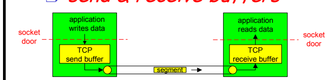
7: TCP

Last Modified:
2/25/2003 8:15:19 PM

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries" like with UDP datagrams
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*

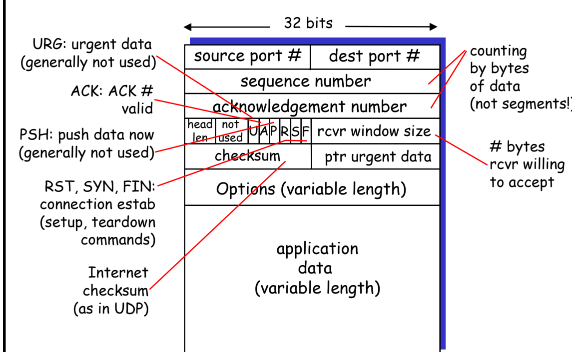


- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow and congestion controlled:
  - sender will not overwhelm receiver or network

Roadmap

- TCP header and segment format
- Connection establishment and termination
- Normal Data flow

TCP segment structure



TCP Headers: like UDP?

- Source and destination port numbers
- Checksum
- Data length? Rely on length in IP header?

TCP Headers: Familiar?

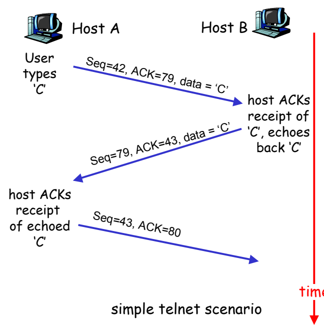
- Sequence Number field (32 bit)
  - Sequence Number field indicates number of first byte in the packet
- Receiver Window Size (16 bit)
  - Window like for GBN or selective repeat, but window size not fixed – variable based on receiver feedback
- Acknowledgment Field (32 bit)
  - The acknowledgement field contains the next sequence number it is expecting thus implicitly acknowledging all previous segments.
  - Cumulative acks not individual acks or negative acks

TCP seq. #'s and ACKs

Seq. #'s:
- byte stream "number" of first byte in segment's data

ACKs:
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments
- A: TCP spec doesn't say, - up to implementor
- Can buffer or not, in either case still ACK next in order byte expected



simple telnet scenario

TCP Header: Header Length

- Header Length (4 bits)
  - needed because options field make header variable length
  - Expressed in number of 32 bit words = 4 bytes
  - 4 bits field => 4 bytes*2^4 = 60 bytes; 20 bytes of required header gives 40 bytes possible of options
  - Recall UDP header was 8 bytes

Implications of Field Length

- 32 bits for sequence number (and acknowledgement) ; 16 bits for advertised window size
- Implication for maximum window size? Window size <= $\frac{1}{2}$ SequenceNumberSpace
  - Requirement easily satisfied because receiver advertised window field is 16 bits
    - $2^{32} \gg 2 * 2^{16}$
    - Even if increase possible advertised window to 2^{31} that would still be ok

Implications of Field Length (cont)

- Advertised Window is 16 bit field => maximum window is 64 KB
  - Is this enough to fill the pipeline? Not always
  - Pipeline = delay*BW product
  - 100 ms roundtrip and 100 Mbps => 1.19 MB
- Sequence Number is 32 bit field => 4 GB
  - Wrap –around?
  - Maximum Segment Lifetime of 120 seconds
  - Will this ever wrap too soon? Yes it might
    - 4 GB/120 sec = 273 Mbps
    - Gigabit Ethernet? STS-12 at 622 Mbps?

TCP Header: Common Options

- Options used to extend and test TCP
- Each option is:
  - 1 byte of option kind
  - 1 byte of option length (except for kind = 0 for end of options and kind =1 for no operation)
- Examples
  - window scale factor: if don't want to be limited to 2^{16} bytes in receiver advertised window
  - timestamp option: if 32 bit sequence number space will wrap in MSL; add 32 bit timestamp to distinguish between two segments with the same sequence number
  - Maximum Segment Size can be set in SYN packets

TCP Header: Flags (6 bits)

- Connection establishment/termination
  - SYN – establish; sequence number field contains valid initial sequence number
  - FIN - terminate
- RESET - abort connection because one side received something unexpected
- PUSH - sender invoked push to send
- URG – indicated urgent pointer field is valid; special data - record boundary
- ACK - indicates Acknowledgement field is valid

TCP Header: ACK flag

- ACK flag – if on then acknowledgement field valid
- Once connection established no reason to turn off
  - Acknowledgment field is always in header so acknowledgements are free to send along with data

TCP Header: URG

- If URG flag on, then URG pointer contains a positive offset to be added to the sequence number field to indicate the last byte of urgent data
- No way to tell where urgent data starts – left to application
- TCP layer informs receiving process that there is urgent data

Out-of-band data

- URG is not really out-of-band data; Receiver must continue to read byte stream till reach end of urgent data
- If multiple urgent segments received, first urgent mark is lost; just one urgent pointer
- How to get out-of-band data if need it?
  - Separate TCP connection?

URG

- How helpful is this?
- Telnet and Rlogin use URG when user types the interrupt key; FTP uses when user aborts a file transfer
- Is this worth a whole header field and a flag?
- Doesn't help that implementations vary in how they interpret the urgent pointer (point to last byte in urgent data or byte just past the last byte of urgent data)

TCP Header: PSH

- Intention: use to indicate not to leave the data in a TCP buffer waiting for more data before it is sent
  - In practice, programming interface rarely allows application to specify
  - Instead TCP will set if this segment used all the data in its send buffer
- Receiver is supposed to interpret as deliver to application immediately; most TCP/IP implementations don't delay delivery in the first place though

TCP Header: Data boundaries?

