Outline

• Big overall picture
• Then bottom-up
Basic Network Abstraction

• A process can create “endpoints”
• Each endpoint has a unique address
• A message is a byte array
• Processes can:
  • receive messages on endpoints
  • send messages to endpoints

Just another form of I/O
Network “protocol”

Agreement between processes about messages

Message:

**Syntax:** Layout of bits, bytes, fields, etc.
- message format

**Semantics:** what fields, messages mean

**Example:**
- HTTP “get” requests and responses
Network Layering

Network abstraction is usually *layered*

- Like Object Oriented-style inheritance

<table>
<thead>
<tr>
<th>Application</th>
<th>Presentation</th>
<th>Session</th>
<th>Transport</th>
<th>Network</th>
<th>Link</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7-Layer ISO/OSI reference model (1970’s)  
Current 5-Layer Internet Protocol Stack
## Internet Protocol Stack

<table>
<thead>
<tr>
<th>Layer</th>
<th>Functions</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application</strong></td>
<td>Exchanges <strong>messages</strong></td>
<td>HTTP, FTP, DNS, SSH, Skype, ...</td>
</tr>
<tr>
<td><strong>Transport</strong></td>
<td>Transports messages; exchanges <strong>segments</strong></td>
<td>TCP, UDP</td>
</tr>
<tr>
<td><strong>Network</strong></td>
<td>Transports segments; exchanges <strong>datagrams</strong></td>
<td>IP, ICMP (ping)</td>
</tr>
<tr>
<td><strong>Link</strong></td>
<td>Transports datagrams; exchanges <strong>frames</strong></td>
<td>Ethernet, WiFi</td>
</tr>
<tr>
<td><strong>Physical</strong></td>
<td>Transports frames; exchanges <strong>bits</strong></td>
<td>wires, signal encoding</td>
</tr>
</tbody>
</table>

We can call them all **packets**
Who does what?

<table>
<thead>
<tr>
<th>Who</th>
<th>What</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>HTTP, FTP, DNS</td>
</tr>
<tr>
<td>Transport</td>
<td>TCP, UDP</td>
</tr>
<tr>
<td>Network</td>
<td>IP, ICMP (ping)</td>
</tr>
<tr>
<td>Link</td>
<td>Ethernet, WiFi</td>
</tr>
<tr>
<td>Physical</td>
<td>wires, signal encoding</td>
</tr>
</tbody>
</table>

- Each host has 1+ Network Interface Cards (NIC)
- Attaches into host’s system buses
- Combination of hardware, software, firmware
Layers support

Modularity and Abstraction

Each layer:
• relies on services from layer below
• exports services to layer above

Interfaces between layers:
• Hide implementation details
• Ease maintenance, updates
  • change of implementation of layer’s service transparent to rest of system
Internet, The Big Picture

Routers

Endpoints
The Big Picture

Application → **messages** → **segments** → **datagrams** → **frames** → **bits** → Physical

Application-specific multiplexing

Transport

Segments → Ports (http: 80, DNS: 53, Telnet: 23)

Network

Datagrams → IP addresses (192.168.100.254)

Data Link

Frames → MAC Addresses (00:12:F4:AB:0C:82)

Physical
The Big Picture

Application

Transport

Network

Data Link

Physical

messages

segments

datagrams

frames

bits

Router1

Router2

Network

Data Link

Physical
Encapsulation

**source**
- message
- segment
- datagram
- frame

**destination**
- message
- segment
- datagram
- frame

**Headers**
- Transport
  - src & dst ports + …
- Network
  - src & dest IP addr + …
- Link
  - src & dest MAC addr + …

**Diagram**
- Source and destination nodes with encapsulation layers.
- Router in the middle with encapsulation details.
End-to-End Argument

• Application-specific properties are best provided by the applications, **not the network**
  – Guaranteed, or ordered, packet delivery, duplicate suppression, security, *etc.*

• Internet performs best effort packet routing
  – Higher-level applications do the rest
End-to-End Example

Should the network guarantee packet delivery?

Consider: a file transfer program (read file from disk, send it, receiver reads packets & writes them to disk)

- Q: If network guarantees delivery, wouldn’t applications be simpler? (no retransmissions!)
- A: no, still need to check that file was written to remote disk intact. Just because a message was delivered does not imply that it was acted upon.

A check is necessary if nodes can fail.

→ Applications need to be written to perform their own retransmits (or at least report an error)

Why burden the network with properties that can, and must, be implemented at the periphery?
Some issues…

- How do endpoints find each other?
- What does a packet look like?
- Can packets be lost or duplicated?
- Can packets be jumbled?
- How large can packets be?
- …
Properties of a network connection

• Bandwidth (aka throughput):
  • #bytes / second

• Latency
  • How long does it take a bit to travel from one end to the other?

