Transport Protocol Design: UDP, TCP

Shivkumar Kalyanaraman
Rensselaer Polytechnic Institute
shivkuma@ecse.rpi.edu
http://www.ecse.rpi.edu/Homepages/shivkuma

Based in part upon slides of Prof. Raj Jain (OSU), Srini Seshan (CMU), J. Kurose (U Mass), I.Stoica (UCB)
Overview

- UDP: connectionless, end-to-end service
- UDP Servers
- TCP features, Header format
- Connection Establishment
- Connection Termination
- TCP Server Design
- Ref: Chap 11, 17, 18; RFC 793, 1323
Transport Protocols

- Protocol implemented entirely at the ends
  - Fate-sharing
  - Completeness/correctness of function implementations

- UDP provides just integrity and demux
- TCP adds...
  - Connection-oriented
  - Reliable
  - Ordered
  - Point-to-point
  - Byte-stream
  - Full duplex
  - Flow and congestion controlled
UDP: User Datagram Protocol [RFC 768]

- **Minimal** Transport Service:
  - “Best effort” service, UDP segments may be:
    - Lost
    - Delivered out of order to app

- **Connectionless**:
  - No handshaking between UDP sender, receiver
  - Each UDP segment handled independently of others

Why is there a UDP?

- *No connection* establishment (which can add delay)
- *Simple*: no connection state at sender, receiver
- *Small header*
- *No congestion control*: UDP can blast away as fast as desired: dubious!
Multiplexing / demultiplexing

Recall: \textit{segment} - unit of data exchanged between transport layer entities

- aka TPDU: transport protocol data unit

Demultiplexing: delivering received segments to correct app layer processes

\[ \text{application-layer data} \]

\[ \text{segment header} \]

\[ H_t \rightarrow M \rightarrow H_n \text{ segment} \]

\[ \text{application} \]
\[ \text{transport} \]
\[ \text{network} \]

\[ P_1 \rightarrow P_3 \rightarrow \text{receiver} \rightarrow P_4 \rightarrow P_2 \]
Multiplexing / demultiplexing

gathering data from multiple app processes, enveloping data with header (later used for demultiplexing)

Multiplexing: gathering data from multiple app processes, enveloping data with header (later used for demultiplexing)

- based on sender, receiver port numbers, IP addresses
  - source, dest port #s in each segment
  - recall: well-known port numbers for specific applications

multiplexing/demultiplexing:

TCP/UDP segment format
UDP, cont.

- Often used for **streaming multimedia** apps
  - Loss tolerant
  - Rate sensitive
- Other UDP uses (**why**?):
  - DNS
  - SNMP
- Reliable transfer over UDP: add reliability at application layer
  - Application-specific error recover!

<table>
<thead>
<tr>
<th>Source port #</th>
<th>Dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>Checksum</td>
</tr>
<tr>
<td>Application data (message)</td>
<td></td>
</tr>
</tbody>
</table>

**UDP segment format**
**UDP Checksum**

**Goal:** detect "errors" (e.g., flipped bits) in transmitted segment.

**Note:** IP only has a *header* checksum.

**Sender:**
- Treat segment contents as sequence of 16-bit integers
- Checksum: addition (1’s complement sum) of segment contents
- Sender puts checksum value into UDP checksum field

**Receiver:**
- Compute checksum of received segment
- Check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?*
Introduction to TCP

- Communication abstraction:
  - Reliable
  - Ordered
  - Point-to-point
  - Byte-stream
  - Full duplex
  - Flow and congestion controlled
- Protocol implemented entirely at the ends
- Fate sharing
Evolution of TCP

1974
TCP described by Vint Cerf and Bob Kahn
In IEEE Trans Comm

1975
Three-way handshake
Raymond Tomlinson
In SIGCOMM 75

1975
TCP
described by
Vint Cerf and Bob Kahn
In IEEE Trans Comm

1980
TCP & IP
RFC 793 & 791

1982
TCP & IP
RFC 793 & 791

1983
BSD Unix 4.2
supports TCP/IP

1984
Nagel's algorithm
to reduce overhead
of small packets;
predicts congestion collapse

1986
Congestion collapse observed

1987
Karn's algorithm
to better estimate round-trip time

1988
Van Jacobson's algorithms
congestion avoidance and congestion control
(most implemented in 4.3BSD Tahoe)

1989
4.3BSD Reno
fast retransmit delayed ACK’s

1990
TCP Through the 1990s

- 1993: TCP Vegas (Brakmo et al) - real congestion avoidance
- 1994: T/TCP (Braden) - Transaction TCP
- 1994: ECN (Floyd) - Explicit Congestion Notification
- 1996: Hoe - Improving TCP startup
- 1996: SACK TCP (Floyd et al) - Selective Acknowledgement
- 1996: FACK TCP (Mathis et al) - extension to SACK
## TCP Header

<table>
<thead>
<tr>
<th>Flags:</th>
<th>SYN</th>
<th>FIN</th>
<th>RESET</th>
<th>PUSH</th>
<th>URG</th>
<th>ACK</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Source port</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Destination port</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acknowledgement</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Advertised window</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Urgent pointer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
Reliability Models

- Reliability => requires redundancy to recover from uncertain loss or other failure modes.