- In general with UDP, application write of X bytes data results in a UDP datagram with X bytes of data – not so with TCP
- In TCP, the stream of bytes coming from an application is broken at arbitrary points by TCP into the "best" size chunks to send
- Sender may write 10 bytes then 15 then 30 but this is not in general visible to the receiver

Record Boundaries

- Could try to use URG and PSH to indicate record boundaries
  - socket interface does not notify app that push bit or urgent bit is on though!
- If need record boundaries, applications must always insert their own by indicating it in the data (ie. Data is record len + record format)

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. `RcvWindow`)
- *client:* connection initiator `Socket clientSocket = new Socket("hostname","port number");`
- *server:* contacted by client `Socket connectionSocket = welcomeSocket.accept();`

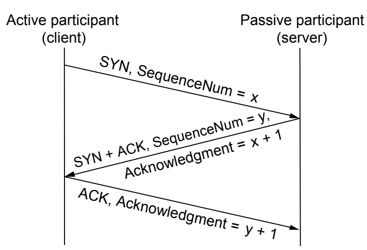
Three way handshake:

Step 1: client end system sends TCP SYN control segment to server
- specifies initial seq #

Step 2: server end system receives SYN, replies with SYNACK control segment
- ACKs received SYN
- allocates buffers
- specifies server-> receiver initial seq. #

Step 3: client acknowledges servers initial seq. #

Three-Way Handshake



Note: SYNs take up a sequence number even though no data bytes

Connection Establishment

- Both data channels opened at once
- Three-way handshake used to agree on a set of parameters for this communication channel
  - Initial sequence number for both sides
  - Receiver advertised window size for both sides
  - Optionally, Maximum Segment Size (MSS) for each side; if not specified MSS of 536 bytes is assumed to fit into 576 byte datagram

Initial Sequence Numbers

- Chosen at random in the sequence number space?
- Well not really randomly; intention of RFC is for initial sequence numbers to change over time
  - 32 bit counter incrementing every 4 microseconds
- Vary initial sequence number to avoid packets that are delayed in network from being delivered later and interpreted as a part of a newly established connection

Special Case: Timeout of SYN

- Client will send three SYN messages
  - Increasing amount of time between them (ex. 5.8 seconds after first, 24 seconds after second)
- If no responding SYNACK received, client will stop trying to open the connection

Special Case: Simultaneous active SYNs

- Possible (but improbable ?) for two ends to generate SYNs for the same connection at the same time
- SYNs cross in the network
- Both reply with SYNACK and connection is established

Connection Termination

- Each side of the bi-directional connection may be closed independently
  - 4 messages: FIN message and ACK of that FIN in each direction
- Each side closes the data channel it can send on
- One side can be closed and data can continue to flow in the other direction, but not usually
- FINs consume sequence numbers like SYNs

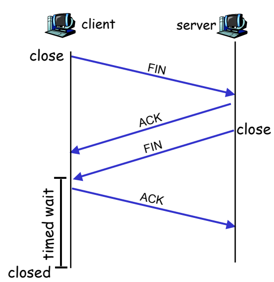
TCP Connection Management (cont.)

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.

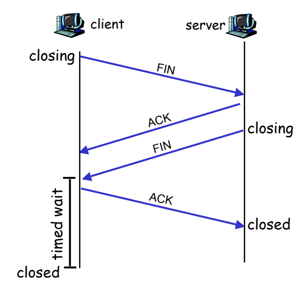


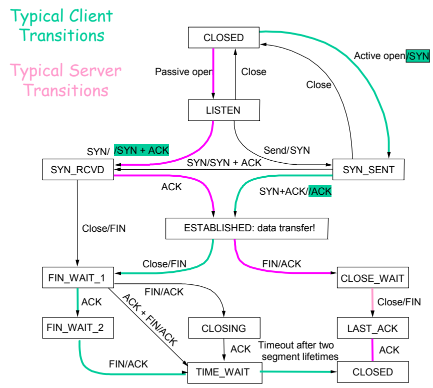
TCP Connection Management (cont.)

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.

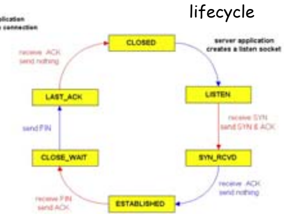




TCP Connection Management

TCP client lifecycle

TCP server lifecycle



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
- 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

Netstat

- netstat –a –n
  - Shows open connections in various states
  - Example:

Active Connections

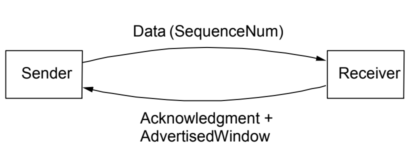
Proto	LocalAddr	ForeignAddr	State
TCP	0.0.0.0:23	0.0.0.0:0	LISTENING
TCP	192.168.0.100:139	207.200.89.225:80	CLOSE_WAIT
TCP	192.168.0.100:1275	128.32.44.96:22	ESTABLISHED
UDP	127.0.0.1:1070	*:*	

RST

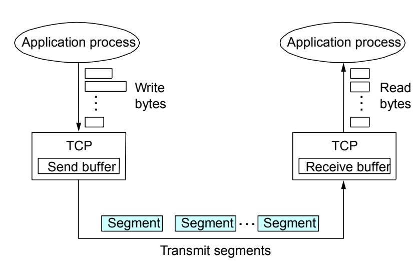
- RST flag
- Abortive release of a connection rather than the orderly release with FINs
- Client web browsers often end their connections that way - not good form but faster

Data Transfer in the ESTABLISHED state

Data Transfer (Simplified One-Way)