• Packet drop ratio
  • probability of packet loss

• Packet drop rate
  • #packets dropped / second
Link Layer: Local Area Networking (LAN) and Ethernet
Link Layer

• Each host has one or more NICs
  • Network Interface Cards
    – Ethernet, 802.11, etc.
• Each NIC has a MAC address
  • Media Access Control address
  • Ethernet example: b8:e3:56:15:6a:72 (48 bits)
  • Unique to network instance
    – often even globally unique
• Packets are frames
Example: Ethernet

- 1976, Metcalfe & Boggs at Xerox
  - Later at 3COM
- Based on the Aloha network in Hawaii
- Named after the “luminiferous ether”
- Centered around a broadcast bus
- Simple link-level protocol, scales pretty well
- Tremendously successful
- Still in widespread use
  - many orders of magnitude increase in bandwidth since early versions
# Ethernet basics

An Ethernet packet

<table>
<thead>
<tr>
<th>Destination Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Address</td>
</tr>
<tr>
<td>Type</td>
</tr>
<tr>
<td>…Payload…</td>
</tr>
<tr>
<td>Checksum</td>
</tr>
</tbody>
</table>
“CSMA/CD”

- **Carrier Sense**
  - Listen before you speak

- **Multiple Access**
  - Multiple hosts can access the network

- **Collision Detect**
  - Detect and respond to cases where two hosts collide
Sending packets

- Carrier sense, broadcast if ether is available
Collisions

• What happens if two people decide to transmit simultaneously?
Collision Detection & Retransmission

- The hosts involved in the collision stop data transmission, sleep for a while, and attempt to retransmit.
- How long they sleep is determined by how many collisions have occurred before:
  - Exponential back-off, but randomized.
- They abort after 16 retries, hence no guarantee that a packet will get to its destination.
CRC Checksum
(Cyclic Redundancy Check)

• Basically a hash function on the packet
  • Same as CRC checksums on disk blocks
• Added to the end of a packet
• Used to detect malformed packets, e.g. electrical interference, noise
Ethernet Features

• Completely distributed
  – No central arbiter

• Inexpensive
  – No state in the network
  – No arbiter
  – Cheap physical links (twisted pair of wires)
Ethernet Problems

• Gets slow when there’s lots of contention
• Hosts are trusted to only listen to packets destined for them
  – But the data is available for all to see
    • All packets are broadcast on the wire
    • Can place Ethernet NIC in “promiscuous mode” and listen
Switched Ethernet

- Today’s Ethernet deployments are much faster
- In wired settings, *Switched Ethernet* has become the norm
  - All hosts connect to a switch
  - Each p2p connection consists of two mini Ethernet set-ups
  - More secure, snooping more difficult
  - Switches organize into a spanning tree
    - Outside the scope of this class

- Not to be confused with Ethernet *Hub*
  - A hub simply connects the wires
Wireless

• 802.11 protocols inherit many of the Ethernet concepts
• Full compatibility with Ethernet interface
  • Same address and frame formats
• Be aware of security vulnerabilities
  • WPA[123] tries to emulate security of switched Ethernet
Lessons for LAN design

• Best-effort delivery simplifies network design
• A simple, distributed protocol can tolerate failures and be easy to administer
Network Layer
Network Layer

• There are lots of Local Area Networks
  • each with their own
    – address format and allocation scheme
    – packet format
    – LAN-level protocols, reliability guarantees
• Wouldn’t it be nice to tie them all together?
  • Nodes with multiple NICs can provide the glue!
  • Standardize address and packet formats
• This gives rise to an “Internetwork”
  • aka WAN (wide-area network)
Internetworking Origins

- Expensive supercomputers scattered throughout the US
- Researchers scattered differently throughout the US
- Needed a way to connect researchers to expensive machinery
Internetworking Origins

• Department of Defense initiated studies on how to build a resilient global network (60s, 70s)
  – How do you coordinate a nuclear attack?
• Interoperability and dynamic routing are a must
  – Along with a lot of other properties
• Result: Internet (orig. ARPAnet, then NSFnet)
• A complex system with simple components
Internet Overview

• Every NIC is assigned, and identified by, an IP address
  • NIC is Network Interface Card
  • IP is Internet Protocol
• Packets are called datagrams
• Each datagram contains a header that specifies the destination address
• The network routes datagrams from the source NIC to the destination NIC
IP

