TCP Overview

Jeff Chase
Duke University

These slides draw extensively on material from Srini Seshan and Dave Andersen at CMU (mostly figures), and they also incorporate earlier material by Adolfo Rodriguez and Amin Vahdat. Some congestion control slides are from Ion Stoica.

The Internet Protocol Suite

- Application
- Presentation
- Session
- Transport
- Network
- Data link
- Physical

The waist facilitates Interoperability.

The Hourglass Model

UDP

- User Datagram Protocol (UDP)
  - Thin veneer on top of IP
  - Data sent as individual datagrams
  - From a sender to a receiver: like USPS
  - Demultiplex receiver: (IPaddr, port) pair
  - No guarantees about reliability, in-order delivery
  - Checksum to prevent corruption of data

TCP

- Transmission Control Protocol (TCP)
  - Reliable in-order delivery of byte stream
  - Full-duplex: each endpoint may send and receive
  - Flow control
    - To ensure that sender does not overrun receiver by sending too fast
  - Congestion control
    - Keep the sender from overrunning the network
    - Many simultaneous connections across routers (cross traffic)

A Brief Internet History

1969 ARPANET created
1972 TELNET developed
1975 FTP developed
1977 MAIL developed
1981 WW/HTTP developed
1983 TCP/IP developed
1985 ARPANET dissolved
1986 VAX-WEB developed
1990 Multi-backbone Internet
1995 Mosaic

TCP/IP implementation

Some TCP Challenges

- Segment byte stream into individual packets
  - How big should the packets/segments be?
- What if packets are delivered out of order?
  - May take different paths through the network
- What if a packet is lost?
  - Packets may be dropped in the network
- What if a packet is corrupted in transit?
  - Detect error and fix it or resend
- How fast should the sender send?

Mechanism: Checksums

- Checksum \( C = F(\text{contents}) \)
- Checksum \( C \) is small, fixed-size (in essence, a hash)
- Generate at sender and place in segment
- Verify at receiver
- If checksum matches, packet is not corrupt
  - Probably...

Sequence Number Space

- Each byte in byte stream is numbered:
  - 32 bit value
  - Wraps around
  - Initial values selected at startup time
- Each packet/segment has a sequence number and length
  - Indicates where it fits in the byte stream

<table>
<thead>
<tr>
<th>Value</th>
<th>Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>13450</td>
<td>8</td>
</tr>
<tr>
<td>14950</td>
<td>9</td>
</tr>
<tr>
<td>16050</td>
<td>10</td>
</tr>
<tr>
<td>17550</td>
<td></td>
</tr>
</tbody>
</table>

Sequence Numbers

- 32 Bits, Unsigned
  - Circular Comparison

<table>
<thead>
<tr>
<th>Value</th>
<th>Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b &lt; a</td>
<td>a &lt; b</td>
</tr>
</tbody>
</table>

Why So Big?

- Guard against stray packets
  - With IP, packets have maximum lifetime of 120s
  - Sequence number would wrap around in this time at 286MB/s

Using the Sequence Numbers

- Reassembly buffer
  - Packets/segments received into (kernel) memory
  - Sort them by sequence number
  - Deliver segments to application in order!
  - \( \text{Seq}(i) + \text{Len}(i) < \text{Seq}(i+1) \)?
    - Gap: defer delivery of segment \( i+1 \)
- Acknowledgments
  - Periodically send back the sequence number of the latest (newest) byte received in order.
  - No ack received? Lost segment: retransmit.
  - How long to wait?