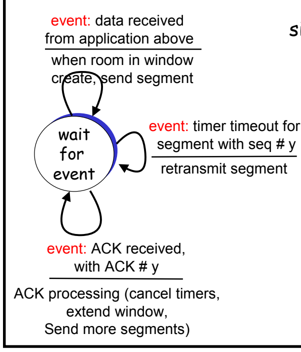
TCP connection: One Direction



Segment Transmission

- Maximum segment size reached
  - If accumulate MSS worth of data, send
  - MSS usually set to MTU of the directly connected network (minus TCP/IP headers)
- Sender explicitly requests
  - If sender requests a push, send
- Periodic timer
  - If data held for too long, sent

TCP Sender: Simplified State Machine



simplified sender, assuming

- one way data transfer
- no flow, congestion control
- Also assuming synchronous sends at the application layer (not buffer and send later)

TCP Sender: Simplified Pseudo-code

Simplified TCP sender

```
sendbase = initial_sequence number
nextseqnum = initial_sequence number

loop (forever) {
  switch(event)
  event: data received from application above
      create TCP segment with sequence number nextseqnum
      start timer for segment nextseqnum
      pass segment to IP
      nextseqnum = nextseqnum + length(data)
  event: timer timeout for segment with sequence number y
      retransmit segment with sequence number y
      compue new timeout interval for segment y
      restart timer for sequence number y
  event: ACK received, with ACK field value of y
      if (y > sendbase) { /* cumulative ACK of all data up to y */
         cancel all timers for segments with sequence numbers < y
         sendbase = y
         }
      else { /* a duplicate ACK for already ACKed segment */
         increment number of duplicate ACKs received for y
         if (number of duplicate ACKS received for y == 3) {
            /* TCP fast retransmit */
            resend segment with sequence number y
            restart timer for segment y
         }
  } /* end of loop forever */
```


TCP Receiver: ACK generation

[RFC 1122, RFC 2581]

Event	TCP Receiver action
in-order segment arrival, no gaps, everything else already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
in-order segment arrival, no gaps, one delayed ACK pending	immediately send single cumulative ACK
out-of-order segment arrival higher-than-expect seq. # gap detected	send duplicate ACK, indicating seq. # of next expected byte (sender can use as hint of selective repeat)
arrival of segment that partially or completely fills gap	immediate ACK if segment starts at lower end of gap

TCP Details: Roadmap

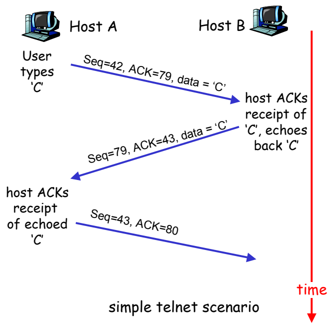
- Data Flow
  - Interactive
  - Bulk Data
- Timeout/Retransmission
- Slow Start/ Congestion Avoidance

Interactive data: Small packets

- Example: Telnet/Rlogin
  - Send each interactive key stroke in a separate TCP packet
  - server side echos that same character back to be displayed on the local screen
- How big are these TCP packets containing a single byte of data?
  - 1 byte data
  - 20 bytes (at least) for TCP header
  - 20 bytes for IP header
  - < 3% data!
- Do we want to fill the pipeline with small packets like this?

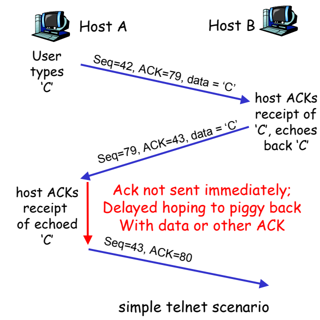
Piggybacking ACKs

- Telnet/Rlogin: each interactive key stroke in a separate TCP packet
- Server side echos that same character back to be displayed on the local screen
- ACK of data is piggy backed on echo of data



Delayed ACKs

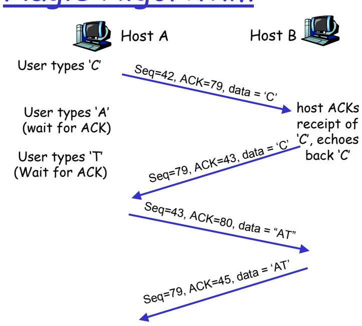
- **Problem:** Would like to send more data at once or at least piggyback the acks
- **Solution:** Delay the ACK for some time hoping for some data to go in the other direction or for more incoming data for a cumulative ack
- **Can we do better than this?**



Nagle Algorithm

- If a TCP connection has outstanding data for which an acknowledgement has not yet been received, do not send small segments
  - Instead wait for an acknowledgement to be received then send all data collected to that point
  - If collect MSS, go ahead and send without waiting for ACK
- Adjusts to network conditions
  - If ACKs coming back rapidly (like on a LAN), data will be sent rapidly
  - If ACKs coming back slowly (like on a WAN), will collect more data together in that time to send together

Nagle Algorithm



Able to send AT together
In one TCP segment rather than
Each having one

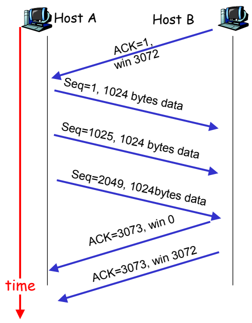
Experiment: Interactive Data

- Use Ethereal to trace a telnet or rlogin session

Bulk Data Transfer

- Don't have any problem collecting full size TCP segments
- Receiver may have trouble keeping up with sender
  - Use advertised window to throttle the sender
  - Some problems with small window sizes though….

Bulk Data Transfer

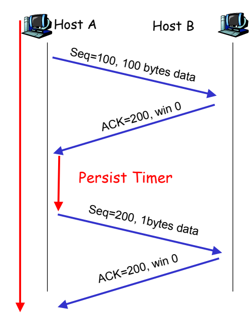


- Receiver will send ACKs of data received but with reduced window sizes
- When window opens up (I.e. app reads data from kernel buffers), send a "window update" message

Lost Window Update?