• Internetworking protocol
  – Network layer
• Common address format
• Common packet format for the Internet
  – Specifies what packets look like
  – *Fragments* long packets into shorter packets
  – *Reassembles* fragments into original shape
• IPv4 vs IPv6
  – IPv4 is what most applications use
  – IPv6 more scalable and clears up some of the messy parts
IP: Narrow Waist

Application Layer
- HTTP
- SMTP
- NTP
- DHCP
- DNS
- SIP

Transport Layer
- TCP
- UDP

Network Layer

Data Link Layer
- LTE
- 3G
- WiFi
- Ethernet
- DOCSIS
- DSL

Physical Layer
- Radio
- Coaxial
- Fiber
- Twisted Pair

from: http://if-we.clients.labzero.com/code/posts/what-title-ii-means-for-tcp/
IP Addressing

• Every (active) NIC has an IP address
  – IPv4: 32-bit, e.g. 128.84.254.43
  – IPv6: 128-bit (but only 64 bits “functional”)
  – We use IPv4 unless specified otherwise…

• Each Internet Service Provider (ISP) owns a set of IP addresses
• ISPs assign IP addresses to NICs
• IP addresses can be re-used
• Same NIC may have different IP addresses over time
IP “subnetting”

- An IP address consists of a prefix of size $n$ and a suffix of size $32 - n$
  - Either specified by an integer, $0 \leq n \leq 32$
    - e.g., 128.84.32.00/24 or 128.84.32/24
  - Or a “netmask”
    - e.g., 255.255.255.0 or 0xFFFFFFFF00 (in case $n = 24$)
- A “subnet” is identified by a prefix and has $2^{32-n}$ addresses
  - Suffix of “all zeroes” or “all ones” reserved for broadcast
  - Big subnets have a short prefix and a long suffix
  - Small subnets have a long prefix and a short suffix
Addressing & DHCP

DHCP is used to learn IP address and subnet mask (and more)

DHCP = Dynamic Host Configuration Protocol

“I just got here. My physical address is 1a:34:2c:9a:de:cc. What’s my IP?”

“Your IP is 128.84.96.89 for the next 24 hours”
DHCP

- Each LAN (usually) runs a DHCP server
  - you probably run one at home inside your “router box”
- DHCP server maintains
  - the IP subnet that it owns (say, 128.84.245.00/24)
  - a map of IP address <-> MAC address
    - possibly with a timeout (called a “lease”)
- When a NIC comes up, it broadcasts a DHCPDISCOVER message
  - if MAC address in the map, respond with corresponding IP address
  - if not, but an IP address is unmapped and thus available, map that IP address and respond with that
- DHCP also returns the netmask
- Note: NICs can also be statically configured and don’t need DHCP in that case
Addressing & ARP

- ARP is used to discover MAC addresses on same subnet
  - ARP = Address Resolution Protocol

“I’m at 1a:34:2c:9a:de:cc”

“What is the physical address of the host named 128.84.96.89”
Scale?

- ARP and DHCP only scale to single subnet
- Need more to scale to the Internet!
IPv4 packet layout

<table>
<thead>
<tr>
<th></th>
<th>Version</th>
<th>IHL</th>
<th>TOS</th>
<th>Total Length</th>
<th>Identification</th>
<th>Flags</th>
<th>Fragment Offset</th>
<th>TTL</th>
<th>Protocol</th>
<th>Header Checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Identification</td>
<td>Flags</td>
<td>Fragment Offset</td>
<td>TTL</td>
<td>Protocol</td>
<td>Header Checksum</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Source Address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Destination Address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Options</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Payload</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
IP Header Fields

- Version (4 bits): 4 or 6
- IHL (4 bits): Internet Header Length in 32-bit words
  - usually 5 unless options are present
- TOS (1 byte): type of service (not used much)
- Total Length (2 bytes): length of packet in bytes
- Id (2 bytes), Flags (3 bits), Fragment Offset (13 bits)
  - used for fragmentation/reassembly. Stay tuned
- TTL (1 byte): Time To Live. Decremented at each hop
- Protocol (1 byte): TCP, UDP, ICMP, ...
- Header Checksum (2 bytes): to detect corrupted headers
- Options: mostly never used
IP Fragmentation

• Networks have different maximum packet sizes
• “MTU”: Maximum Transmission Unit
• High-level protocols could try to figure out the minimum MTU along the network path, but
  – Inefficient for links with large MTUs
  – The route can change underneath
• Consequently, IP can transparently fragment and reassemble packets
IP Fragmentation Mechanics

- Source assigns each datagram an “identification”
- At each hop, IP can divide a long datagram into N smaller datagrams
- Sets the More Fragments bit except on the last packet
- Receiving end puts the fragments together based on Identification and More Fragments and Fragment Offset (times 8)
Routing
The Internet is Big...
Routing