- Two types of redundancy:
  - **Spatial redundancy:** independent backup copies
    - Forward error correction (FEC) codes
    - Problem: requires huge overhead, since the FEC is also part of the packet(s) it cannot recover from erasure of all packets
  - **Temporal redundancy:** retransmit if packets lost/error
    - Lazy: trades off response time for reliability
    - Design of status reports and retransmission optimization important
Temporal Redundancy Model

- Packets
- Sequence Numbers
- CRC or Checksum
- ACKs
- NAKs
- SACKs
- Bitmaps
- Packets
- FEC information

Timeout

Status Reports

Retransmissions

Packets
Types of errors and effects

- **Forward** channel bit-errors (garbled packets)
- Forward channel **packet**-errors (lost packets)
- **Reverse** channel bit-errors (garbled status reports)
- Reverse channel bit-errors (lost status reports)

- **Protocol-induced** effects:
  - Duplicate packets
  - Duplicate status reports
  - Out-of-order packets
  - Out-of-order status reports
  - Out-of-range packets/status reports (in window-based transmissions)
Mechanisms ...

- **Mechanisms:**
  - **Checksum in pkts:** detects pkt corruption
  - **ACK:** “packet correctly received”
  - **NAK:** “packet *incorrectly* correctly received”
  - [aka: *stop-and-wait Automatic Repeat reQuest (ARQ)* protocols]

- **Provides reliable transmission over:**
  - An *error-free* forward and reverse channel
  - A *forward* channel which has *bit*-errors; reverse: ok

- **Cannot handle reverse-channel bit-errors; or packet-losses in either direction.**
More mechanisms ...

- **Mechanisms:**
  - **Checksum:** detects corruption in pkts & acks
  - **ACK:** "packet correctly received"
  - **NAK:** "packet incorrectly received"
  - **Sequence number:** identifies packet or ack
    - 1-bit sequence number used only in forward channel
      - [aka: alternating-bit protocols]

- **Provides reliable transmission over:**
  - An **error-free** channel
  - A **forward & reverse** channel with **bit**-errors
  - Detects **duplicates** of packets/acks/NAKs

- Still needs NAKs, and cannot recover from packet errors...
More Mechanisms …

- **Mechanisms:**
  - **Checksum:** detects corruption in pkts & acks
  - **ACK:** “packet correctly received”
  - **Duplicate ACK:** “packet incorrectly received”
  - **Sequence number:** identifies packet or ack
    - 1-bit sequence number used both in forward & reverse channel

- **Provides reliable transmission over:**
  - An **error-free** channel
  - A **forward & reverse** channel with bit-errors
  - Detects **duplicates** of packets/acks
  - **NAKs eliminated**

- **Packet errors in either direction not handled…**
Reliability Mechanisms...

- **Mechanisms:**
  - **Checksum:** detects corruption in pkts & acks
  - **ACK:** “packet correctly received”
  - **Duplicate ACK:** “packet incorrectly received”
  - **Sequence number:** identifies packet or ack
    - 1-bit sequence number used both in forward & reverse channel
  - **Timeout:** only at sender

- **Provides reliable transmission over:**
  - An *error-free* channel
  - A *forward & reverse* channel with *bit*-errors
  - Detects *duplicates* of packets/acks
  - *NAKs eliminated*
  - A *forward & reverse* channel with *packet*-errors (loss)
Example: *Three-Way* Handshake

- TCP connection-establishment: 3-way-handshake necessary and sufficient for unambiguous setup/teardown even under conditions of loss, duplication, and delay.
TCP Connection Setup: FSM

- **CLOSED**
  - passive OPEN
  - create TCB
  - CLOSE
  - delete TCB

- **LISTEN**
  - rcv SYN
  - snd SYN ACK
  - SEND
  - snd SYN
  - CLOSE
  - delete TCB

- **SYN RCVD**
  - rcv SYN
  - snd SYN
  - rcv ACK of SYN
  - snd ACK

- **ESTAB**
  - rcv ACK of SYN

- **SYN SENT**
  - Rcv SYN, ACK
  - Snd ACK

- **active OPEN**
  - create TCB
  - Snd SYN

- **CLOSE**
  - Send FIN
  - Snd ACK
More Connection Establishment

- **Socket**: BSD term to denote an IP address + a port number.
  - *A connection is fully specified by a socket pair* i.e. the source IP address, source port, destination IP address, destination port.