- What if the last window update message is lost?
  - Receiver waiting for data
  - Sender not allowed to send anything
- Solutions?
  - Set timer on receiver after sending window update; If don't here from sender retransmit
  - Sender periodically sends 1 byte of data even if window is 0
- Which do you think was chosen? Internet Principle of putting complexity on sender?

TCP Persist Timer



- Sender set persist timer when window size goes to 0
- When timer expires, sends a "window probe" message (TCP packets with 1 byte of data)
- If receiver still has window 0, it will send an ack but the ack will not cover the "illegal" 1 byte just sent

Silly Window Syndrome

- Occurs when small amounts of data are exchanged over a connection instead of large amounts
  - Sender only knows they can send X bytes of data
  - Receiver can really take 2X but hasn't had a chance to announce it; gets X bytes so can only advertise X again
- Solutions?
  - Receiver doesn't advertise small windows; Instead waits till larger window opens up
  - Sender holds off sending data till larger amount accumulated
  - Which? **In this case both**

Preventing Silly Window

- 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
- 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

Bulk Data: Fully Utilizing the Link

- How do we fully utilize the link? (Hint: we saw this before)
- Need window large enough to fill the pipeline
- Window >= bandwidth * round trip time
- Note: If use window scaling option not limited to 64K

Fully utilizing the link?

- Receiver's advertised window
- Header overhead
- Ack traffic in other direction
- ..

Experiment: Bulk Data

- Use Ethereal to trace an ftp session
- Use ttcp to generate a TCP stream on a quiet local network – how close to peak network capacity?

Interactive vs Bulk

- Interactive tries to accumulate as much data together as possible without compromising acceptable interactive experience
  - Delayed Acks
  - Nagle Algorithm
- Bulk has no problem with accumulating data together, but can have problem with overwhelming the receiver
  - Receiver Advertised Window
  - Persist Timer
- Bulk also tries to fully utilize the link (interactive has no chance of doing that)

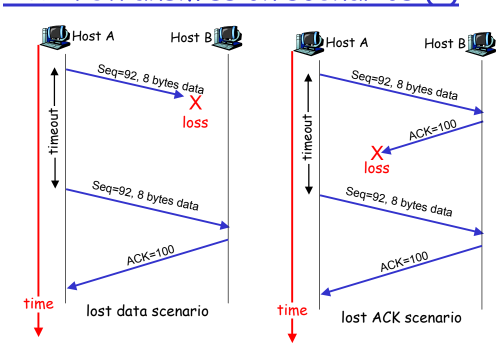
Roadmap

- Data Flow
  - Interactive
  - Bulk Data
- Timeout and Retransmission
- Slow Start and Congestion Avoidance

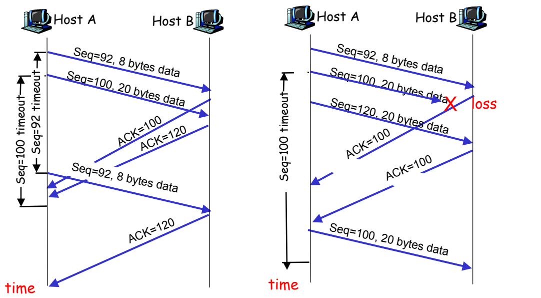
Timeout and Retransmission

- Receiver must acknowledge receipt of all packets
- Sender sets a timer if acknowledgement has not arrived before timer expires then sender will retransmit packet
- Adaptive retransmission: timer value computed as a function of average round trip times and variance

TCP: retransmission scenarios (1)



TCP: retransmission scenarios (2)



premature timeout, cumulative ACKs

Duplicate ACK, fast retransmit (really need 3 dup acks before fast retransmit)

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?
- Based on RTT
  - but longer than RTT to avoid premature time out because RTT will vary
- Tensions
  - too short: premature timeout = unnecessary retransmissions
  - too long: slow reaction to segment loss

Q: how to estimate RTT?
- `SampleRTT`: note time when packet sent; when receive ACK, RTT = currentTime – sentTime
  - Not 1:1 correspondence between segments sent and ACKs
  - ignore retransmissions, cumulatively ACKed segments (Not part of original spec; Karn and Partridge 1987)
- `SampleRTT` will vary, want estimated RTT "smoother"
  - use several recent measurements, not just current `SampleRTT`

TCP Round Trip Time Estimate

```
EstimatedRTT = (1-x)*EstimatedRTT + x*SampleRTT
```

- Exponential weighted moving average
- Influence of given sample decreases exponentially fast
- Typical value of x: 0.1 (90% weight to accumulated average; 10% to new measurement)
- Larger x means adapts more quickly to new conditions Would this be good?
  - Yes if real shift in base RTT; No if leads to jumpy reactions to transient conditions
  - Which is more likely?

Original TCP Timeout Calculation

We've estimated RTT, now how do we set the timeout?

- `EstimtedRTT` plus "safety margin"
- large variation in `EstimatedRTT` -> larger safety margin

```
Timeout = EstimatedRTT * DelayVarianceFactor

     Recommended DelayVarianceFactor = 2
```

Problems?

- Observe problems in the presence of wide variations in RTT [Jacobson1988]
- Hypothesis: Better if base Timeout on both mean and variance of RTT measurements

Jacobson/Karels Timeout Calculation

Base on Mean and Variance

- Mean deviation good approximation of standard deviation but easier to compute (no square root ☺)

```
EstimatedRTT = (1-x)*EstimatedRTT + x*SampleRTT

Error = |SampleRTT-EstimatedRTT|

Deviation = Deviation +
            h*(Error – Deviation)

Timeout = EstimatedRTT + 4*Deviation
```

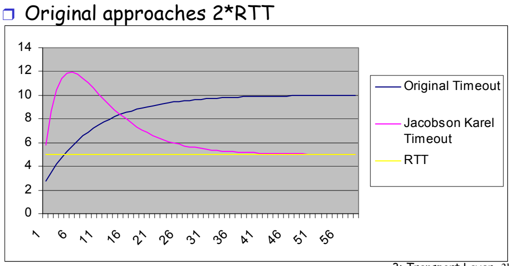
- Recommended: x =0.125 (higher than for original); Timeout responds more rapidly to changes in RTT
- Recommended: h = 0.25