• How do we route messages from one machine to another?
• Subject to
  – churn
  – efficiency
  – reliability
  – economical considerations
  – political considerations
Internet Protocol (IP)

- The Internet is subdivided into disjoint Autonomous Systems (AS)

Graph of subgraphs
Autonomous Systems

• Each AS is a routing domain in its own right
  • has a private IP network
  • runs its own routing protocols
  • may have multiple IP subnets
    – each with their own IP prefix
• has a unique “AS number”
• ASs are organized in a graph
  • routing between ASs using BGP (Border Gateway Protocol)
Thus routing is hierarchical!

Three steps:
1. A packet is first routed to an “edge router” at the source AS---using the internal routing protocol used by the source AS
2. Next the packet is routed to an edge router at the destination AS---determined by the destination address prefix---using BGP
3. The destination AS’s edge router then forwards the packet to its ultimate destination---determined by the address suffix---using the internal routing protocol used by the destination AS
Internet Routing, observations

• There are no longer special “government” routers that route between ASs. Instead, each AS has one or more “edge routers” that are connected by interdomain links.

• Two types:
  • **Transit AS**: forwards packets coming from one AS to another AS
  • **Stub AS**: has only “upstream” links and does not do any forwarding
What’s an ISP?

• An ISP (Internet Service Provider) is simply an AS (or collection of ASs) that provides, to its customers (which may be people or other ASs), access to the “The Internet”
• Provides one or more PoPs (Points of Presence) for its customers.
Routers (Layer-3 Switches)

• Connects multiple LANs (subnets)

• Two classes:
  • Edge or Border router: Resides at the edge of an AS, and has two faces
    – one faces outside to connect to one or more per edge router in other ASs
    – one faces inside, connecting to zero or more other routers within the same AS

• Interior router:
  – has no connections to routers in other ASs
The Big Picture

Application → messages → Application

Transport → segments → Ports (http: 80, DNS: 53, Telnet: 23)

Network → datagrams → IP addresses (192.168.100.254)

Data Link → frames → MAC Addresses (00:12:F4:AB:0C:82)

Physical → bits → Physical

Datagrams, segments, messages, frames, bits, ports, IP addresses, MAC addresses, application-specific multiplexing.
Internet, The Big Picture

Endpoints

Routers
Routing Table

- Maps IP address to interface or port and to MAC address
- Longest Prefix Matching
- Your laptop/phone has a routing table too!

<table>
<thead>
<tr>
<th>Address</th>
<th>IF or Port</th>
<th>MAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>128.84.216/23</td>
<td>en0</td>
<td>c4:2c:03:28:a1:39</td>
</tr>
<tr>
<td>127/8</td>
<td>lo0</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>128.84.216.36/32</td>
<td>en0</td>
<td>74:ea:3a:ef:60:03</td>
</tr>
<tr>
<td>128.84.216.80/32</td>
<td>en0</td>
<td>20:aa:4b:38:03:24</td>
</tr>
<tr>
<td>128.84.217.255/32</td>
<td>en0</td>
<td>ff:ff:ff:ff:ff:ff</td>
</tr>
</tbody>
</table>
Routing Loops?

• In steady state, there should be no routing loops
• But steady state is rare. Routing tables are constantly updated.
• If routing tables are not in sync, loops can occur.
• IP packets maintain a maximum hop count (TTL) that is decreased on every hop until 0 is reached, at which point a packet is dropped.
Router Function: Longest Prefix

often implemented in hardware

for ever:

receive IP packet \( p \)

if isLocal(\( p \).dest): return localDelivery(\( p \))

if --\( p \).TTL == 0: return dropPacket(\( p \))

\( matches = \{ \} \)

for each entry \( e \) in routing table:

    if \( p \).dest & \( e \).netmask == \( e \).address & \( e \).netmask:
        \( matches \).add(\( e \))

\( bestmatch = matches \).maxarg(\( e \).netmask)

forward \( p \) to \( bestmatch \).port/bestmatch.MAC
How are these routing tables constructed?