- **Initial Sequence Number (ISN)**: counter maintained in OS.
  - BSD increments it by 64000 every 500ms or new connection setup => time to wrap around < 9.5 hours.
TCP Connection Tear-down

Sender

FIN

Receiver

FIN-ACK

Data write

FIN

Data ack

FIN-ACK
TCP Connection Tear-down: FSM

1. **CLOSE**
   - send FIN

2. **FIN WAIT-1**
   - send FIN

3. **FIN WAIT-2**
   - rcv FIN
   - snd ACK
   - rcv FIN+ACK
   - snd ACK

4. **CLOSING**
   - rcv FIN
   - snd ACK
   - rcv ACK of FIN

5. **TIME WAIT**
   - rcv ACK of FIN
   - Timeout=2msl
   - delete TCB

6. **CLOSE WAIT**
   - send ACK
   - rcv FIN

7. **LAST-ACK**
   - CLOSE
   - snd FIN

8. **CLOSED**
   - CLOSE
   - delete TCB
Time Wait Issues

- Web servers not clients close connection first
  - Established $\rightarrow$ Fin-Waits $\rightarrow$ Time-Wait $\rightarrow$ Closed
- Why would this be a problem?
- Time-Wait state lasts for $2 \times \text{MSL}$
  - MSL should be 120 seconds (is often 60s)
  - Servers often have order of magnitude more connections in Time-Wait
Stop-and-Wait Efficiency

\[ \alpha = \frac{t_{\text{prop}}}{t_{\text{frame}}} = \frac{\text{Distance/Speed of Signal}}{\text{Frame size /Bit rate}} = \frac{\text{Distance} \times \text{Bit rate}}{\text{Frame size} \times \text{Speed of Signal}} \]

\[ U = \frac{t_{\text{frame}}}{2t_{\text{prop}} + t_{\text{frame}}} = \frac{1}{2\alpha + 1} \]

Light in vacuum
- \( = 300 \text{ m/\mu s} \)
Light in fiber
- \( = 200 \text{ m/\mu s} \)
Electricity
- \( = 250 \text{ m/\mu s} \)

No loss or bit-errors!
Sliding Window: Efficiency

Sender

Max ACK received  Next seqnum

.................................................................
Sent & Acked  Sent Not Acked
OK to Send  Not Usable

.................................................................
Receiver

Next expected  Max acceptable

.................................................................
Received & Acked  Acceptable Packet
Not Usable  Not Usable
Data

\[ U = \frac{N t_{\text{frame}}}{2t_{\text{prop}} + t_{\text{frame}}} \]

\[ = \begin{cases} \frac{N}{2\alpha + 1} & \text{if } N > 2\alpha + 1 \\ 1 & \text{if } N \leq 2\alpha + 1 \end{cases} \]

Note: no loss or bit-errors!
Go-Back-N

Sender:
- **k-bit** seq # in pkt header
  - Allows *upto* $N = 2^k - 1$ packets *in-flight*, unacked
- “Window”: *limit* on # of consecutive unacked pkts
- In GBN, window = $N$
Go-Back-N

- ACK(n): ACKs all pkts up to, including seq # n — “cumulative ACK”
- Sender may receive duplicate ACKs (see receiver)
  - Robust to losses on the reverse channel
  - Can pinpoint the first packet lost, but cannot identify blocks of lost packets in window
- One timer for oldest-in-flight pkt
- Timeout => retransmit pkt “base” and all higher seq # pkts in window
Selective Repeat: Sender, Receiver Windows

(a) sender view of sequence numbers

(b) receiver view of sequence numbers
Reliability Mechanisms: Summary

- **Checksum**: detects corruption in *pkts & acks*
- **ACK**: “packet correctly received”
  - **Duplicate ACK**: “packet incorrectly received”
  - **Cumulative ACK**: acks all pkts *upto & incl. seq #* (GBN)
  - **Selective ACK**: *acks pkt “n” only* (selective repeat)
- **Sequence number**: identifies packet or ack
  - 1-bit sequence number used *both in forward & reverse channels*
  - *k-bit sequence number in both forward & reverse channels.*
  - Let \( N = 2^k - 1 \) = sequence number space size
Reliability Mechanisms: Summary