Experiment

- Experiment with a spreadsheet to see how the calculated timeout times changes with changes in the measured round trip time
- Experiment with Original vs Jacobson/Karels
- Can also experiment with alternate methods of estimating the round trip time
- See RTTall.xls for an example

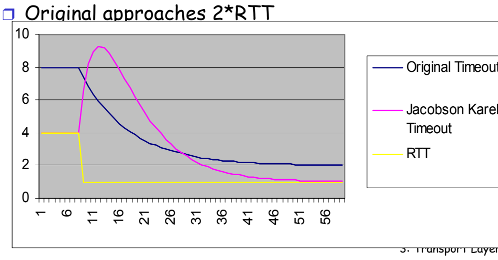
RTT 1 to 5

- RTT steady at 1 – transitions to steady at 5
- Original has timeouts; Jacobson Karel doesn't
- Jacobson/Karel approaches the RTT exactly
- Original approaches 2*RTT



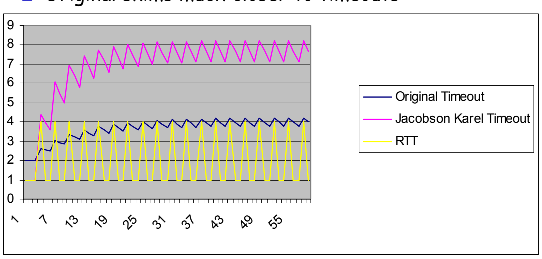
RTT 4 to 1

- RTT steady at 4 – transitions to steady at 1
- Even though transition down; Jacobson Karel timeout spikes up
- Jacobson/Karel approaches the RTT exactly
- Original approaches 2*RTT



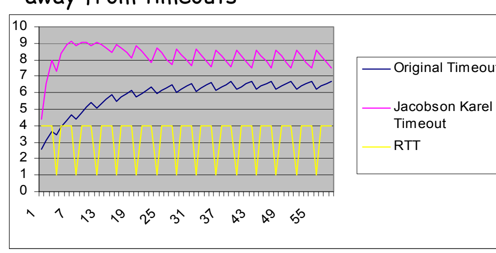
RTT Periodic Spike Up

- RTT 1 except every N is 4 (here N = 4)
- Jacobson/Karel stays well away from timeouts
- Original skims much closer to timeouts



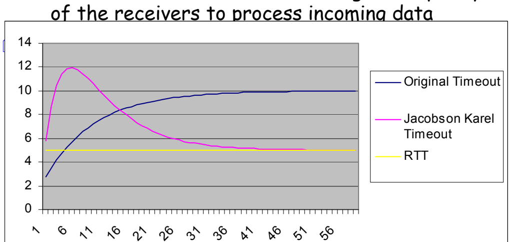
RTT Periodic Spike Down

- RTT 4 except every N is 1 (here N = 4)
- Both Original and Jacobson/Karel stay well away from timeouts



Flow Control vs Congestion Control

- Flow Control
  - Prevent senders from overrunning the capacity of the receivers to process incoming data



Principles of Congestion Control

Congestion:

- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- a top-10 problem!

Congestion Prevention?

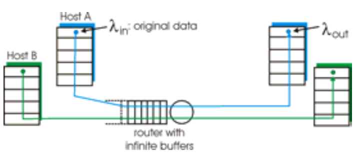
- In a connection-oriented network:
  - Prevent congestion by requiring resources to be reserved in advance
- In a connectionless network:
  - No prevention for congestion, just detect congestion and react appropriately (congestion control)

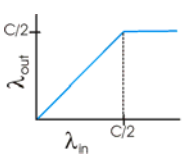
Detecting congestion?

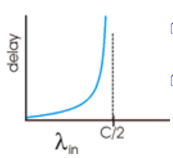
- Network could inform sender of congestion
  - Explicit notification: Routers can alter packet headers to notify end hosts
- Senders notice congestion for themselves?
  - Lost packets:If there are more packets than resources (ex. Buffer space) along some path, then no choice but to drop some
  - Delayed packets: Router queues get full and packets wait longer for service

Causes/costs of congestion: Increased Delays

- two senders, two receivers
- one router, infinite buffers
- no retransmission



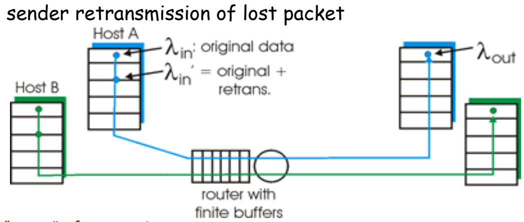




- large delays when congested
- maximum achievable throughput

Causes/costs of congestion: Retransmission

- one router, *finite* buffers
- sender retransmission of lost packet



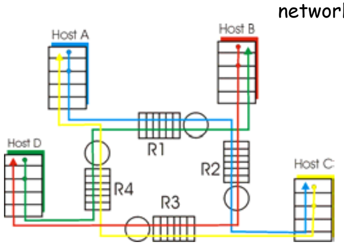
"costs" of congestion:

- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt

Causes/costs of congestion: Upstream capacity wasted

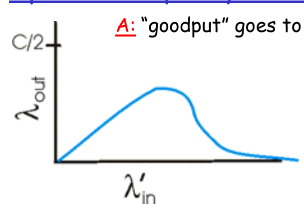
- four senders
- multihop paths
- timeout/retransmit

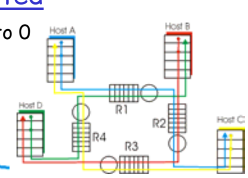
Q: what happens as λ_{in} and λ'_{in} increase (I.e. send more and more into a congested network ?