TCP Header Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>SrcPort</td>
<td>16</td>
</tr>
<tr>
<td>DestPort</td>
<td>16</td>
</tr>
<tr>
<td>SequenceNum</td>
<td>32</td>
</tr>
<tr>
<td>Acknowledgment</td>
<td>32</td>
</tr>
<tr>
<td>HdrLen</td>
<td>4</td>
</tr>
<tr>
<td>Flags</td>
<td>1</td>
</tr>
<tr>
<td>AdvertisedWindow</td>
<td>16</td>
</tr>
<tr>
<td>CheckSum</td>
<td>16</td>
</tr>
<tr>
<td>UrgPtr</td>
<td>16</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>

- Without options, TCP header 20 bytes
  - Thus, typical Internet packet minimum of 40 bytes
Establishing Connection: Three-Way handshake

- Each side notifies other of starting sequence number it will use for sending
  - Why not simply choose 0?
  - Must avoid overlap with earlier incarnation
  - Security issues
- Each side acknowledges other's sequence number
  - SYN-ACK: Acknowledge sequence number + 1
- Can combine second SYN with first ACK

TCP State Diagram: Connection Setup

Tearing Down Connection

- Either side can initiate tear down
  - Send FIN signal
  - "I'm not going to send any more data"
- Other side can continue sending data
  - Half-open connection
  - Must continue to acknowledge
- Acknowledging FIN
  - Acknowledge last sequence number + 1

Reliable Transmission

- How do we send a packet reliably when it can be lost?
- Two mechanisms
  - Acks
  - Timeouts
- Simplest reliable protocol: Stop and Wait

Stop and Wait

Send a packet, stop and wait until ack arrives

Recovering From Error
Problems with Stop and Wait

- How to recognize a duplicate transmission?
  - Solution: put sequence number in packet

- Performance
  - Unless Latency-Bandwidth product is very small, sender cannot fill the pipe
  - Solution: sliding window protocols

Keeping the Pipe Full

- Bandwidth-Delay product measures network capacity
- How much data can you put into the network before the first byte reaches receiver
- Stop and Wait: 1 data packet per RTT
  - Ex. 1.5-Mbps link with 45-ms RTT
  - Stop-and-wait: 182 Kbps
- Ideally, send enough packets to fill the pipe before requiring first ACK

How Do We Keep the Pipe Full?

- Send multiple packets without waiting for first to be ACKed
  - How many? Limited by the "window" wnd.
  - Flow/congestion policies set wnd.
- Self-clocking sliding window
  - Arrival of an ack opens up another window "slot" to send
  - Ideally, first ACK arrives immediately after window is filled
  - Else pipeline "bubbles" waste bandwidth
- Throughput = wnd/RTT

Flow Control

- Receiver devotes some buffer space to hold incoming bytes until the application consumes them
  - Socket buffers
- How much? Must place a bound on it.
  - Advertise wnd: max number of bytes to accept
  - Receiver returns advertisedWindow in TCP header of its acknowledgments back to the sender.
- Sliding window
  - Flow window is range of bytes receiver will accept
  - [ack+1, ack + wnd]
  - Receiver drops segments/bytes outside the window
  - Sender stops transmitting when it fills the window
  - Bytes in transit <= wnd
  - Each side advances window as data is delivered.

Window Flow Control: Send Side

*Packet Sent*
- Source Port
- Dest. Port
- Sequence Number
- Acknowledgement
- HL/Flags
- Window
- D. Checksum
- Urgent Pointer
- Options

*Packet Received*
- Source Port
- Dest. Port
- Sequence Number
- Acknowledgement
- HL/Flags
- Window
- D. Checksum
- Urgent Pointer
- Options

App write

acknowledged

sent

to be sent

outside window
Window Flow Control: Receive Side

What should receiver do with an arriving segment?

- Aced but not delivered to user
- Not yet asked
- New
- Receive buffer
- Duplicate out-of-order outside window
- Outside window
- Window

TCP Persist

- What happens if window is 0?
  - Application has not consumed data fast enough.
  - Receiver buffer exhausted: grind to halt
- Must reopen window when application reads data
  - App reads data: opens up buffer space
  - Receiver sends segment with “window update”
  - What if this update is lost?
- TCP Persist state (sender idled by closed window)
  - Sender periodically sends 1 byte packets
  - Receiver responds with ACK even if it can't store the packet
  - ACK segment includes current window

Performance

Limits to Throughput

- How fast can the app produce or consume data?
- Hardware/path limitations
  - Wire speed
  - Host limitations/overhead
- Efficient use of the wire
  - Leaving network idle
  - Sending duplicate data
  - High ratio of control to data
  - What causes loss of efficiency?