- **Timeout** only at sender.
  - One timer for entire window (go-back-N)
  - One timer per pkt (selective repeat)
- **Window**: sender and receiver side.
  - *Limits* on what can be sent (or expected to be received).
  - Window size (W) upto N –1 (Go-back-N)
  - Window size (W) upto N/2 (Selective Repeat)
- **Buffering**
  - Only at sender (Go-back-N)
  - Out-of-order buffering at sender & receiver (Selective Repeat)
Reliability capabilities: Summary

- Provides reliable transmission over:
  - An *error-free* channel
  - A *forward & reverse* channel with *bit*-errors
  - Detects *duplicates* of packets/acks
  - *NAKs eliminated*
  - A *forward & reverse* channel with *packet*-errors (loss)

- *Pipelining efficiency:*
  - *Go-back-N*: Entire outstanding window retransmitted if pkt loss/error
  - *Selective Repeat*: only lost packets retransmitted
    - performance penalty if ACKs lost (because acks non-cumulative) & more complexity
What’s Different in TCP From Link Layers?

- Logical link vs. physical link
  - Must establish connection
- Variable RTT
  - May vary within a connection => Timeout variable
- Reordering
  - How long can packets live → max segment lifetime (MSL)
- Can’t expect endpoints to exactly match link rate
  - Buffer space availability, flow control
- Transmission rate
  - Don’t directly know transmission rate
Sequence Number Space

- Each byte in byte stream is numbered.
  - 32 bit value
  - Wraps around
  - Initial values selected at start up time
- TCP breaks up the byte stream in packets.
  - Packet size is limited to the Maximum Segment Size
- Each packet has a sequence number.
  - Indicates where it fits in the byte stream
MSS

- Maximum Segment Size (MSS)
- Largest “chunk” sent between TCPs.
  - Default = 536 bytes. Not negotiated.
  - Announced in connection establishment.
  - Different MSS possible for forward/reverse paths.
  - Does not include TCP header
- What all does this effect?
  - Efficiency
  - Congestion control
  - Retransmission
- Path MTU discovery
  - Why should MTU match MSS?
TCP Window Flow Control: Send Side

- Sent and acked
- Sent but not acked
- Not yet sent
- Next to be sent
Window Flow Control: Send Side

Packet Sent

- Source Port
- Dest. Port
- Sequence Number
- Acknowledgment
- HL/Flags
- Window
- D. Checksum
- Urgent Pointer
- Options...

Packet Received

- Source Port
- Dest. Port
- Sequence Number
- Acknowledgment
- HL/Flags
- Window
- D. Checksum
- Urgent Pointer
- Options...

acknowledged sent to be sent outside window
Window Flow Control: Receive Side

Receive buffer

- Acked but not delivered to user
- Not yet acked

window
Silly Window Syndrome

- Problem: (Clark, 1982)
  - If receiver advertises small increases in the receive window then the sender may waste time sending lots of small packets
- Solution
  - Receiver must not advertise small window increases
  - Increase window by \( \min(\text{MSS}, \text{RecvBuffer}/2) \)
Nagel’s Algorithm & Delayed Acks

- **Small packet problem:**
  - Don’t want to send a 41 byte packet for each keystroke
  - How long to wait for more data?

- **Solution: Nagel’s algorithm**
  - Allow only one outstanding small (not full sized) segment that has not yet been acknowledged

- **Batching acknowledgements:**
  - **Delay-ack timer:** piggyback ack on reverse traffic if available
  - 200 ms timer will trigger ack if no reverse traffic available
Timeout and RTT Estimation

- Problem:
  - Unlike a physical link, the RTT of a logical link can vary, quite substantially
  - How long should timeout be?
  - Too long => underutilization
  - Too short => wasteful retransmissions

- Solution: *adaptive timeout*: based on a **good estimate** of **maximum current value of RTT**
How to estimate max RTT?

- RTT = prop + queuing delay
  - Queuing delay highly variable
  - So, different samples of RTTs will give different random values of queuing delay

- Chebyshev’s Theorem:
  - MaxRTT = Avg RTT + k*Deviation
  - Error probability is less than 1/(k**2)
  - Result true for ANY distribution of samples
Round Trip Time and Timeout (II)

**Q:** how to estimate RTT?

- **SampleRTT:** measured time from segment transmission until ACK receipt
- **SampleRTT** will vary wildly
  - use several recent measurements, not just current
  - SampleRTT to calculate "AverageRTT"
  - **AverageRTT = (1-x)*AverageRTT + x*SampleRTT**
  - Exponential weighted moving average (EWMA)
  - Influence of given sample decreases exponentially fast; x = 0.1