Causes/costs of congestion: Upstream capacity wasted

A: "goodput" goes to 0





Another "cost" of congestion:

- when packet dropped, any "upstream transmission capacity used for that packet was wasted!

How important is this?

- No congestion control = Congestion Collapse
- As number of packets entering network increases, number of packets arriving at destination increases but only up to a point
- Packet dropped in network => all the resources it used along the way are wasted => no forward progress
- Internet 1987

TCP Details: Roadmap

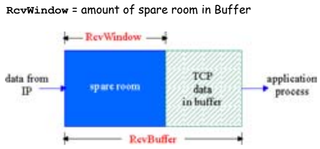
- TCP Flow Control
- Slow Start/ Congestion Avoidance
- TCP Fairness
- TCP Performance
- Transport Layer Wrap-up

TCP Flow Control

flow control
sender won't overrun receiver's buffers by transmitting too much, too fast

`RcvBuffer` = size or TCP Receive Buffer

`RcvWindow` = amount of spare room in Buffer



receiver buffering

receiver: explicitly informs sender of (dynamically changing) amount of free buffer space
- `RcvWindow` field in TCP segment

sender: keeps the amount of transmitted, unACKed data less than most recently received `RcvWindow`

TCP Congestion Control

- No explicit feedback from network layer (IP)
- Congestion inferred from end-system observed loss, delay
- Limit window size by both receiver advertised window *and* a "congestion window"
  - ActualWindow < = minimum (ReceiverAdvertised Window, Congestion Window)

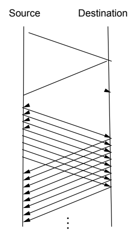
TCP Congestion Control: Two Phases

- Don't just send the entire receiver's advertised window worth of data right away
- Start with a congestion window of 1 or 2 packets and a threshold typically the receiver's advertised window
- Slow Start (Multiplicative Increase): For each ack received, double window up until a threshold
- Congestion Avoidance (Additive Increase): Fore each RTT, increase window by 1;

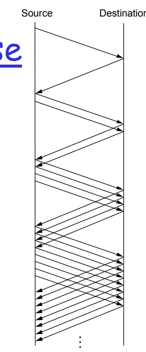
Slow Start vs Congestion Avoidance

- Two important variable
  - Congwin = current congestion window
  - Threshhold = boundary between multiplicative increase and additive increase
- Below threshhold we are in slow start; Above threshhold we are congestion avoidance
- In slow start, congwin goes up multiplicatively in a RTT; In congestion avoidance congwin goes up additively in a RTT
- Both congwin and threshhold will vary over the lifetime of a TCP Connection!

Original: With Just Flow Control



"Slow" Start: Multiplicative Increase

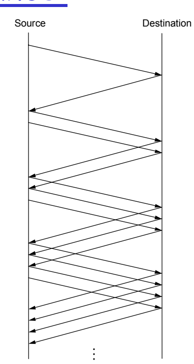


Multiplicative Increase Up to the Threshold

"Slower" than full receiver's advertised window

Faster than additive increase

TCP Congestion Avoidance: Additive Increase



Additive Increase Past the Threshhold

For each RTT, add 1 MSS segment to the congestion window

Typically done as small increments based on each ack rather than a single increase by MSS after acks for complete window

TCP congestion control:

- Even additive increase can't go on for ever, right?
- "probing" for usable bandwidth and eventually will hit the limit
  - ideally: transmit as fast as possible (`Congwin` as large as possible) without loss but in reality keep stepping off cliff and then adjusting
- Loss is inevitable
  - *increase* `Congwin` until loss (congestion)
  - loss: *decrease* `Congwin`, then begin probing (increasing) again
- **Question is how to "detect" loss and how to react to it?**

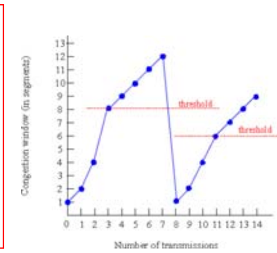
Timeout

- The most obvious way to detect losses is with the timeout of retransmission timer
- For large values of congwin and large RTTs this will have a big penalty
- Consider window of 10 MSS segments
  - Sender transmits 1-10; First is lost
  - In best case, retransmission timer won't expire until > ~2*RTT; then retransmission traverses network and ACK travels back (another RTT)
  - So lose more than two full windows (2*RTT worth of data transmissions)
- Also TCP imposes an even larger penalty in adjustments to congwin (1) and threshhold (cut in half)

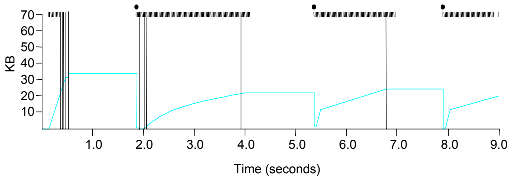
TCP Congestion Avoidance: Multiplicative Decrease too

Congestion avoidance

```
/* slowstart is over     */
/* Congwin > threshold */
Until (loss event) {
  every w segments ACKed:
     Congwin++
 }
threshold = Congwin/2
Congwin = 1
perform slowstart
```



Connection Timeline



- blue line = value of congestion window in KB
- Short hash marks = segment transmission
- Long hash lines = time when a packet eventually retransmitted was first transmitted
- Dot at top of graph = timeout
- 0-0.4 Slow start; 2.0 timeout, start back at 1;
- 5.5-5.6 slow start; 5.6 – 6.8 congestion avoidance

Fast Retransmit

- Signs of loss besides timeout?
- Interpret 3 duplicate acks (ie 4 acks for the same thing) as an early warning of loss
  - other causes? Reordering or duplication in network
- Retransmit packet immediately without waiting for retransmission timer to expire
- If getting ACKS can still rely on them to clock the connection

Fast Retransmit

- Recall window of 10 MSS segments
  - Sender transmits 1-10; First is lost
  - In best case, retransmission timer won't expire until > ~2*RTT; then retransmission traverses network and ACK travels back (another RTT)
  - So lose more than two full windows (2*RTT worth of data transmissions) without fast retransmit
  - With retransmit, will get dup ack triggered by receipt of 2,3,4,5 then will retransmit 1 so only loose ½ RTT
- In addition, TCP imposes a lighter penalty in terms of adjustments to congwin and threshhold
  - Fast Recovery..