Efficiency: Flow Window Size

- OS system calls to allow receiving application to set the socket buffer size.
- Defaults are often too small for long/fat networks
- Leaves network bandwidth idle
- The window size field in the TCP header limits the window that the receiver can advertise.
  - 16 bits → 64 KBytes
  - 10 msec RTT → 51 Mbit/second
  - 100 msec RTT → 5 Mbit/second
- Solution: TCP options added to get around 64KB limit
  - Window scaling (RFC 1323)
  - Shift advertised window field by specified number of bits
Efficiency: RTT Estimation

- Retransmission timer (RTO)
  - Underestimate RTT → unnecessary retransmits
  - Overestimate RTT → network idles after drops
- Solution: sender samples RTT by measuring time between transmit of segment and received ack.
  - TCP now has an option to make this easier
    - Receiver reflects timestamp placed by sender
- But samples will vary: how to get a stable estimate?
  - Key technique: exponential smoothing
  - Exponentially weighted moving average
  - But it’s tricky...

TCP RTT Estimator

- Exponential smoothing:
  - Identify persistent behaviors/trends, but do not over-react to transient changes.
  - \[ \text{RTT} = \alpha \times (\text{old RTT}) + (1 - \alpha) \times (\text{new sample}) \]
  - Recommended value for \( \alpha \): 0.8 - 0.9
    - 0.875 for most TCPs

Note: it is also tricky to convert the RTT estimate into a good value for the RTO:
- Track RTT variance.
- “Loosen” the timer when the RTT variance is high.

Efficiency: Tactical Delays

- Want to send full-size segments, but what if app writes just a small amount of data?
  - E.g., a user typing at a remote shell.
  - Wait for more data? How long?
    - Nagle heuristic: allow at most one outstanding unacked short segment.
- Want to advertise reopenings in the flow window, but what if app reads just a small amount of data?
  - Silly window syndrome (Clark 1982)
  - Wait for more reads? How long?
    - Heuristic: increase window by \( \min(MSS, \text{ReceiveBuffer}/2) \)
- Want to ack data in a timely fashion, but also take advantage of cumulative acks to reduce ack traffic.
  - Heuristic: after receiving an in-order segment, wait up to 500 ms for another one before sending an ack.

Overhead

Sources of CPU Overhead

Although TCP/IP family protocol processing itself is reasonably efficient, managing a dumb NIC steals CPU/memory cycles away from the application.

\( a = \) application processing per unit of bandwidth
\( o = \) host communication overhead per unit of bandwidth
The host/network gap

If overhead is high, then the host CPU will "saturate" below wire speed.

What matters: \( a+o \).

Outrunning Moore’s Law?

High-speed Small-Area Networks (SANs) and Ethernet are both advancing ahead of Moore’s Law, e.g., roughly one order of magnitude every 4 years.

Network Bandwidth per CPU cycle

Host overhead (a)

Throughput improves as hosts advance, but bandwidth per cycle is constant once the host saturation point is reached.

"IP SANs"

- If you believe in the problem, then the solution is to attach hosts to the faster wires with smarter NICs.
  - Hardware checksums, interrupt suppression
  - Transport offload (TOE)
  - Connection-aware w/ early demultiplexing
  - ULP offload (e.g., iSCSI)
  - Direct data placement/RDMA ("remote DMA")
- Since these NICs take on the key characteristics of SANs, let’s use the generic term "IP-SAN".
  - S stands for Small, Server, Storage, System, ...
  - Non-IP SANs: Giganet, FibreChannel, Infiniband...

Hitting the wall

How much can IP-SANs help?

