TCP Lectures
Paul Francis
TCP service

• TCP is a quasi-reliable byte-stream transport service
  • As long as the TCP connection is established, bytes arrive in the order they were sent
• Quasi-reliable, because:
  • Weak checksum
  • Termination can be botched
• TCP (the protocol) provides flow control
• TCP (the implementation) provides congestion control
TCP flow control

TCP prevents loss here

Avoids & recovers from loss everywhere

network
Some TCP issues we’ll look at

• TCP uses a sliding window
• However, TCP must contend with different issues
  • Round trip may vary
    • (so don’t know how best to fill the pipe)
  • The network may be congested
    • (so may need to go even slower than receive window allows)
• Packets may not arrive in order
• A TCP connection has to synchronize the beginning and end (SOS and EOS)
TCP bytes and “segments”
TCP Header (segment)

Flags: SYN FIN RESET PUSH URG ACK

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgement</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hdr len</th>
<th>Flags</th>
<th>Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Checksum</th>
<th>Urgent pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Options (variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
</tr>
</tbody>
</table>
Connection Establishment (with Initial Sequence Numbers)

Connection Setup
3-way handshake
Connection terminate

• Connection establish is fairly straightforward
• Connection terminate is more complex
  • Because both sides must fully close
  • One side can close while the other still sends the last of its data
  • Or both can close at once
TCP Connection terminate

Closing a connection:

client closes socket:
    `clientSocket.close();`

**Step 1:** client end system
    sends TCP FIN control segment to server

**Step 2:** server receives
    FIN, replies with ACK.
    Closes connection, sends FIN.

[Diagram of TCP connection termination with arrows showing close, FIN, and ACK messages between client and server.]
TCP Connection Terminate

**Step 3:** client receives FIN, replies with ACK.
- Enters “timed wait” - will respond with ACK to received FINs

**Step 4:** server, receives ACK. Connection closed.

**Note:** with small modification, can handle simultaneous FINs.
TIME_WAIT state

• On client, Wait 2 times Maximum Segment Lifetime (2 MSL)
  • Provides protection against delayed segments from an earlier incarnation of a connection being interpreted as for a new connection

• Maximum time segment can exist in the network before being discarded
  • Time-To-Live field in IP is expressed in terms of hops not time
  • TCP estimates it as 2 minutes
TIME_WAIT state

• During this time, combination of client IP and port, server IP and port cannot be reused
  • Some implementations say local port cannot be reused at all while it is involved in time wait state even to establish a connection to different dest IP/port combo
TCP advertised window

- **app**
  - **app is emptying buffer**
  - **TCP ACKs this even if not seen by ACK**
  - **sender is filling buffer**

- **receive buffer**
TCP flow control

if sender buffer fills, socket won't allow more writes (it blocks)

if buffer fills, window shrinks to zero
if window is zero, senders' buffer fills

if app doesn't read, this buffer fills
TCP retransmission mechanism originally Go-back-N

- Say sender sends bytes 1000 – 1499, in 5 100-byte packets
- Receiver ACKs up to 1100
- Sender knows that receiver missed packet 1100-1199, but doesn’t know about other three packets
- Sender “goes back” to 1100, and starts retransmitting everything
- It may therefore resend received packets
  - Lots of them, if the pipe is long and fat
Later TCP added Selective Acknowledgement (SACK)

- Use TCP option space during ESTABLISHED state to send “hints” about data received ahead of acknowledged data
- TCP option that says SACK enabled on SYN => “I am a SACK enabled sender, receiver feel free to send selective ack info”
- Normal ACK field still authoritative!
- SACK usage is growing, but still not universal
SACK Details

• Format:

```
+------------+
| Kind=5     |
+------------+
| Length     |
+------------+
| Left Edge 1st Block |
+------------+
| Right Edge 1st Block |
+------------+
| ...        |
+------------+
| Left Edge nth Block |
+------------+
| Right Edge nth Block |
+------------+
```

- TCP option 5
- In 40 bytes of option can specify a max of 4 blocks
- If used with other options space reduced
- Ex. With Timestamp option (10 bytes), max 3 blocks
TCP sliding window

Window Size

Data ACK'd
Outstanding Un-ack'd data
Data OK to send
Data not OK to send yet
Big Fat Pipes and TCP

- TCP has a 32-bit sequence number space, and a 16-bit window size (65Kbytes)
- At 1.2 Gbps:
  - the seq number can wrap in 28 seconds
  - The delay x BW at 100ms is 14.8MB
    - 200 window’s worth!!!
TCP extensions for big fat pipes (RFC 1323)