Fast Recovery

- After a fast retransmit,
  - threshold = ½ (congestion window)
  - But do not set Congestion window = 1
  - Instead Congestion Window = threshold + 3* MSS
  - If more dup acks arrive, congestion Window += MSS
  - Transmit more segments if allowed by the new congestion window
- Why +MSS for each duplication ack?
  - Artificially inflate the congestion window for packets we expect have left the network (triggered dup ack at receiver)
- Finally, when ack arrives for new data,deflate congestion window back to threshold
  - congestionWindow = threshold
  - Still better than back to 1 though!

TCP Congestion Control History

- Before 1988, only flow control!
- TCP Tahoe 1988
  - TCP with Slow-Start, Congestion Avoidance and Fast Retransmit
- TCP Reno 1990
  - Add fast recovery (and delayed acknowledgements)
- TCP Vegas 1993
- TCP NewReno and SACK 1996
- TCP FACK
- …..

TCP Vegas

- Sender side only modifications to TCP including
  - Higher precision RTT calculations
  - Don't wait for low precision timeout to occur if higher precision difference between segment transmit time and time dup ack received indicates timeout should have already occurred
  - If a non-dup ACK is received immediately after a retransmission, check to see if any segment should have already timed out and if so retransmit
  - Avoid reducing congwin several times for same window (reduce congwin only due to losses that occurred at new lower rate!)
- Vegas in not a recommended version of TCP

TCK SACK

- Adds selective acknowledgements to TCP
  - Like selective repeat
- How do you think they do it?
  - TCP option that says SACK enabled on SYN => "I am a SACK enabled sender, receiver feel free to send selective ack info"
  - Use TCP option space during ESTABLISHED state to send hints about data received ahead of acknowledged data
- Does not change meaning of normal Acknowledgement field in TCP Header
- Receiver allowed to renege on SACK hints

Details

- TCP option 5 sends SACK info
- Format:

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

- 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 New Reno

- Proposed and evaluated in conjunction with SACK
- Modified version of Reno that avoids some of Reno's problems when multiple packets are dropped in a single window of data
- Conclusion?
  - SACK not required to solve Reno's performance problems when multiple packets dropped
  - But without SACK, TCP constrained to retransmit at most one dropped packet per RTT or to retransmit packets that have already been successful received (heart of the Go-Back N vs Selective Repeat discussion)

Other

- TCP FACK (Forward Acknowledgments)
- TCP Rate-Halving
  - Evolved from FACK
- TCP ECN (Explicit Congestion Notification)

Game Theory Analysis of TCP

- Game theory = Balance cost and benefit of greedy behavior
  - Benefit of higher send rate = higher receive rate
  - Cost of higher send rate = higher loss rate
- Balance point for Reno is relatively efficient
- SACK reduces the cost of a loss so changes the balance in favor of more aggressive behavior
- Balance point for flow control only? Favors aggressive behavior even more
- Note: TCP based on Additive Increase Multiplicative Decrease (AIMD); Show AIAD would be stable as well

Status

- Reno is most widely deployed
- SACK/FACK/ECN being deployed slowly?
  - NetBSD has SACK/FACK?ECN
  - SACK Turned on by default in Windows 98 but not later Windows
- Why?
  - Performance Improvements not sufficiently dramatic
  - Less stable in face of greedy behaviour (Sigcomm 2002)

TCP latency modeling

Q: How long does it take to receive an object from a Web server after sending a request?

A: That is a natural question, but not very easy to answer.

Even if you know BW and round trip time, depends on loss profile (remember loss is fundamental), receiver's advertised window

- Model slow start and congestion avoidance separately and then alternate between then based on loss profile

TCP Latency Model: Fixed Window

If assume no losses , two cases to consider:

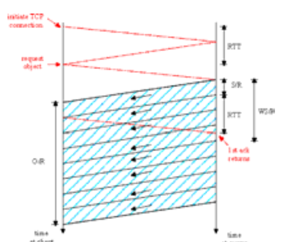
- Slow Sender (Big Window): Still sending when ACK returns
  - time to send window > time to get first ack
  - W*S/R > RTT + S/R
- Fast Sender (Small Window):Wait for ACK to send more data
  - time to send window < time to get first ack
  - W*S/R < RTT + S/R

Notation, assumptions:

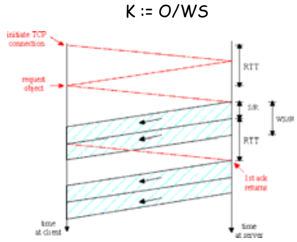
- O: object size (bits)
- R: Assume one link between client and server of rate R
- W: number of segments in the fixed congestion window
- S: MSS (bits)
- no retransmissions (no loss, no corruption)

TCP Latency Model: Fixed Window

Number of windows:
K := O/WS



Slow Sender (Big Window):
latency = 2RTT + O/R



Fast Sender (Small Window):
latency = 2RTT + O/R
+ (K-1)[S/R + RTT - WS/R]

(S/R + RTT) – (WS/R) = Time Till Ack Arrives – Time to Transmit Window

TCP Latency Modeling: Slow Start

- Now suppose window grows according to slow start (not slow start + congestion avoidance).
- Latency of one object of size O is:

$$Latency = 2RTT + \frac{O}{R} + P\left[RTT + \frac{S}{R}\right] - (2^P - 1)\frac{S}{R}$$

where *P* is the number of times TCP stalls at server waiting for Ack to arrive and open the window:

$$P = \min\{Q, K-1\}$$

- Q is the number of times the server would stall if the object were of infinite size - maybe 0.
- K is the number of windows that cover the object.
- S/R is time to transmit one segment
- RTT+ S/R is time to get ACK of one segment

TCP Latency Modeling: Slow Start (cont.)

