient ok`
Client: `DATA`
(Indicates that the clients wants to transmit the message.) Server: `354 Enter mail, end with "." on a line by itself`
Client: `Dear Bob,`
Client: `This is the e-mail content.`
Client: `Kind regards!`
Client: `.` (A line consisting of a single period indicates the end of the message.)
Server: `250 Message accepted for delivery`
Client: `QUIT` (Closes the connection if no more messages need to be transferred.)
Server: `221 recipientname.edu closing connection`

Similarly to HTTP, each message has header lines and a (ASCII) message body:

`From: alice@sendername.fr`Mandatory header line.
`To: bob@recipientname.edu` Mandatory header line.
`Subject: Hotel booking.` Optional header line.

5.3 IMAP (E-mail access)

SMTP is a push protocol; it can only be used to send messages. In order for an e-mail application to retrieve messages from its mail server, it can use HTTP if it's a Web or phone application, but most desktop applications (such as Microsoft Outlook) use the *Internet Mail Access Protocol* (IMAP). Both methods allow for managing messages and folders on the mail server.

5.4 DNS (Hostname-to-IP mapping)

Hosts on the internet can be identified either by mnemonic hostnames (e.g. `website.com` and by IP adresses (e.g. 127.0.0.1). The former method is easiest for humans, but provide little information to routers of where the host resides within the internet. Routers therefore use IP addresses, which consist of four bytes and have a rigid hierarchical structure, meaning that from left to right, the bytes specify more fine-grained location data.
These hostnames need to be translated to IP addresses whenever a user requests a connection to the hostname. This requires a directory service that performs this lookup, which is the task of the internet's *Domain Name System* (DNS). DNS is both a distributed database implemented in a hierarchy of DNS servers[19] and an application-layer protocol that

[19] These servers are often UNIX machines running the Berkeley Internet Name Domain (BIND) software.

allows hosts to query this distributed database. The DNS protocol runs over UDP and uses port 53; it would be significantly slower if it had to establish a TCP connection for every request/response pair.

DNS servers can be roughly divided into four classes, the first three of which correspond to the three levels in the database hierarchy, and a fourth which does not, strictly speaking:

1. *Root* DNS servers. There are more than 1000 root servers scattered across the globe, which provide the IP addresses of the TLD servers. All of the root servers are copies of 13 different root servers, managed by 12 different organizations, and coordinated through the Internet Assigned Numbers Authority. They are listed here.
2. *Top-level domain* (TLD) DNS servers. For each of the top-level domains (e.g. `com`, `org`, `net`, etc.) and country top-level domains (e.g. `uk`, `nl`, `jp`, etc.) there is TLD server (cluster), which provide the IP addresses for authoritative DNS servers. For example, the company Verisign Global Registry Services maintains the TLD servers for the `com` top-level domain, and the company Educause does so for the `edu` top-level domain. All top-level domains are listed here.
3. *Authoritative* DNS servers. Every organization with publicly accessible hosts must have an authoritative DNS server – either self-maintained or externally leased – which houses DNS records that map the names of those hosts to IP addresses. Most universities and large companies implement and maintain their own primary and secondary (backup) authoritative DNS server.
   In actuality, this class may be a subhierarchy. For example, a company or university may have an authoritative DNS server per department, and one main [authoritative?] DNS server that provides the IP address of the specific department's authoritative DNS server.

- *Local* DNS servers. Handles DNS queries. A local DNS server is essentially a proxy that takes DNS queries from hosts and forwards and into the DNS server hierarchy. A host's local DNS server is supposed to be "close to" the requesting host. Each ISP has a local DNS server, and provides the host –typically through DHCP – with IP address(es) of one or more of its local DNS servers when a host connects to it. Institutional ISPs often host their own local DNS server on the same LAN. Actually, everyone can self-host a local DNS server, e.g. a Pi-hole on a Raspberry Pi.