- IP-SAN is a difficult engineering challenge.
  - It takes time and money to get it right.
- LAWS (Shiva&Chase03) is a "back of napkin" analysis to explore potential benefits and limitations.
- Figure of merit: marginal improvement in peak application throughput ("speedup")
- Premise: Internet servers are fully pipelined
  - Ignore latency (your mileage may vary)
  - IP-SANs can improve throughput if host saturates.

Application ratio (\( \gamma \))

Application ratio (\( \gamma \)) captures "compute-intensity".

For a given application, lower overhead increases \( \gamma \).

For a given communication system, \( \gamma \) is a property of the application: it captures processing per unit of bandwidth.


**What to Know**

- **Overhead** often matters for network performance, sometimes more than latency and bandwidth.
  - That goes for other I/O as well.
  - Overhead matters if but only if the CPU saturates.
- Reducing overhead might improve performance... or not.
  - Your mileage may vary: throughput < 1/(a+o)
  - Be skeptical of claims.
- **Amdahl's Law** (diminishing returns):
  - How much speedup can I get by optimizing one part of the system? (e.g., eliminating o)
  - Depends on its share of the total cost.
  - Throughput < 1/a → speedup bounded by o/a

**Congestion**

- Different sources compete for resources inside network
- Why is it a problem?
  - Sources are unaware of current state of resource
  - Sources are unaware of each other
- Manifestations:
  - Lost packets (buffer overflow at routers)
  - Long delays (queuing in router buffers)
  - Can result in throughput less than bottleneck link (1.5Mbps for the above topology) → a.k.a. congestion collapse

**Causes & Costs of Congestion**

- Four senders - multihop paths
- Timeout/retransmit

Q: What happens as rate increases?

- When packet dropped, any "upstream transmission capacity used for that packet was wasted!"
**Congestion Collapse**

- Definition: Increase in network load results in decrease of useful work done
- Many possible causes
  - Spurious retransmissions of packets still in flight
  - Classical congestion collapse
  - Solution: better timers and TCP congestion control
  - Undelivered packets
    - Packets consume resources and are dropped elsewhere in network
    - Solution: congestion control for ALL traffic

**Congestion Control and Avoidance**

- A mechanism which:
  - Uses network resources efficiently
  - Preserves fair network resource allocation
  - Prevents or avoids collapse
- Congestion collapse is not just a theory
  - Has been frequently observed in many networks

**What's Really Happening?**

- Knee - point after which
  - Throughput increases very slow
  - Delay increases fast
- Cliff - point after which
  - Throughput starts to decrease very fast to zero (congestion collapse)
  - Delay approaches infinity
- Note (in an M/M/1 queue)
  - Delay = 1/(1 - utilization)

**Congestion Control vs. Congestion Avoidance**

- Congestion control goal
  - Stay left of cliff
- Congestion avoidance goal
  - Stay left of knee

**TCP Congestion Control**

- Sender maintains congestion window cwnd
  - Max number of packets in flight is limited by both flow window and congestion window
  - MIN(wnd, cwnd)
- Sender probes the network by sending faster and faster, until it encounters congestion.
  - Grow the cwnd as acks come back
- Congestion? Throttle back.
- Driven by senders: distributed, fair and efficient...
  - If we can get the policies right....
  - And if everybody plays nice.

**AIMD and the “TCP Sawtooth”**

- Additive increase multiplicative decrease
- TCP periodically probes for available bandwidth by increasing its window (rate) additively by one segment per window (per RTT).
- Congestion?
  - Cut window in half →cut sending rate in half
  - multiplicative rate decrease
- AIMD turns out to be stable, and it can be shown that most alternatives are either too indolent or too aggressive, e.g., waste bandwidth or can still drive the network into congestion collapse.
Detecting Congestion