• Timestamp extension
  • Allows the sender to put a 32-bit timestamp in the header
  • Mainly for RTT estimation
    • Receiver echoes it back, sender gets an accurate RTT
    • But receiver can also use it to detect wraparound

• Window scaling extension
  • Negotiate to interpret window as power-of-2 factor (i.e., left-shift window X bits)
TCP performance

- Making interactive TCP efficient for low-bandwidth links
- Filling the pipe for bulk-data applications
- Estimating round trip time (RTT)
- Keeping the pipe full
- Avoiding congestion
When to schedule transmission

• As we saw, TCP segment transmit doesn’t have to correspond to app send()

• When should TCP send a fragment?
  • As soon as it gets data to send?
  • As soon as it has a packet’s worth to send (MSS Max Segment Size)?
  • Not until some timer goes off?
When to schedule transmission

• If TCP sends right away, it may send many small packets
• If TCP waits for a full MSS, it may delay important data
• If TCP waits for a timer, then bad behavior can result
  • Lots of small packets get sent anyway
  • Silly Window Syndrome
Silly Window Syndrome

- This is a nice situation:
  - (nice big packets, full pipe)
Silly Window Syndrome

• Imagine this situation:
  • How could we get out of it???
Silly Window Syndrome

• Small packets introduced into the loop tend to stay in the loop
• How do small packets get introduced into the loop?
Silly Window Syndrome: Small packet introduced
Silly Window Syndrome prevention

• Receiver and sender both wait until they have larger segments to ACK or send

• Receiver:
  • Receiver will not advertise a larger window until the window can be increased by one full-sized segment or
  • by half of the receiver’s buffer space whichever is smaller
Silly Window Syndrome prevention

• Sender:
  • Waits to transmit until either a full sized segment (MSS) can be sent or
  • at least half of the largest window ever advertised by the receiver can be sent or
  • it can send everything in the buffer
When to schedule transmission (again)

- App can force sender to send immediately when data is available
  - Sockopt TCP_NODELAY
- Otherwise, sender sends when a full MSS is available
- Or when a timer goes off
  - But with silly window constraints...
TCP: Retransmission and Timeouts

TCP uses an adaptive retransmission timeout value:
- Congestion
- Changes in Routing

RTT changes frequently

Next few slides from Nick McKeown, Stanford
TCP: Retransmission and Timeouts

Picking the RTO is important:

- Pick a value too big and it will wait too long to retransmit a packet,
- Pick a value too small, and it will unnecessarily retransmit packets.

The original algorithm for picking RTO:

1. $\text{EstimatedRTT}_k = \alpha \text{EstimatedRTT}_{k-1} + (1 - \alpha) \text{SampleRTT}$
2. $RTO = 2 \times \text{EstimatedRTT}$

Characteristics of the original algorithm:

- Variance is assumed to be fixed.
- But in practice, variance increases as congestion increases.
TCP: Retransmission and Timeouts

- There will be some (unknown) distribution of RTTs.
- We are trying to estimate an RTO to minimize the probability of a false timeout.

- Router queues grow when there is more traffic, until they become unstable.
- As load grows, variance of delay grows rapidly.

<table>
<thead>
<tr>
<th>Load (Amount of traffic arriving to router)</th>
<th>Variance grows rapidly with load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Queueing Delay</td>
<td></td>
</tr>
</tbody>
</table>

![Graph showing probability distribution of RTTs with mean and variance labeled.](graph1.png)

![Graph showing average queueing delay with load and variance labeled.](graph2.png)
TCP: Retransmission and Timeouts

*Karn’s Algorithm*

**Problem:**
How can we estimate RTT when packets are retransmitted?

**Solution:**
On retransmission, don’t update estimated RTT (and double RTO).
TCP: Retransmission and Timeouts

Newer Algorithm includes estimate of variance in RTT:

- Difference = SampleRTT - EstimatedRTT
- EstimatedRTT\textsubscript{k} = EstimatedRTT\textsubscript{k-1} + (\delta \text{Difference})
- Deviation = Deviation + \delta (|\text{Difference}| - \text{Deviation})

- RTO = \mu \times \text{EstimatedRTT} + \phi \times \text{Deviation}

  \mu \approx 1
  \phi \approx 4

Decaying average, as before
Fast Retransmit

• Even with all this fancy RTT estimation, retransmits still tend to over-estimate, and TCP can stall while waiting for a time-out
  • Stall because pipe often bigger than window!
• This leads to the notion of “fast retransmit”
Delayed connection