While DNS is an application-level protocol in every sense, it provides a core internet function that is used by other user applications rather than being used by users directly.[20] When a hostname needs to be looked up, the client side of DNS in the host sends the query to its local DNS server. This query is called a *recursive query* because the hosts asks the local DNS server obtain the mapping on its behalf. The local DNS server first contacts one of the root servers, receiving an IP addresses for TLD servers for the queried hostname's top-level domain. This is an *iterative* query, because the reply is returned directly to the local DNS server. The local DNS server then contacts one of these TLD servers, receiving the IP address of an authoritative server for the organisation. Finally, the client contacts one of more of the authoritative servers, as many as are needed to receive the IP address for the hostname. In theory, any DNS query can be iterative or recursive, e.g. a chain of recursive queries through the hierarchy from top to bottom. In practice, queries typically follow the pattern just describes: with only the query from the host to the local DNS server recursive, and all queries from the local DNS server being iterative.

DNS provides a few other important services. DNS lookup incurs an often significant delay and increase in internet traffic (a regular query takes at least six messages: half of which are requests and half of which are replies). Therefore, DNS extensively exploits DNS caching; whenever a DNS server receives a DNS reply, it caches the mapping in its local memory. A local DNS server can also cache the IP addresses of TLD servers, which allows it to bypass the root DNS servers in a query chain (for this reason, only a very small fraction of DNS queries go through root DNS servers). Because hosts and their DNS mappings are by no means permanent, DNS servers discard cached information after a period of time (often set to two days).
DNS provides hostname aliasing for both Web servers and mail servers, which allows to specify aliases for complicated *canonical hostnames* (e.g. `sitename.com` instead of `relay1.west-coast.sitename.com`). When an alias is queried, DNS will retrieve the canonical hostname as well as the IP address. In fact, the MX resource record type permits a company's mail server and Web server to have identical (aliased) hostnames (see Section 5.4.1 below). DNS is also used to perform load distribution among replicated servers, which are multiple servers that serve the same content but run on distinct end systems with distinct IP addresses. Such servers may share an (alias) hostname, i.e. the DNS database associates this hostname with a set of IP addresses. DNS will rotate the ordering of the addresses within each reply, and because a client typically sends its HTTP request message to the first listed IP address, DNS rotation distributes the traffic among the replicated servers.

[20] On many UNIX-based machines, `gethostbyname()` is the function call that an application calls in order to perform the translation.

5.4.1 Resource records

The DNS servers that together implement the DNS distributed database store *resource records* (RRs), including RRs that provide hostname-to-IP mappings. Each DNS reply carries one or more RRs. Each RR is a four-tuple: `(Name, Value, Type, TTL)`, where `TTL` is time-to-live – the expiration date of the record – and the meaning of `Name` and `Value` depend on `Type` is one of the following:

- If `Type=A`, then `Name` is a hostname and `Value` is the IP address for the hostname; a regular mapping. If a DNS server is authoritative for a hostname, then it will have an `A` record for it. (A DNS server that is not authoritative for a hostname may still hold an `A` record for it in cache.) E.g. `(relay1.bar.foo.com, 145.37.93.126, A, ...)`.
- If `Type=NS`, then `Name` is a domain (such as `foo.com`) and `Value` is the hostname of an authoritative DNS server that knows how to obtain the IP addresses for hosts in that domain. This record is used to route DNS queries further along in the query chain. If a DNS server is not authoritative for a hostname, then it will have a `NS` record for the domain that includes the hostname and an `A` record with the IP address of the DNS server in the `Value` field of the `NS` record. E.g. `(foo.com, dns.foo.com, NS)`.
- If `Type=CNAME`, then `Value` is a canonical hostname for the alias hostname `Name`. This record can provide querying hosts the canonical name for a hostname. E.g. `(foo.com, relay1.bar.foo.com, CNAME)`.
- If `Type=MX`, then `Value` is the canonical name of a mail server that has an alias hostname `Name`. Such records allow the hostnames of mail servers to have simple aliases. Note that by using both the MX and CNAME record, a company can have the same aliased name for its mail server and for one of its other servers (such as its Web server), because a DNS client can query those separately. E.g. `(foo.com, mail.bar.foo.com, MX)`.