- Very simple mechanisms in IP routers
  - congestion → loss
  - Many proposals for routers to mark packets to warn of congestion (ECN, XCP), but not yet widely deployed.
- So: TCP interprets packet drops as a congestion indicator.
  - loss → congestion
  - Not necessarily true...but a good heuristic.
  - Loss can also result from transient errors, e.g., on wireless
- Duplicate acknowledgments are also a good early indication that a packet was dropped...
  - ...or maybe the packets were just reordered.
  - Heuristic: "one duplicate ack could be a reorder, but if it happens again then it must be congestion".
  - triple dup ack

Detecting Congestion: Summary of Approaches

- End-end congestion control:
  - No explicit feedback from network
  - Congestion inferred from end-system observed loss, delay
  - Approach taken by TCP
- Network-assisted congestion control:
  - Routers provide feedback to end systems
  - Single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - Explicit send rate
  - Problem: makes routers complicated

Under the Hood (RFC 2581)

- Ack each segment with highest seqnum received.
- Acks drive actions at sender: self-clocking.
- Below ssthresh, cwnd = cwnd+1 for each acked segment.
  - "Slow start" is really fast, e.g., doubles cwnd per RTT.
- Above ssthresh, cwnd = cwnd + (1/cwnd) on each acked segment.
  - Additive increase, e.g., grows cwnd by one per RTT.
  - "congestion avoidance"
- Loss? Congestion!
  - Multiplicative decrease: ssthresh = cwnd/2; cwnd = 1
  - "congestion control": back to "slow start"
  - Triple dup ack? Load "Fast retransmit". (Tahoe)
  - After fast-retransmit loss: cwnd = ssthresh, every ack adds 1/cwnd to window, even dupes.
  - Multiplicative decrease with "fast recovery" (Reno)

These details are not to be tested for CPS 196.

The big picture: Basic AIMD

TCP Saw Tooth Behavior

- Retransmit after 3 duplicated acks
  - prevent expensive timeouts
- No need to slow start again
- At steady state, cwnd oscillates around the optimal window size.
Sending Too Fast

- Overflow at receiver? Receiver drops packet.
- Overflow network link? NIC drops packet.
- Overflow router? Router drops packet.
- Faster than fair share?
  - **Pro**: you win
  - **Con**: somebody else loses
  - TCP is a game

Max-Min Fairness Criteria

- "Fair sharing"
- But flows have differing demands...
- Flows demanding less than their share get as much as they need.
- Flows demanding more than their share split the surplus.
- Generalizes to proportional sharing

Trust and Rate Control

- Is gaming TCP a security problem?
- How should the network deal with this?
- Whose responsibility is it?
- What incentive does anyone have to play the game by the rules?
  - Good Samaritan?
  - Rodney King: “Can’t we all just get along?”
  - Judge Judy?
  - Adam Smith?

Savage TCP (Daytona)

- Attack: “Ack early, ack often”.
  - Three variations on a theme.
  - “Ack early hides congestion loss.”
  - “Big ack attack”
- Defense:
  - Don’t make hidden assumptions about a peer’s good behavior.
  - One ack per segment? Uh uh.
  - Remove incentives to cheat.
  - Trust but verify.
  - Nonces and cumulative nonces.

Summary: a malicious TCP receiver can fool an honest sender into sending faster than the network allows, consuming an unfair share of network bandwidth.


The Role of Routers?

- **Flow**: a stream of related packets or demands, e.g., between a given source and destination endpoint.
- **TCP-friendly flow**: arrival rate does not exceed a compliant TCP with a given RTT and drop rate.
- **Unresponsive flow**: not TCP-friendly.
  - Does not throttle back (enough) when packets are dropped.
  - Can a router identify unresponsive flows? How much time/state does the router need? (hard problem)
- **Principle**: congested routers should preferentially drop packets of unresponsive flows to punish the guilty and create an incentive for cooperation.