• For end-hosts, mostly DHCP and ARP as discussed before
• For routers, using a “routing protocol”
  • take Prof. Agarwal’s networking course!
Network Address Translation

• IPv6 adoption is very slow, and IPv4 addresses have run out
• NAT allows entire sites to use a single globally routable IPv4 address for a collection of machines
  • exploits sparsity of the 16-bit TCP/UDP port number space
  • combined with “private IP addresses” (see next slide)
• A “NAT box” keeps a table that maps global TCP/IP addresses into local ones
• Overwrites the local source address with the globally addressable address
“Private” IP addresses

• The IPv4 addresses 10.x.x.x and 192.168.x.x are freely available for any LAN to use

• Many machines have the IP address 192.168.0.100, for example

• (but never on the same LAN)
From your laptop to Google…

- **NIC (your laptop)**: 192.168.1.100
- **NIC 1 (inside)**: 192.168.1.1
- **NIC 2 (outside)**: 128.84.34.124
- **NIC (Google)**: 74.125.141.147

**NAT**

- **dst**: 74.125.141.147:80
- **src**: 192.168.1.100:4410

Internet

- **dst**: 74.125.141.147:80
- **src**: 128.84.34.124:123
Vice versa: punching holes or “game ports”

• When an external host tries to send a message to one of your machines in your house, it first arrives at the NAT box
  • Because you advertise your global IP address
  • How does the NAT box know which of your machines to forward the message to?
• Answer: a table. It is indexed by the destination TCP or UDP port in the message
Loopback Interface

• 127.0.0.1/8
• Like a mini-LAN consisting of only the host itself
• Entirely virtual – no hardware required
• Useful for communicating between processes on the same machine
Transport Layer: UDP & TCP

Several figures in this section come from “Computer Networking: A Top Down Approach” by Jim Kurose, Keith Ross
Transport services and protocols

• Provide **logical** communication between processes on different hosts
• Run in end systems
  • **Sender**: packages messages into segments, passes to network layer
  • **Receiver**: turns segments into messages, passes to application layer

App chooses protocol it wants (e.g., TCP or UDP)
Transport services and protocols

User Datagram Protocol (UDP)
- unreliable, unordered delivery
- no connection set-up
- short application messages
- no-frills extension of best-effort IP

Transmission Control Protocol (TCP)
- reliable, in-order delivery
- session-based / connection set-up
- byte stream
  - ”a single but unbounded message”
- congestion control
- flow control

Services not available:
- delay guarantees
- bandwidth guarantees
## Applications & their transport protocols

<table>
<thead>
<tr>
<th>Application</th>
<th>Application-Layer Protocol</th>
<th>Underlying Transport Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electronic mail</td>
<td>SMTP</td>
<td>TCP</td>
</tr>
<tr>
<td>Remote terminal access</td>
<td>Telnet</td>
<td>TCP</td>
</tr>
<tr>
<td>Web</td>
<td>HTTP</td>
<td>TCP</td>
</tr>
<tr>
<td>File transfer</td>
<td>FTP</td>
<td>TCP</td>
</tr>
<tr>
<td>Remote file server</td>
<td>NFS</td>
<td>Typically UDP</td>
</tr>
<tr>
<td>Streaming multimedia</td>
<td>typically proprietary</td>
<td>UDP or TCP</td>
</tr>
<tr>
<td>Internet telephony</td>
<td>typically proprietary</td>
<td>UDP or TCP</td>
</tr>
<tr>
<td>Network management</td>
<td>SNMP</td>
<td>Typically UDP</td>
</tr>
<tr>
<td>Routing protocol</td>
<td>RIP</td>
<td>Typically UDP</td>
</tr>
<tr>
<td>Name translation</td>
<td>DNS</td>
<td>Typically UDP</td>
</tr>
</tbody>
</table>
How to create a segment

Sending application:
- specifies IP address and destination port
- uses socket bound to a source port

Transport Layer:
- breaks/combines application data into chunks
- adds transport-layer header to each

Network Layer:
- adds network-layer header (with IP address)
Multiplexing at Sender

- handles data from multiple sockets
- adds transport header (later used for demultiplexing)
Demultiplexing at Receiver

- use header information to deliver received segments to correct socket
User Datagram Protocol (UDP)

- no frills, bare bones transport protocol
- **best effort** service, UDP segments may be:
  - lost
  - delivered out-of-order, duplicated to app
- **connectionless:**
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others
- reliable transfer still possible:
  - **add reliability at application layer**
  - application-specific error recovery!

*I was gonna tell you guys a joke about UDP…
But you might not get it*
Host receives 2 UDP segments:
- checks **dst port**, directs segment to socket w/that port
- **different src IP or port** but **same dst port** → **same socket**
- application must sort it out
UDP Segment Format

length (in bytes) of UDP segment, including header

UDP header size: 8 bytes

(application message (payload))

(IP address will be added when the segment is turned into a datagram at the Network Layer)
UDP Advantages & Disadvantages

**Speed:**
- no connection establishment (which can add delay)
- no congestion control: UDP can blast away as fast as desired

**Simplicity:**
- no connection state at sender, receiver
- small header size (8 bytes)

*(Possibly) Extra work for applications:*
Need to handle reordering, duplicate suppression, missing packets

*Not all applications will care about these!*
Who uses UDP?