5.4.2 Message format

Both DNS queries and replies have the same format:

- Header (12 bytes)
  - Message ID (2 bytes). A reply is given the same ID as its query, so that the client can match received replies with sent queries.
  - Flags (4 bits)
    - Whether the message is a query (0) or a reply (1)
    - Whether the DNS server is authoritative for a queried name (only relevant for replies)
    - Whether the DNS server should perform recursion when it doesn't have the record (only relevant for queries)
    - Whether the DNS server supports recursion (only relevant for replies)
  - Number-of fields:
    -
- Body
  - Questions; a variable number of questions, each of which includes the hostname being queried and the type of the question (`A`, `NS`, `CNAME`, or `MX`).
  - Answers; a variable number of resource records. Per question in the query, a reply may include multiple resource records, since a hostname can have multiple IP addresses.
  - Authority; a variable number of resource records for authoritative servers
  - Additional information; a variable number of other helpful resource records. For example, for an `MX` reply, this section contains an `A` record with the IP address for the queried canonical hostname of the mail server.

The `nslookup` program available in most Windows and UNIX platforms allows to send a DNS query message directly to to some DNS server, after which it will display the resource records in the reply.

5.4.3 Domain registration

If you want to register a domain, you pay a *registrar*, a commercial entity that verifies the uniqueness of the domain name, enters the domain name into the DNS database. You provide the registrar with your desired domain name and the hostnames and IP addresses of your primary and secondary authoritative DNS servers, and the registrar will make sure that a `NS` and `A` record are entered into the relevant TLD servers. Prior to 1999, a single registrar, Network Solutions, had a monopoly on domain name registration for `com`, `net`, and `org` domains, but now there are many registrars, which the Internet Corporation for Assigned Names and Numbers (ICANN) accredits. A complete list of accredited registrars is available here.
You'll have to make sure that the `A` resource record for your Web server and `MX` resource record for your mail server are entered into your authoritative DNS servers. Until recently, the contents of each DNS server were configured statically, e.g. from a configuration file. Recently, an UPDATE option has been added to the DNS protocol to allow data to be dynamically added or deleted from the database via DNS messages.

5.4.4 DNSSEC

DNSSEC is a secured version of DNS that addresses many possible DNS attacks and is gaining popularity in the Internet.

5.5 BitTorrent (P2P file distribution)

As of writing, BitTorrent is the most popular peer-to-peer file distribution protocol. The BitTorrent ecosystem is wildly successful, especially for piracy, with millions of simultaneous peers actively sharing files in hundreds of thousands of torrents.

In BitTorrent lingo, a *torrent* is a group of peers sharing the same particular (set of) file(s). Peers in a torrent download and redistribute equisized chunks (typically of size 256 KBytes) of the file from and to one another. Each torrent has an infrastructure node called a *tracker*, which keeps track of the peers in the torrent. When a new peer joins a torrent, it registers itself with the tracker (and periodically informs the tracker that it is still in the torrent), after which the tracker randomly selects a subset of peers and sends the IP addresses of these peers to the new peer, which then attempts to establish concurrent TCP connections with all these peers. This set of "neighbouring" peers to which the new peer is connected may fluctuate over time, because peers may leave and other peers may attempt to establish TCP connections with the new peer. The new peer will periodically ask each of its neighboring peers for the list of the chunks they have, and will – based on this knowledge – issue requests for chunks it currently does not have. It will request the rarest chunks first; the chunks that occur the least among its neighbors.

In determining which requests it responds to, a peer continually measures the instantaneous throughput per neighbour and reciprocates the four neighbours with the highest such throughput; in BitTorrent lingo, these four neighbours are said to be *unchoked*. The set of four unchoked neighbours is recalculated every ten seconds. Additionally, every thirty seconds, one additional *optimistically unchoked* neighbor is picked at random and also send chunks. The sending peer may now become an unchoked neighbour of this optimistically unchoked neighbour, and get send chunks back. In other words, an optimistically unchoked peer is a potential trading partner, and if the two peers are satisfied with the trading, they will be mutually unchoked neighbours until one of them finds a better partner. The effect is that peers capable of uploading at compatible rates tend to find each other. The random neighbor selection also allows new peers to get chunks, so that they can have something to trade. Besides the five unchoked neighbours, all other neighbours are *choked*. This incentive mechanism is referred to as tit-for-tat.

Once a peer has acquired the entire file, it may (selfishly) leave the torrent, or (altruistically) remain in the torrent and continue to upload chunks to other peers. Also, any peer may leave the torrent at any time with only a subset of chunks, and later rejoin the torrent.
BitTorrent has a number of interesting mechanisms not discussed here, including pieces (mini-chunks), pipelining, random first selection, endgame mode, and anti-snubbing.