**Setting the timeout**

Timeout = AverageRTT + 4*Deviation

Deviation = (1-x)*Deviation + x|SampleRTT - AverageRTT|
Timer Granularity

- Many TCP implementations set RTO in multiples of 200, 500, 1000ms
- Why?
  - Avoid spurious timeouts – RTTs can vary quickly due to cross traffic
  - Delayed-ack timer can delay valid acks by up to 200ms
  - Make timers interrupts efficient
- What happens for the first couple of packets?
  - Pick a very conservative value (seconds)
  - Can lead to stall if early packet lost…
Retransmission Ambiguity

Sample RTT

Original transmission

retransmission

ACK

RTO

A

B

Sample RTT

Original transmission

retransmission

ACK

RTO

A

B
Karn’s RTT Estimator

- Accounts for *retransmission ambiguity*
- If a segment has been retransmitted:
  - *Don’t update RTT estimators during retransmission.*
- Timer backoff: If timeout, RTO = 2*RTO
  - {exponential backoff}
  - Keep backed off time-out for next packet
- Reuse RTT estimate only after one successful packet transmission
Timestamp Extension

- Used to improve timeout mechanism by more accurate measurement of RTT
- When sending a packet, insert current timestamp into option
  - 4 bytes for seconds, 4 bytes for microseconds
- Receiver echoes timestamp in ACK
  - Actually will echo whatever is in timestamp
- Removes retransmission ambiguity!
  - Can get RTT sample on any packet
Recap: Stability of a Multiplexed System

Average Input Rate > Average Output Rate
=> system is unstable!

How to ensure stability?
1. Reserve enough capacity so that demand is less than reserved capacity
2. Dynamically detect overload and adapt either the demand or capacity to resolve overload
Congestion Problem in Packet Switching

- Cost: self-descriptive header per-packet, buffering and delays for applications.

- Need to **either reserve resources or dynamically detect/adapt to overload for stability**
Congestion: Tragedy of Commons

- Different sources compete for “common” or “shared” resources inside network.
- Sources are unaware of current state of resource
- Sources are unaware of each other
- Source has self-interest. Assumes that increasing rate by N% will lead to N% increase in throughput!
- Conflicts with collective interests: if all sources do this to drive the system to overload, throughput gain is NEGATIVE, and worsens rapidly with incremental overload => congestion collapse!!

Need “enlightened” self-interest!

- 10 Mbps
- 100 Mbps
- 1.5 Mbps
Congestion: A Close-up View

- **knee** – point after which
  - throughput increases very slowly
  - 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 = \( 1/(1 - \text{utilization}) \)
Congestion Control vs. Congestion Avoidance

- **Congestion control** goal
  - stay left of cliff

- **Congestion avoidance** goal
  - stay left of knee

- Right of cliff:
  - **Congestion collapse**
Congestion Collapse

- **Definition:** Increase in network load results in decrease of useful work done

- Many possible causes
  - Spurious retransmissions of packets still in flight
  - Undelivered packets
    - Packets consume resources and are dropped elsewhere in network
  - Fragments
    - Mismatch of transmission and retransmission units
  - Control traffic
    - Large percentage of traffic is for control
  - Stale or unwanted packets
    - Packets that are delayed on long queues
**Solution Directions....**

- **Problem:** *demand outstrips available capacity*

- If information about $\lambda_i$, $\lambda$, and $\mu$ is known in a central location where control of $\lambda_i$ or $\mu$ can be effected with zero time delays, the congestion problem is solved!

- Capacity ($\mu$) cannot be provisioned very fast => demand must be managed

- **Perfect callback:** Admit packets into the network from the user only when the network has capacity (bandwidth and buffers) to get the packet across.
Issues

- **If information** about $\lambda_i$, $\lambda$, and $\mu$ is **known in a central location** where control of $\lambda_i$ or $\mu$ can be effected with **zero time delays**, the congestion problem is solved!

- **Information/knowledge:** Only incomplete information about the congestion situation is known (e.g., loss indications, single bit, explicit rate field, measure of backlog etc)

- **Central vs distributed:** A distributed solution is required

- **Demand vs capacity control:** Usually only the demand is controllable on small time-scales. Capacity provisioning may be possible on larger time-scales.

- **Measurement/control points:** The congestion point, congestion detection/measurement point, and the control points may be different.