**Target Users:** streaming multimedia apps

- loss tolerant *(occasional packet drop OK)*
- rate sensitive *(want constant, fast speeds)*
Transmission Control Protocol (TCP)

- Reliable, ordered communication
- Standard, adaptive protocol that delivers good-enough performance and deals well with congestion
- All web traffic travels over TCP/IP

Why? Enough applications demand reliable ordered delivery that they should not have to implement their own protocol

But... not really end-to-end (just socket-to-socket)
TCP Segment Format

HL: header len
U: urgent data
A: ACK # valid
P: push data now
RST, SYN, FIN:
connection commands
(setup, teardown)
# bytes receiver willing to accept

TCP header size: 20-60 bytes (usually 20)

(IP address will be added when the segment is turned into a datagram at the Network Layer)
TCP Segments

Each segment carries a unique sequence #
• The initial number is chosen randomly
• The SEQ is incremented by the data length
  
  4410 simplification: assume all payloads of size 1

Each segment carries an acknowledgment
• Acknowledge a set of packets by ACK-ing the latest SEQ received
• The acknowledgment is the sequence number of the next expected byte

Reliable transport is implemented using these identifiers
TCP Connection

TCP is *connection*-oriented

TCP connection identified by

- source IP address
- source port number
- dest IP address
- dest port number
TCP Connection Setup

- A connection is initiated with a three-way handshake
- Three-way handshake ensures against duplicate SYN packets
- Takes 3 packets, 1.5 RTT (Round Trip Time)

**SYN** = Synchronize
**ACK** = Acknowledgment
TCP Handshakes

3-way handshake establishes common state on both sides of a connection.
Both sides will:
- have seen one packet from the other side ➔ know connection identification and seq numbers
- know that the other side is ready to receive

Server will typically create a new socket for the client upon connection.
Typical handshake to web server
(not showing IP addresses)

1. Browser \( \rightarrow \) Server:
   Send SYN(src_port=1234, dst_port=80, seq=31415)

2. Server \( \rightarrow \) Browser:
   Send SYN-ACK(src_port=2345, dst_port=1234, seq=27182, ack=31416)

3. Browser \( \rightarrow \) Server:
   Send ACK(src_port=1234, dst_port=2345, seq=31416, ack=27183)

now both sides now connection identification and initial sequence numbers
Example TCP Usage Pattern

3 round-trips:
1. set up a connection
2. send data & receive a response
3. tear down connection

FINs tear down connections
Need to wait after a FIN for straggling packets
Sender-side: TCP keeps a copy of all sent, but unacknowledged segments
If acknowledgment does not arrive within a “send timeout” period, segments are resent
Send timeout adjusts to the round-trip delay

Here's a joke about TCP. Did you get it? Did you get it? Did you get it? Did you get it?
TCP timeouts

What is a good timeout period?
- Goal: improve throughput without unnecessary transmissions

\[
\text{NewAverageRTT} = (1 - \alpha) \cdot \text{OldAverageRTT} + \alpha \cdot \text{LatestRTT}
\]

\[
\text{NewAverageVar} = (1 - \beta) \cdot \text{OldAverageVar} + \beta \cdot \text{LatestVar}
\]

where \( \text{LatestRTT} = (\text{ack\_receive\_time} - \text{send\_time}) \),
\( \text{LatestVar} = |\text{LatestRTT} - \text{AverageRTT}| \),
\( \alpha = 1/8 \), \( \beta = 1/4 \) typically.

\[
\text{Timeout} = \text{AverageRTT} + 4 \cdot \text{AverageVar}
\]

→ Timeout is a function of RTT and variance
Aside: Bandwidth vs Latency of a network link

- Bandwidth: #bytes per second
- (one-way) Latency: delay in seconds
- Round Trip Time (RTT): 2 x Latency

- Traffic analogy:
  - Bandwidth: #lanes in the road
  - Latency: length of the road

- Capacity: bandwidth x latency
  - in bytes
How long does it take to send a segment?

- S: size of segment in bytes
- L: one-way latency in seconds
- B: bandwidth in bytes per second

Then the time between the start of sending and the completion of receiving is $L + \frac{S}{B}$ seconds (ignoring headers)

- And another $L$ seconds (total: $2L + \frac{S}{B}$) before the acknowledgment is received by the sender
  - assuming ack segments are small
- The resulting end-to-end throughput (without pipelining) would be about $\frac{S}{2L + \frac{S}{B}}$ bytes/second
  → throughput goes to zero as $L$ grows to infinity
Pipelined Protocols

Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

- increases throughput

1. How big should the window be?
2. What if a packet in the middle goes missing?
Example: TCP Window Size = 4

When first item in window is acknowledged, sender can send the 5th item.
How much data “fits” in a pipe?