Pipe is full.
Life is good!
Delayed connection

sender

receiver

window

timeout

sender stalled all this time
Fast Retransmit

- Receiver should send an ACK every time it receives a packet, not only when it gets something new to ACK
  - If same bytes are ACK’d, this is called “duplicate ACK”
- Sender interprets 3 duplicate ACKs as a loss signal, retransmits right away
  - Don’t wait for timeout
TCP congestion control

• How TCP avoids and controls congestion in the network
• Without this, TCP still won’t perform well...
What is congestion?

• Lets distinguish between a strict definition of congestion and a working definition of congestion
• Strictly:
  • Congestion occurs anytime more than one packet competes for the same link at the same time
Question:

• Do we want to prevent instances of multiple packets competing for the same link at the same time?
Answer:

• No!
• Pure circuit networks avoid ever having two packets compete for the same link at the same time
  • (more or less)
• By reserving a fixed amount of bandwidth at each link for each connection
• But for bursty traffic, utilization is low!
Queues in switches

- Queues deal with congestion at packet timescales
  - Two packets arrive at the same time, one is queued behind the other
- Queues allow us to increase the utilization of links
  - At the expense of packet delay
- In this sense, packet timescale congestion is actually good!
Delay and throughput

• With no queue, sender can never send at more than 400Kbps
• If sender bursty, then bursts are limited to 400Kbps,
  • with links unused during periods between bursts
Delay and throughput

- With a queue, sender can burst at 10Mbps
- Burst will start to fill the queue
- After burst is over, queue empties into slow link
  - Link utilized during silent periods!

![Diagram showing 10Mbps link connected to a router, with a 400Kbps link on the other side.]

Load, delay and power

Typical behavior of queuing systems with random arrivals:

A simple metric of how well the network is performing:

$$Power = \frac{Load}{Delay}$$

Slide from Nick McKeown, Stanford
Our definition of congestion

• Where network load is large enough that queues overflow and packets are lost
• We are also concerned with “congestion collapse”
Load, delay, and throughput: what’s wrong with this picture??
Queue’s aren’t infinite, packets get dropped

Congestion Collapse!
Why congestion collapse?

- Lost packets leads to retransmissions
- Retransmissions add to load, resulting in more lost packets
- Packets may go several hops before being dropped
  - Using up resources along the way
- Note congestion collapse doesn’t occur where there are no retransmissions
TCP was causing congestion collapse

• In the late 1980’s---Internet was becoming unusable!
• Solution attributed to Van Jacobsen
• Problem was that the network did not signal the host when there was congestion
  • ICMP source quench wasn’t widely implemented
TCP congestion control

• Basic idea:
  • TCP gently “probes” the network to determine its capacity
  • Uses dropped packets as a sign of congestion
  • Backs off when congestion sensed
TCP congestion control goals

• First and foremost, prevent congestion collapse
• Also, fairly apportion resources
  • Each TCP flow gets an equal amount of the link bandwidth
• While achieving good performance
  • Keep the pipe full, but not too full!
Ideal TCP behavior

• Bottleneck bandwidth determines inter-packet spacing
  • Sender should space packets
How can TCP sender space packets properly?

• Any ideas?
How can TCP sender space packets properly?

• Simple solution: use returned ACKs to clock packets out!
Ideal TCP behavior

• Get the pipe full
• Once full, use return ACKs to clock out new packets

• Now the question is, how to you know when the pipe is full???
Answer:

• You don’t know when the pipe is full!
• You only know when it is too full!
  • When there is a packet loss
  • Actually, more recent work challenges this...
• So, what TCP does is slowly fill the pipe until it is too full, then drain the pipe some and start filling again . . .
TCP congestion control

- Sender maintains two windows:
  - The advertised receive window we learned about
  - A congestion window (cwnd)
- The actual window is the minimum of the two:
  - Window = min{Advertized window, cwnd}
- In other words, send at the rate of the slowest component: network or receiver
Setting the congestion window (cwnd)

• Increase cwnd conservatively

• Decrease cwnd aggressively
  • When loss detected, cut in half!
  • Multiplicative decrease

• Cwnd increase has two phases:
  • Additive phase (when pipe is full)
  • Multiplicative phase (when pipe is empty)
    • Called “slow start”!
AIMD: Additive Increase Multiplicative Decrease