- **Time-delays:** Between the various points, there may be time-varying and heterogeneous time-delays
Q: Will the “congestion” problem be solved when:

- a) **Memory** becomes cheap (infinite memory)?
  - No buffer
  - Too late

- b) **Links** become cheap (high speed links)?
  - All links 19.2 kb/s
  - File Transfer time = 5 mins
  - Replace with 1 Mb/s
  - File Transfer Time = 7 hours
c) *Processors* become cheap (fast routers & switches)

Scenario: All links 1 Gb/s.
A & B send to C
=> “high-speed” congestion!!
(lose more packets faster!)
Two models of congestion control

1. End-to-end model:
   - End-systems is ultimately the source of “demand”
   - End-system must robustly estimate the timing and degree of congestion and reduce its demand appropriately
   - Must trust other end hosts to do right thing
   - Intermediate nodes relied upon to send timely and appropriate penalty indications (e.g., packet loss rate) during congestion
     - Enhanced routers could send more accurate congestion signals, and help end-system avoid other side-effects in the control process (e.g., early packet marks instead of late packet drops)
   - Key: trust and complexity resides at end-systems
   - Issue: What about misbehaving flows?
Two models of congestion control...

2. Network-based model:

- A) All end-systems cannot be trusted and/or
- B) The network node has more control over isolation/scheduling of flows

- Assumes network nodes can be trusted.
- Each network node implements isolation and fairness mechanisms (eg: scheduling, buffer management)
- A flow which is misbehaving hurts only itself

Problems:

- **Partial soln:** if flows don’t back off, each flow has congestion collapse, i.e. lousy throughput during overload
- Significant complexity in network nodes
- If some routers do not support this complexity, congestion still exists

- **Classic justification of the end-to-end principle**
Goals of Congestion Control

- To guarantee stable operation of packet networks
  - Sub-goal: avoid congestion collapse

- To keep networks working in an efficient status
  - Eg: high throughput, low loss, low delay, and high utilization

- To provide fair allocations of network bandwidth among competing flows in steady state
  - For some value of “fair” 😊
What is stability?

- Equilibrium point(s) of a dynamic system

- For packet networks
  - Each user will get an allocation of bandwidth
  - Changes of network or user parameters will move the equilibrium from one point, (hopefully) after a brief transient period, to a new one
  - System should not remain indefinitely away from equilibrium if there are no more external perturbations

- Example of instability: unbounded queue growth
What is fairness?

- one of the most over-defined (and probably over-rated) concepts
  - fairness index
  - max-min
  - proportional
  - ...
  - infinite number of notions!

- Fairness for best-effort service, roughly means that services are provided to selfish, competing users in a predictable way
Eg: max-min fairness

- If link not congested, then
  \[ f = \max(x_i) \]

- Otherwise, if link congested
  \[ \sum_{i} \min(x_i, f) = C \]

Allocations

- \( f = 4: \)
  - \( \min(8, 4) = 4 \)
  - \( \min(6, 4) = 4 \)
  - \( \min(2, 4) = 2 \)
Flow Control Optimization Model

- Given a set $S$ of flows, and a set $L$ of links
- Each flow $s$ has utility $U_s(x_s)$, $x_s$ is its sending rate
- Each link $l$ has capacity $c_l$
- Modeled as optimization (Eg: Kelly’98, Low’99)

$$\max \sum_{s \in S} U_s(x_s)$$

subject to

$$\sum_{s \in S_l} x_s \leq c_l, \forall l \in L$$

where $S_l = \{ s \mid \text{flow } s \text{ passes the link } l \}$
What is Fairness?

- Achieves \((w, \alpha)\) fairness if for any other feasible allocation \([\text{mo’00}]\):

\[
\sum_{s \in S} w_s \cdot \frac{x_s^* - x_s}{x_s^*} \leq 0
\]

where \(w_s\) is the weight for flow \(s\)

- weighted maximum throughput fairness is \((w, 0)\)
- weighted proportional fairness is \((w, 1)\)
- weighted minimum potential delay fairness is \((w, 2)\)
- weighted max-min fairness is \((w, \infty)\)
- “Weight” could be driven by economic considerations, or scheme dependencies on factors like RTT, loss rate etc.
What is fairness? (contd)

- fairness ($\alpha$-) axis

- $\alpha = 0$: maximum throughput fairness
- $\alpha = 1$: proportional fairness
- $\alpha = 2$: minimum delay fairness
- .......
- $\alpha = \infty$: max-min fairness
Proportional vs Max-min Fairness

- **proportional** fairness
  - the more a flow consumes critical network resources, the less allocation
  - network as a **white box**
  - network operators’ view
  - $f_0 = 0.1, f_{1-9} = 0.9$

- **max-min** fairness
  - every flow has the same right to all network resources
  - network as a **black box**
  - network users’ view
  - $f_0 = f_{1-9} = 0.5$
Equilibrium