Suppose:
- b/w is $b$ bytes / second
- RTT is $r$ seconds
- ACK is a small message

→ you can send $b \times r$ bytes before receiving an ACK for the first byte

but b/w and RTT are both variable…
TCP Congestion Control

Additive-Increase/Multiplicative-Decrease (AIMD):
• window size++ every RTT if no packets dropped
• window size/2 if packet is dropped
  – drop evident from the acknowledgments

→ slowly builds up to max bandwidth, and hover there
  – Does not achieve the max possible
  + Shares bandwidth well with other TCP connections

This linear-increase, exponential backoff in the face of congestion is termed \textit{TCP-friendliness}
TCP Window Size

- Linear increase
- Exponential backoff

(Assuming no other losses in the network except those due to bandwidth)

Window Sizes:
1,2,3,4,5,6,7,8,9,10,
5,6,7,8,9,10,
5,6,7,8,9,10,
...
TCP Fairness

**Fairness goal:** if $k$ TCP sessions share same bottleneck link of bandwidth $R$, each should have average rate of $R/k$
Why is TCP fair?

Two competing sessions:
- additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally

Diagram illustrating equal bandwidth share and congestion avoidance:
- Congestion avoidance: additive increase
- Loss: decrease window by factor of 2

Diagram shows two connections:
- Connection 1 throughput
- Connection 2 throughput

Legend:
- $R_0$
TCP Slow Start

Problem:
• linear increase takes a long time to build up a window size that matches the link bandwidth*delay
• most file transactions are short
→ TCP spends a lot of time with small windows, never reaching large window size

Solution: Allow TCP to increase window size by doubling until first loss

Initial rate is slow but ramps up exponentially fast
TCP Slow Start

- Initial phase: **exponential increase**
- Assuming no other losses in the network except those due to bandwidth
TCP Fast Retransmit

- Receiver detects a lost packet (i.e., a missing seq), ACKs the last id it successfully received.
- Sender can detect the loss without waiting for timeout (uses 3rd duplicate ack).
TCP Summary

- Reliable ordered message delivery
  - Connection oriented, 3-way handshake
- Transmission window for better throughput
  - Timeouts based on link parameters
- Congestion control
  - Linear increase, exponential backoff
- Fast adaptation
  - Exponential increase in the initial phase
Application Layer

Several figures in this section come from “Computer Networking: A Top Down Approach” by Jim Kurose, Keith Ross
Naming

People
• SSN, NetID, Passport #

Internet Hosts, Routers
1. IP address (32 bit), 151.101.117.67
   • For now, 32-bit descriptor, like a phone number
   • Longer addresses in the works…
   • Assigned to hosts by their internet service providers
   • Not physical: does not identify a single node, can swap machines and reuse the same IP address
   • Not entirely virtual: determines how packets get to you, changes when you change your ISP
2. Virtual: “name” www.cnn.com
   • Used by humans (no one wants to remember a bunch of #s)

How to convert hostname to IP address?
Domain Name System (DNS)

**Root DNS Servers**

- .com DNS servers
- .org DNS servers
- .edu DNS servers

**Distributed, Hierarchical Database of Domains**

- Application-Layer Protocol: hosts & name servers communicate to resolve names
- Names are separated by dots into components
  
  *Not to be confused with dots in IP addresses (in which the order of least significant to most significant is reversed)*
- Components resolved from right to left
- All siblings in a domain must have unique names
DNS: root name servers

Contacted by local name server that cannot resolve top-level name
• owned by Internet Corporation for Assigned Names & Numbers (ICANN)
• returns mapping to local name server

13 root name “servers” worldwide

a. Verisign, Los Angeles CA (5 other sites)
b. USC-ISI Marina del Rey, CA
c. Cogent, Herndon, VA (5 other sites)
d. U Maryland College Park, MD
h. ARL Aberdeen, MD
j. Verisign, Dulles VA (69 other sites)
i. Netnod, Stockholm (37 other sites)
k. RIPE London (17 other sites)
l. ICANN Los Angeles, CA (41 other sites)
g. US DoD Columbus, OH (5 other sites)
m. WIDE Tokyo (5 other sites)
DNS Lookup

1. the client asks its local nameserver
2. the local nameserver asks one of the root nameservers
3. the root nameserver replies with the address of the authoritative nameserver
4. the server then queries that nameserver
5. repeat until host is reached, cache result.