5.6 DASH (Video streaming)

From a networking perspective, the most salient characteristic of video is arguably its bit rate, ranging from 100 kbps for low-quality video to more than 10 Mbps for 4K. In order to provide continuous play, the network must provide an average throughput at least as large as the bit rate of the video.

With HTTP streaming, a video file is simply send in a HTTP response message to a `GET` request. On the client side, the bytes are collected in a client application buffer and once the number of bytes in this buffer exceeds a predetermined threshold, the client streaming application begins playback: it periodically grabs video frames from the client application

buffer, decompresses the frames, and displays them on the user's screen. This has a major shortcoming: all clients receive the same video encoding, despite large variations in bandwidth available to clients.
This led to *Dynamic Adaptive Streaming over HTTP* (DASH). In DASH, a video is encoded multiple times at different bit rate (and therefore quality) levels and stored at different URLs on the Web server. The Web server also has a *manifest file* that lists all versions along with bit rate and URL. The client first requests the manifest file, and then selects one chunk of video (a few seconds in length) at a time by specifying a URL and byte range in an HTTP `GET` request message for each chunk. While downloading a chunk, the client also measures the received bandwidth and runs a rate determination algorithm to select the video version from which to request the next chunk; if measured receive bandwidth is high and the client has a lot of video buffered, it will choose a chunk from a high-bitrate version, otherwise it will choose a low-bitrate chunk. DASH, like HTTP, uses TCP.

5.7 Telnet

Telnet, running over TCP, is an application-layer protocol for remote login. Unlike the bulk data transfer applications discussed so far, Telnet is an interactive application. However, its popularity is waning in favour of SSH, since data sent in a Telnet connection (including passwords) are not encrypted, making Telnet vulnerable to eavesdropping attacks.

A client initiates a session and remotely operates a terminal running on a server. Each character typed by the user at the client will be sent to the remote host, which will "echo back" a copy of each character, in turn displayed on the client's screen.

5.8 SSH

SSH uses TCP.

6 Cybersecurity

A *denial-of-service* (DoS) attack renders a network, host, or other piece of infrastructure unusable by legitimate users. In a *distributed denial-of-service* (DDoS) attack, multiple sources blast traffic at the target, which is much harder to defend against.

A *packet sniffer* is a passive receiver that records a copy of every packet that flies by. The best defenses against packet sniffing involve cryptography.

IP spoofing is an attack in which packets with a false source address are injected into the network. For example, an unsuspecting receiver may take such a packet and update its forwarding table based on the false source. To solve this problem, we will need end-point authentication, that is, a mechanism that will allow us to determine with certainty if a message originates from where we think it does.

7 Outroduction

7.1 Internet, Part II

7.1.1 CDNs

Content Distribution Networks (CDNs) are networks of many geographically distributed servers with caches throughout the internet, thereby localizing much of the traffic. A CDN stores copies of (Web) content in its servers, and attempts to direct each user request to the nearest – in terms of number of links and routers – CDN location. Just like most forms of caching, CDNs do not push all Web content to their servers, but use a simple pull strategy: only if a client requests content from a server (that doesn't have it yet) will it retrieve the content from a central repository or from another cluster and cache a copy (after which it provides the content to the requestor). If the cache becomes full, the server deletes the least popular content. CDNs are slightly similar to tier-1 ISPs.

CDNs typically adopt one of two different server placement philosophies:

- *Enter deep*: thousands of small CDN server clusters are deployed in access ISPs all over the world, in order to get close to end users. Difficult to maintain due to the number of clusters.
- *Bring home*: tens of large CDN server clusters are deployed typically in Internet Exchange Points. Requires less aintenance and management overhead, at the expense of higher delay and lower throughput to end users.

A *private CDN* is owned and operated by the content provider itself. Two case studies:

- Google's CDN has three tiers of server clusters:
  - 19 major data centers in North America, Europe, and Asia, each with around hundreds of thousands of servers, responsible for serving dynamic (and often personalized) content.
  - Around 90 "bring home" clusters in IXPs scattered throughout the world, each with hundreds of servers, responsible for serving static content (e.g. YouTube videos).
  - Hundreds of "enter deep" clusters inside access ISPs, each with tens of servers within a single rack, responsible for TCP splitting and serving static content (e.g. static portions of Web pages of search results).