TCP Link Sharing Behavior

- Even if all flows are faithful to the Congestion Game, there are other behaviors/anomalies to consider:
  - Is it fair? What if some flows through a congested link have higher RTTs than others?
  - What if sender and receiver use multiple TCP streams to communicate?
  - Browsers, gridFTP
  - What if congestion patterns are dynamic at time scales smaller than an RTT?
  - How effectively can one TCP flow use the bandwidth? What if the network is long/fat?
  - Is more buffering at the routers always better?
  - Does it work for small/bursty connections?
Suitability of TCP

- Rate control: required to be a good citizen?
  - “Stick with TCP” or “stuck with TCP”?
- In order delivery always good?
- Loss tolerance vs. jitter?
  - Sometimes jitter is worse than loss.
- Alternative transports in IETF: SCTP, DCCP
- Security? TLS/SSL (coming soon…)
- Does content distribution need different protocols?
  - Multicast
  - Different sending/receiving rates
  - Forward Error Correction

TCP Timeline

- 1975: Three-way handshake
  - Increased bandwidth is slow to adapt
  - TCP described by Vint Cerf and Bob Kahn

- 1974: TCP
  - Ray Tomlinson

- 1974: Proto-OSI
  - TOCB TCP

- 1975: Three-way handshake
  - Increased bandwidth is slow to adapt

- 1974: TCP
  - Ray Tomlinson

- 1982: TCP & IP
  - RFC 793 & 791

- 1984: Nagel’s algorithm
  - To reduce overhead
  - Predict congestion collapse
  - TCP/IP

- 1984: TCP & IP
  - Nagel’s algorithm

- 1985: Karn’s algorithm
  - To better estimate round-trip time

- 1988: Van Jacobson’s congestion avoidance
  - Reno congestion control
  - Tahoe congestion control
  - 4.3BSD Tahoe

- 1990: 4.3BSD Reno
  - Fast recovery
  - Delayed ACKs

TCP: After 1990

- 1993: ECN
  - Explicit Congestion Notification
  - TCP Vegas
  - Source-based congestion avoidance

- 1994: TCP Vegas
  - Source-based congestion avoidance

- 1996: SACK TCP
  - Selective Acknowledgement

TCP Flavors

- TCP Tahoe
  - Jacobson’s implementation of congestion control
  - No fast recovery
- TCP Reno
  - Fast recovery
  - Delayed ACKs
- TCP Vegas
  - Source-based congestion avoidance rather than control
  - TCP Reno needs to cause congestion to determine available bandwidth

Extra slides: may be of interest, but will not be tested.
**TCP Connection Teardown**

- Each side of a TCP connection can independently close the connection.
  - Thus, possible to have a half duplex connection
- Closing process sends a FIN message
  - Waits for ACK of FIN to come back
  - This side of the connection is now closed

**Jacobson's Retransmission Timeout**

- Where to set RTO? Originally 2*RTT.
  - Timeout? Cut RTO in half
  - Lots of spurious timeouts
- At high loads RTT variance is high
- Solution:
  - Base RTO on RTT and standard deviation
    - RTO = RTT + 4 * rttvar
  - new_rttvar = β * dev + (1- β) old_rttvar
  - Dev = smoothed linear deviation

**TCP ACK Generation** [RFC 1122, RFC 2581]

<table>
<thead>
<tr>
<th>Event</th>
<th>TCP Receiver action</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-order segment arrival, No gaps,</td>
<td>Delayed ACK. Wait up to 500ms for next segment. If no</td>
</tr>
<tr>
<td>Everything else already ACKed</td>
<td>next segment, send ACK</td>
</tr>
<tr>
<td>In-order segment arrival, No gaps,</td>
<td>Immediately send single cumulative ACK</td>
</tr>
<tr>
<td>One delayed ACK pending</td>
<td></td>
</tr>
<tr>
<td>Out-of-order segment arrival Higher-than-expect seq.</td>
<td>Send duplicate ACK, indicating seq. # of next expected</td>
</tr>
<tr>
<td># Gap detected</td>
<td>byte</td>
</tr>
<tr>
<td>Arrival of segment that partially or completely</td>
<td>Immediate ACK</td>
</tr>
<tr>
<td>fits gap</td>
<td></td>
</tr>
</tbody>
</table>