Example: Client wants IP addr of www.amazon.com

1. Queries root server to find com DNS server
2. Queries .com DNS server to get amazon.com DNS server
3. Queries amazon.com DNS server to get IP address for www.amazon.com
DNS Services

Simple, hierarchical namespace works well
- Can name anything
- Can alias hosts
- Can cache results
- Can share names (replicate web servers by having 1 name corresponding to several IP addresses)
- Can exploit proximity of clients and servers

Q: Why not centralize?
- Single point of failure
- Traffic volume
- Distant Centralized Database
- Maintenance
A: Does not scale!
What about security? (don’t ask!)
Application Layer

- Network-aware applications
- Clients & Servers
- Peer-to-Peer
Sockets

“Door” between application process and end-end-transport protocol

Sending process:

- shoves message out door
- relies on transport infrastructure on other side of door to deliver message to socket at receiving process

![Diagram of sockets and internet](image)
Socket programming

Two socket types for two transport services:
  • UDP: unreliable datagram
  • TCP: reliable, byte stream-oriented

Host could be running many network applications at once. Distinguish them by binding the socket to a **port number**:
  • 16 bit unsigned number
  • 0-1023 are well-known
    (web server = 80, mail = 25, telnet = 23)
Client/server socket interaction: UDP

**Server** (running on serverIP)  
create `serversocket`, bind to port x  
create message  
send message to (serverIP, port x) via clientsocket

**Client**
create clientsocket  
create message  
send message to (serverIP, port x) via clientsocket

read data (and clientAddr) from serversocket

send response data to clientAddr via serversocket

receive message (and serverAddr) from clientsocket

close clientsocket
import socket    # include Python’s socket library
serverName = 'servername'
serverPort = 12000

#create UPD socket
clientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

#get user input
message = input('Input lowercase sentence: ')

# send with server name + port
clientSocket.sendto(message.encode(), (serverName, serverPort))

# get reply from socket and print it
reply, serverAddress = clientSocket.recvfrom(2048)
print(reply.decode())

clientSocket.close()
import socket  # include Python’s socket library
serverPort = 12000

#create UDP socket & bind to local port 12000
serverSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print("The server is ready to receive")

while True:
    # Read from serverSocket into message,
    # getting client’s address (client IP and port)
    message, clientAddress = serverSocket.recvfrom(2048)
    print("received message: " + message.decode())
    modifiedMsg = message.decode().upper()
    print("sending back to client")

    # send uppercase string back to client
    serverSocket.sendto(modifiedMsg.encode(), clientAddress)
Socket programming w/ TCP

Client must contact server

**Server:**
- already running
- server already created
  “welcoming socket”

**Client:**
- Creates TCP socket w/ IP address, port # of server
- Client TCP establishes connection to server TCP

- when contacted by client, server TCP creates new socket to communicate with that particular client
  - allows server to talk with multiple clients
  - source port #s used to distinguish clients

**Application viewpoint:** TCP provides reliable, in-order byte-stream transfer between client & server
Client/server socket interaction: TCP

**Server** (running on hostID)
- create welcoming *serversocket*
- bind to *port x*
- in response to connection request, create *connectionsocket*
- read data from *connectionsocket*
- send response data to *clientAddr* via *connectionsocket*
- close *connectionsocket*

**Client**
- create *clientsocket*
- connect to (*hostID*, *port x*)
- create message
- send message via *clientsocket*
- receive message from *clientsocket*
- close *clientsocket*
import socket  # include Python's socket library
serverName = 'servername'
serverPort = 12000

#create TCP socket for server on port 12000
clientSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clientSocket.connect((serverName, serverPort))

# get user input
message = input('Input lowercase sentence: ')

# send (no need for server name + port)
clientSocket.send(message.encode())

# get reply from socket and print it
reply, serverAddress = clientSocket.recvfrom(1024)
print(reply.decode())

clientSocket.close()
Python TCP Server

import socket  # include Python’s socket library
serverPort = 12000

# create TCP welcoming socket & bind to server port 12000
serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serverSocket.bind(('', serverPort))
# server begins listening for incoming TCP requests
serverSocket.listen(1)
print("The server is ready to receive")

while True:
    # server waits on accept() for incoming requests
    # new socket created on return
    connectionSocket, addr = serverSocket.accept()
    message = connectionSocket.recv(1024).decode()
    print("received message: "+message)
    modifiedMsg = message.upper()

    # send uppercase string back to client
    connectionSocket.send(modifiedMsg.encode())

    # close connection to this client, but not welcoming socket
    connectionSocket.close()