  These datacenters are interconnected via Google's private TCP/IP network, which is separate from the internet and only carries traffic to/from Google servers. In this way, Google bypasses tier-1 ISPs by peering with lower-tier ISPs, and only uses tier-1 ISPs to connect to ISPs it otherwise can't reach.
  When a user uses Google, often the query is first sent over the local ISP to a nearby enter deep cache, from where the static content is retrieved, and while providing this static content, the nearby cache also forwards the query over Google's private network to one of the mega data centers, from where the personalized content is retrieved. For example, a YouTube video itself may come from one of the bring-home caches, whereas portions of the Web page surrounding the video may come from the nearby enter-deep cache, and the advertisements surrounding the video come from the data centers. In short, except for the local ISPs, the Google cloud services are largely provided by a network infrastructure that is independent of the public internet.

- Netflix's Web site (including its user registration and login, billing, movie catalogue browsing and searching, and movie recommendation system) and content processing (the uploading of studio master versions, generation of many different DASH formats and bitrates, and uploading to the CDN) all run on Amazon servers, while video streaming itself happens from Netflix's private CDN. Instead of the usual pull-caching, Netflix uses push-cashing; Netflix distributes videos to its CDN servers by pushing the videos during off-peak hours. This strategy makes sense, since Netflix has a relatively small video library compared to something like YouTube. Netflix's CDN has server racks installed both in over 200 IXPs and within hundreds of residential ISPs themselves. Netflix provides instructions to potential ISP partners on how to install a free server rack here. Each server in a rack has several 10 Gbps Ethernet ports and over 100 terabytes of storage. IXP installations often have tens of servers and contain the entire Netflix streaming video library, while Netflix pushes only the most popular videos (determined on a day-to-day basis) to those locations that cannot hold the entire library. Because Netflix's CDN distributes only video (and not Web pages), Netflix has been able to simplify and tailor its CDN design.

  When a user selects a movie to play, the Netflix software running in the Amazon cloud first determines which of its CDN servers have copies of the movie, and the software then determines the "best" of these servers for that client request. If one of them is in a server rack in the same residential ISP as the client, this server is typically selected. If not, a server at a nearby IXP is typically selected. The Netflix software in the Amazon cloud then sends the client the IP address of the server as well as a manifest file, after which the client and that CDN server then directly interact using a proprietary version of DASH. In other words, Netflix does not need DNS redirect to connect clients to CDN servers; instead, the Netflix software directly tells the client to use a particular CDN server.

A *third-party CDN* is a company whose primary business is maintaining a CDN and providing content on behalf of other content providers. Most third-party CDNs take advantage of DNS to intercept and redirect requests. In the example of a streaming service, if a user clicks on the link for a particular video, e.g. `http://video.streamingservice.com/6Y7B23V`, the user's host sends a DNS query for `video.streamingservice.com`. The user's Local DNS server relays the DNS query to an authoritative DNS server for StreamingService, which observes the string `video` in the hostname. To "hand over" the DNS query to StreamingService's CDN provider ThirdPartyCDN, instead of returning an IP address, the StreamingService authoritative DNS server returns a hostname in the ThirdPartyCDN's domain, e.g. `a1105.ThirdPartyCDN.com`. The user's Local DNS server then sends a second query for `a1105.ThirdPartyCDN.com`, and ThirdPartyCDN's DNS system eventually returns the IP addresses of a ThirdPartyCDN content server. This content server is chosen based on the IP address of the local DNS server; CDNs generally employ proprietary cluster selection strategies.[21] The local DNS server forwards the IP address of the content-serving CDN node to the user's host, after which the client establishes a direct TCP connection with that server and issues an HTTP `GET` request for the video. (If DASH is used, the server will first send the manifest file with a list of URLs for each version of the video.)

[21] A simple strategy is to assign the client to the cluster that is geographically closest, using commercial geo-location databases. This performs well in most cases, but sometimes the geographically closest cluster may not be the closest cluster in terms of the length or number of hops of the network path. Also, in some cases, the local DNS server may be far from its client. Another strategy takes current traffic conditions into account: CDNs can perform periodic real-time measurements of delay and loss performance between their clusters and clients, e.g. by having each of its clusters periodically send ping messages or DNS queries to all of the local DNS servers around the world. However, many LDNSs are configured to not respond to such probes.