• Used when pipe is full
• Every RTT, add one “packet” to the cwnd
  • Actually, one MSS worth of bytes
  • Since multiple ACKs per RTT, a fraction of MSS added per ACK
• If loss detected (timeout or duplicate ACKs), decrease cwnd by half
Additive Increase
The famous AIMD sawtooth

Could take a long time to get started!
“Slow start”

• Additive increase takes too long to fill pipe when pipe is empty
  • i.e. at the beginning of a connection
• During slow start, double the cwnd every RTT
  • Increase the cwnd for every ACK received
Slow start
Two reasons for an empty pipe

• Beginning of the connection
  • In this case, do “slow start” until packet loss

• Restart after a “stalled connection”
  • If timeout, then the pipe is empty
  • In this case, we remember the previous cwnd
  • Do slow start until cwnd reaches 1/2 the previous cwnd, then do additive
Slow start

Window

Timeouts

halved

Exponential “slow start”

Slow start in operation until it reaches half of previous cwnd.
Slow start packet sequence
Continued
(slow start to $\frac{1}{2}$ previous cwnd)
Fast Recovery

• Recall fast retransmit
  • Retransmit after three duplicate ACKs (don’t wait for a timeout)
• We can also use the duplicate ACKs to avoid dropping all the way back to slow start
• This is called fast recovery (always implemented as part of fast retransmit)
Fast Retransmit and Recovery

• If we get 3 duplicate acks for segment N
  • Retransmit segment N
  • Set ssthresh to 0.5*cwnd
  • Set cwnd to ssthresh + 3

• For every subsequent duplicate ack
  • Increase cwnd by 1 segment

• When new ack received
  • Reset cwnd to ssthresh (resume congestion avoidance)
Fast Recovery Example

- Fast retransmit after 3 dup ACKs
- Fast recovery due to additional dup ACKs
TCP performance again...

- TCP performs poorly if the pipe empties.
- The pipe empties if a timeout occurs.
- A timeout occurs if not enough packets were sent after the lost packet to trigger fast retransmit.
- Unfortunately, drop-tail is likely to drop the last packets of a burst!
  - Drop-tail is router drop policy that drops all packets that overflow the queue.
Random Early Detection (RED)

• Modifies the router drop policy to make TCP perform better
• Drop occasional packets before the queue is full, to avoid dropping many packets from a burst
• Select packet to drop randomly, so that a given burst will have only a single packet dropped
  • And later packets passed to trigger fast retransmission
RED

- Queue has two thresholds, min and max
- If queue below min, don’t drop
- If queue above max, drop all received packets
- If queue between min and max, drop received packet with some probability
  - Increase probability with time from last drop
But why drop at all????

Congestion Avoidance

• Dropping not so bad for a long file transfer
• But can be noticeable for interactive applications
• Would be nice to avoid dropping at all
• Two ways to avoid dropping:
  • Explicit Congestion Notification from routers
  • Detect increasing queues before a drop occurs
ECN (Explicit Congestion Notification)

• DECNet had this back in the 80s!
  • “DEC bit”
• Router sets a bit in the packet if queues are above a threshold
  • Receiver echoes bit back to sender
• ECN is an IETF standard for use in conjunction with RED (RFC3168)
  • Two IP TOS bits defined for this
  • Plus new flags defined for TCP
Source-based congestion avoidance

• A router’s queue starts to fill when the outgoing link capacity is reached
• When this happens, the sender will see:
  • Constant throughput (because capacity has been reached)
  • Increasing RTT (because of increasing wait in router queue)
• Sender can look for this, and back-off before packets are dropped!
TCPSIM: Time evolution of a TCP flow

Packet Drops [Pkts/s*10]  
Utilization [Pkts/s]  
Sending Rate [Pkts/s]  
Cong. Window [Pkts*10]  
Effective RTT [ms*100]  

Buffer Occupancy [Pkts]
Additional TCP issues

- TCP assumes that a timeout is the result of a lost packet due to congestion
- But, on many wireless links, timeouts occur because of temporary bad reception
- We don’t want the sender to back-off!
- Often a TCP-aware box placed at the Internet-wireless interface can trick the sender into not backing off
Additional TCP issues

• TCP performs poorly on very large delay X bandwidth pipes
• Takes too long to fill the pipe (slow start)
  • Performance dominated by RTT
• In this case, would like routers to tell the sender what cwnd to use from the start!
  • XCP