- Operate at equilibrium near the knee point
- How to maintain equilibrium?
  - **Packet-conservation**: Don’t put a packet into network until another packet leaves.
  - Use ACK: send a new packet only after you receive and ACK. Why?
    - A.k.a “Self-clocking” or “Ack-clocking”
    - In steady state, keep # packets in network constant
- Problem: how do you know you are at the knee?
  - Network capacity or competing demand may change:
    - Need to probe for knee by increasing demand
    - Need to reduce demand overshoot detected
  - End-result: oscillate around knee
    - Violate packet-conservation each time you probe by the degree of demand increase
Implications of ack-clocking:

- More batching of acks => bursty traffic
- Less batching leads to a large fraction of Internet traffic being just acks (overhead)
Basic Control Model

- Let’s assume window-based operation
  - **Reduce window** when congestion is perceived
    - How is congestion signaled?
      - Either mark or drop packets
    - When is a router congested?
      - Drop tail queues – when queue is full
      - Average queue length – at some threshold
  - **Increase window** otherwise
    - Probe for available bandwidth – how?
Simple linear control

- Many different possibilities for reaction to congestion and methods for probing
  - Examine simple linear controls
  - \( \text{Window}(t + 1) = a + b \text{ Window}(t) \)
  - Different \( a_i/b_i \) for increase and \( a_d/b_d \) for decrease

- Supports various reaction to signals
  - Increase/decrease additively
  - Increased/decrease multiplicatively
  - Which of the four combinations is optimal?
Phase plots

- Simple way to visualize behavior of competing flows over time

- **Caveat:** assumes 2 flows, synchronized feedback, equal RTT, discrete “rounds” of operation
Additive Increase/Decrease

- Both $X_1$ and $X_2$ increase/decrease by the same amount over time
- Additive increase improves fairness & increases load
- Additive decrease reduces fairness & decreases load
Multiplicative Increase/Decrease

- Both $X_1$ and $X_2$ increase by the same factor over time
- Fairness unaffected (constant), but load increases (MI) or decreases (MD)
Additive Increase/Multiplicative Decrease (AIMD) Policy

Assumption: decrease policy must (at minimum) reverse the load increase over-and-above efficiency line

Implication: decrease factor should be conservatively set to account for any congestion detection lags etc
TCP Congestion Control

- Maintains three variables:
  - `cwnd` – congestion window
  - `rcv_win` – receiver advertised window
  - `ssthresh` – threshold size (used to update `cwnd`)
  - Rough estimate of knee point...

- For sending use: $\text{win} = \min(\text{rcv\_win}, \text{cwnd})$
TCP: Slow Start

- Goal: initialize system and discover congestion quickly
- How? Quickly increase $cwnd$ until network congested → get a rough estimate of the optimal $cwnd$
- How do we know when network is congested?
  - packet loss (TCP)
    - over the cliff here → congestion control
  - congestion notification (eg: DEC Bit, ECN)
    - over knee; before the cliff → congestion avoidance
- **Implications of using loss as congestion indicator**
  - Late congestion detection if the buffer sizes larger
  - Higher speed links or large buffers => larger windows
    => higher probability of burst loss
- Interactions with retransmission algorithm and timeouts
TCP: Slow Start

- Whenever starting traffic on a new connection, or whenever increasing traffic after congestion was experienced:
  - Set $cwnd = 1$
  - Each time a segment is acknowledged increment $cwnd$ by one ($cwnd++$).

- Does Slow Start increment slowly? Not really. In fact, the increase of $cwnd$ is exponential!!
  - Window increases to $W$ in $RTT \times \log_2(W)$
Slow Start Example

- The congestion window size grows very rapidly.

- TCP slows down the increase of cwnd when \textit{cwnd} \geq \textit{ssthresh}
Slow Start Example

One RTT

0R

1

One pkt time

1R

1

2

3

2R

2

3

4

6

5

7

3R

4

5

6

7

8

10

12

14

9

11

13

15
Slow Start Sequence Plot

Window doubles every round
Congestion Avoidance

- Goal: maintain operating point at the left of the cliff:
- How?
  - **additive increase:** starting from the rough estimate (ssthresh), slowly increase cwnd to probe for additional available bandwidth
  - **multiplicative decrease:** cut congestion window size aggressively if a loss is detected.
Congestion Avoidance

- Slow down “Slow Start”

- If $cwnd > ssthresh$ then each time a segment is acknowledged increment $cwnd$ by $1/cwnd$
  i.e. $(cwnd += 1/cwnd)$.

- So $cwnd$ is increased by one only if all segments have been acknowledged.