Example:

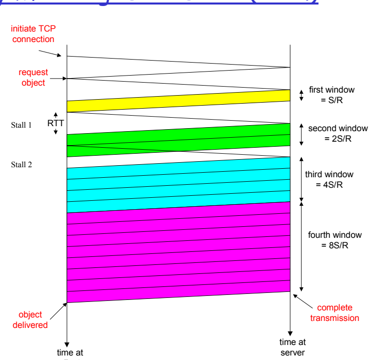
O/S = 15 segments

K = 4 windows

Q = 2

P = min{K-1,Q} = 2

Server stalls P=2 times.



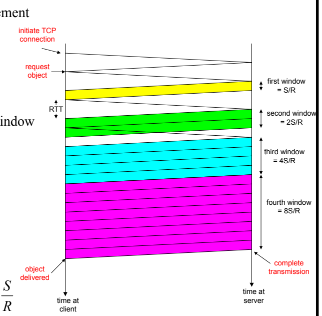
TCP Latency Modeling: Slow Start (cont.)

$\frac{S}{R} + RTT$ = time from when server starts to send segment until server receives acknowledgement

$2^{k-1}\frac{S}{R}$ = time to transmit the kth window

$\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]^+$ = stall time after the *k*th window

$$\begin{aligned} \text{latency} &= \frac{O}{R} + 2RTT + \sum_{p=1}^{P} stallTime_p \\ &= \frac{O}{R} + 2RTT + \sum_{k=1}^{P}\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right] \\ &= \frac{O}{R} + 2RTT + P\left[RTT + \frac{S}{R}\right] - (2^P - 1)\frac{S}{R} \end{aligned}$$



TCP Performance Modeling

- Add in congestion avoidance
  - At threshhold switch to additive increase
- Add in periodic loss
  - Assume kept in congestion avoidance rather than slow start
- Modeling short connections that are dominated by start-up costs
- More general model
  - Model of loss
  - Model of queuing at intermediate links
  - ...

TCP Performance Limits

- Can't go faster than speed of slowest link between sender and receiver
- Can't go faster than receiverAdvertisedWindow/RoundTripTime
- Can't go faster than dataSize/(2*RTT) because of connection establishment overhead
- Can't go faster than memory bandwidth (lost of memory copies in the kernel)

"Overclocking" TCP with a Misbehaving Receiver

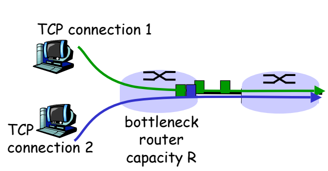
- Optimistic ACKing
  - Send acks for data not yet received
  - If never indicate loss, can ramp TCP send rate through the roof over a long connection!
  - Of course might really loose data that way
- DupAck spoofing
  - Deliberately send dup acks to trigger window inflation
- ACK division
  - Instead of trying to send as few ACKS as possible, send as many as possible
  - Exploits TCP implementation that updates congwin for each ACK rather than explicitly by 1 segment each RTT
  - Dup acks increase congwin ½ as slowly for the same reason

Experiment: Compare TCP and UDP performance

- Use ttcp (or pcattcp) to compare effective BW when transmitting the same size data over TCP and UDP
- UDP not limited by overheads from connection setup or flow control or congestion control
- Use Ethereal to trace both

TCP Fairness

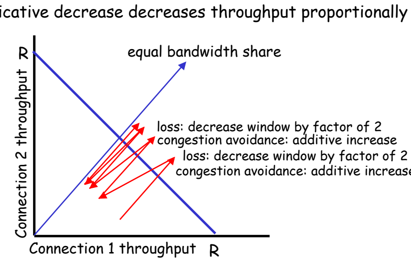
Fairness goal: if N TCP sessions share same bottleneck link, each should get 1/N of link capacity



Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally
-



Bandwidth Sharing

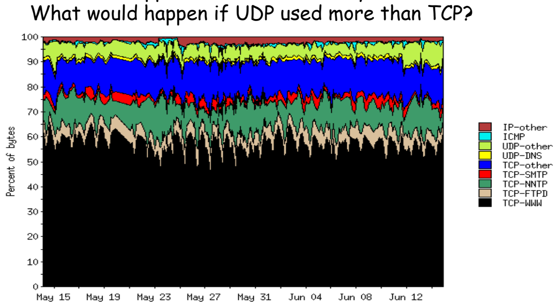
- Multiple TCP streams sharing a link will adjust to share the link fairly (assuming losses get distributed evenly among them)
- Multiple TCP streams in the presence of a UDP stream
  - UDP will take over BW and TCP streams will all drop to nothing
- "TCP Friendly"
  - Respond to signs of congestion and back off agressively like TCP
  - "No no no after you"

TCP vs UDP

- TCP has congestion control; UDP does not
- TCP has flow control; UDP does not
- TCP does retransmission; UDP does not
- TCP delivers in-order; UDP does not
- TCP has connection setup and close; UDP does not
- TCP obeys MSS; UDP reproduces app level send (stream vs datagram)
- TCP has higher header overhead (20-60 vs 8 bytes)
- UDP can be used for multicast/broadcast

% TCP vs % UDP

Apps like reliable delivery!
What would happen if UDP used more than TCP?



Transport Layer Summary

- principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation and implementation in the Internet
  - UDP
  - TCP

Next:

- leaving the network "edge" (application transport layer)
- into the network "core"