- (more about $ssthresh$ latter)
Congestion Avoidance Sequence Plot

Window grows by 1 every round
Assume that $ssthresh = 8$
Putting Everything Together: TCP Pseudo-code

Initially:
   cwnd = 1;
   ssthresh = infinite;

New ack received:
   if (cwnd < ssthresh)
      /* Slow Start*/
      cwnd = cwnd + 1;
   else
      /* Congestion Avoidance */
      cwnd = cwnd + 1/cwnd;

Timeout: (loss detection)
   /* Multiplicative decrease */
   ssthresh = win/2;
   cwnd = 1;

while (next < unack + win)
   transmit next packet;

where win = min(cwnd, flow_win);

Shivkumar Kalyanaraman
Rensselaer Polytechnic Institute
The big picture

- Slow Start
- Timeout
- Congestion Avoidance

Time

cwnd
Packet Loss Detection: Timeout Avoidance

- Wait for Retransmission Time Out (RTO)
- What’s the problem with this?
  - Because RTO is a performance killer
- In BSD TCP implementation, RTO is usually more than 1 second
  - the granularity of RTT estimate is 500 ms
  - retransmission timeout is at least two times of RTT

- Solution: Don’t wait for RTO to expire
  - Use alternate mechanism for loss detection
  - Fall back to RTO only if these alternate mechanisms fail.
Fast Retransmit

- Resend a segment after 3 duplicate ACKs
  - Recall: a duplicate ACK means that an out-of-sequence segment was received

Notes:
- duplicate ACKs due to packet reordering!
- if window is small don’t get duplicate ACKs!
Fast Recovery (Simplified)

- After a fast-retransmit set $cwnd$ to $ssthresh/2$
  - i.e., don’t reset $cwnd$ to 1
- But when RTO expires still do $cwnd = 1$

- Fast Retransmit and Fast Recovery $\rightarrow$
  implemented by TCP Reno; most widely used version of TCP today
Fast Retransmit and Fast Recovery

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

Sequence No

Time

Retransmission
Duplicate Acks
Multiple Losses

Sequence No

Time

Now what?
Retransmission
Duplicate Acks
TCP Version: *Tahoe*
TCP Versions: *Reno*

Now what? - timeout
NewReno

- The ack that arrives after retransmission (partial ack) should indicate that a second loss occurred.
- When does NewReno timeout?
  - When there are fewer than three dupacks for first loss
  - When partial ack is lost
- How fast does it recover losses?
  - One per RTT
Now what? – partial ack recovery
SACK

- Basic problem is that cumulative acks only provide little information
  - Alt: Selective Ack for just the packet received
  - What if selective acks are lost? \(\rightarrow\) carry cumulative ack also!

- Implementation: Bitmask of packets received
  - Selective acknowledgement (SACK)
  - Only provided as an optimization for retransmission
  - Fall back to cumulative acks to guarantee correctness and window updates
Now what? – send retransmissions as soon as detected
Asymmetric Behavior

- Three important characteristics of a path
  - Loss
  - Delay
  - Bandwidth
- Forward and reverse paths are often independent even when they traverse the same set of routers
- Many link types are unidirectional and are used in pairs to create bi-directional link

![Diagram showing asymmetry in bandwidth and loss](attachment:diagram.png)
Asymmetric Loss

- Loss
  - Information in acks is very redundant
  - Low levels of ack loss will not create problems
  - TCP relies on ack clocking – will **burst** out packets when cumulative ack covers large amount of data
    - Burstiness will in turn cause queue overflow/loss
  - Max burst size for TCP and/or simple rate pacing
    - Critical also during restart after idle
Ack Compression

- What if acks encounter queuing delay?
  - Smooth ack clocking is destroyed
    - Basic assumption that acks are spaced due to packets traversing forward bottleneck is violated
  - Sender receives a burst of acks at the same time and sends out corresponding burst of data
- Has been observed and does lead to slightly higher loss rate in subsequent window
Could congestion on the reverse path ever limit the throughput on the forward link?

Let’s assume MSS = 1500 bytes and delayed acks:

- For every 3000 bytes of data need 40 bytes of acks
- 75:1 ratio of bandwidth can be supported
- Modem uplink (28.8 Kbps) can support 2 Mbps downlink
- Many cable and satellite links are worse than this

Solutions: Header compression, link-level support
TCP Congestion Control Summary

- **Sliding** window limited by receiver window.
- Dynamic **windows**: slow start (exponential rise), congestion avoidance (additive rise), multiplicative decrease.
  - Ack clocking
- Adaptive **timeout**: need mean RTT & deviation
- Timer backoff and Karn’s algo during retransmission

- Go-back-N or Selective **retransmission**
- Cumulative and Selective **acknowledgements**
- **Timeout avoidance**: Fast Retransmit