CSE 461: TCP (part 2)

Ben Greenstein
Jeremy Elson

TA: Ivan Beschastnikh

Thanks to Tom Anderson and Ratul Mahajan for slides
Transport: Practice

Protocols

- IP -- Internet protocol
- UDP -- user datagram protocol
- TCP -- transmission control protocol
- RPC -- remote procedure call
- HTTP -- hypertext transfer protocol
- And a bunch more...
How do we connect processes?

IP provides host to host packet delivery
- header has source, destination IP address

For applications to communicate, need to demux packets sent to host to target app
- Web browser (HTTP), Email servers (SMTP), hostname translation (DNS), RealAudio player (RTSP), etc.
- Process id is OS-specific and transient
Ports

Port is a mailbox that processes “rent”
- Uniquely identify communication endpoint as (IP address, protocol, port)

How do we pick port #’s?
- Client needs to know port # to send server a request
- Servers bind to “well-known” port numbers
  - Ex: HTTP 80, SMTP 25, DNS 53, ...
  - Ports below 1024 reserved for “well-known” services
- Clients use OS-assigned temporary (ephemeral) ports
  - Above 1024, recycled by OS when client finished
Sockets

OS abstraction representing communication endpoint
- Layer on top of TCP, UDP, local pipes

server (passive open)
- bind -- socket to specific local port
- listen -- wait for client to connect

client (active open)
- connect -- to specific remote port
User Datagram Protocol (UDP)

Provides application – application delivery
Header has source & dest port #'s
  - IP header provides source, dest IP addresses
Deliver to destination port on dest machine
Reply returns to source port on source machine
No retransmissions, no sequence #s
=> stateless
UDP Delivery
A brief Internet history...

- 1969: ARPANET created
- 1970: ARPANET
- 1972: TELNET, RFC 318
- 1973: FTP, RFC 454
- 1977: MAIL, RFC 733
- 1982: TCP & IP, RFC 793 & 791
- 1984: DNS, RFC 883
- 1986: NNTP, RFC 977
- 1990: ARPANET dissolved
- 1991: WWW/HTTP
- 1992: MBONE
- 1995: Multi-backbone Internet

1990 ARPANET dissolved

1991 WWW/HTTP
TCP: This is your life...

1975
Three-way handshake
Raymond Tomlinson
In SIGCOMM 75

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

1975
TCP & IP
RFC 793 & 791

1982
TCP & IP

1983
Nagel’s algorithm
to reduce overhead of small packets;
predicts congestion collapse

1984
BSD Unix 4.2
support TCP/IP

1985
Congestion collapse observed

1986
Karn’s algorithm
to better estimate round trip time

1987
Van Jacobson’s algorithms
congestion avoidance and congestion control
(most implemented in 4.3BSD Tahoe)

1988
Congestion collapse observed

1990
4.3BSD Reno
fast retransmit delayed ACK’s
TCP: After 1990

1993
TCP Vegas
(Brakmo et al)
real congestion avoidance

1994
TCP

1994
ECN
(Floyd)
Explicit Congestion Notification

1994
T/TCP
(Braden)
Transaction TCP

1996
SACK TCP
(Floyd et al)
Selective Acknowledgement

1996
Hoe
Improving TCP startup

1996
FACK TCP
(Mathis et al)
extension to SACK

2006
PCP
Transmission Control Protocol (TCP)

Reliable bi-directional byte stream
- No message boundaries
- Ports as application endpoints

Sliding window, go back N/SACK, RTT est, ...
- Highly tuned congestion control algorithm

Flow control
- prevent sender from overrunning receiver buffers

Connection setup
- negotiate buffer sizes and initial seq #s
- Needs to work between all types of computers (supercomputer -> 8086)
TCP Packet Header

Source, destination ports
Sequence # (bytes being sent)
Ack # (next byte expected)
Receive window size
Checksum
Flags: SYN, FIN, RST
TCP Delivery

Application process

TCP
Send buffer

Write bytes

TCP
Receive buffer

Read bytes

Transmit segments

Segment
Segment
Segment

IP x.html

IP TCP get inde
TCP Sliding Window

Per-byte, not per-packet (why?)
- send packet says “here are bytes j-k”
- ack says “received up to byte k”

Send buffer $\geq$ send window
- can buffer writes in kernel before sending
- writer blocks if try to write past send buffer

Receive buffer $\geq$ receive window
- buffer acked data in kernel, wait for reads
- reader blocks if try to read past acked data
Visualizing the window

Left side of window advances when data is acknowledged.
Right side controlled by size of window advertisement
Flow Control

What if sender process is faster than receiver process?
- Data builds up in receive window
- if data is acked, sender will send more!
- If data is not acked, sender will retransmit!

Sender must transmit data no faster than it can be consumed by the receiver
- Receiver might be a slow machine
- App might consume data slowly

Sender sliding window $\leq$ free receiver buffer
- Advertised window $=$ # of free bytes; if zero, stop
Sender and Receiver Buffering

Sending application

LastByteWritten

LastByteAcked

LastByteSent

TCP

Receiving application

LastByteRead

NextByteExpected

LastByteRcvd

= available buffer

= buffer in use
Example – Exchange of Packets

Receiver has buffer of size 4 and application doesn’t read

Stall due to flow control here
Example – Buffer at Sender

\[ \begin{align*}
T=1 & : 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \\
T=2 & : 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \\
T=3 & : 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \\
T=4 & : 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \\
T=5 & : 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \\
T=6 & : 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9
\end{align*} \]

=acked

=sent

=advertised

\[ \]
How does sender know when to resume sending?

If receive window = 0, sender stops
  - no data => no acks => no window updates
Sender periodically pings receiver with one byte packet
  - receiver acks with current window size
Why not have receiver ping sender?
Should sender be greedy (I)?

Should sender transmit as soon as any space opens in receive window?
  - Silly window syndrome
    - receive window opens a few bytes
    - sender transmits little packet
    - receive window closes

Solution (Clark, 1982): sender doesn’t resume sending until window is half open
Should sender be greedy (II)?

App writes a few bytes; send a packet?
- Don’t want to send a packet for every keystroke
- If buffered writes >= max segment size
- if app says “push” (ex: telnet, on carriage return)
- after timeout (ex: 0.5 sec)

Nagle’s algorithm
- Never send two partial segments; wait for first to be acked, before sending next
- Self-adaptive: can send lots of tinygrams if network is being responsive

But (!) poor interaction with delayed acks (later)
TCP Connection Management

Setup
- assymetric 3-way handshake

Transfer
- sliding window; data and acks in both directions

Teardown
- symmetric 2-way handshake

Client-server model
- initiator (client) contacts server
- listener (server) responds, provides service
Three-Way Handshake

Opens both directions for transfer

Active participant
(client)

Passive participant
(server)

SYN, SequenceNum = x

SYN + ACK, SequenceNum = y,
Acknowledgment = x + 1

ACK, Acknowledgment = y + 1

+data
Do we need 3-way handshake?

Allows both sides to
- allocate state for buffer size, state variables, ...
- calculate estimated RTT, estimated MTU, etc.

Helps prevent
- Duplicates across incarnations
- Intentional hijacking
  - random nonces => weak form of authentication

Short-circuit?
- Persistent connections in HTTP (keep connection open)
- Transactional TCP (save seq #, reuse on reopen)
- But congestion control effects dominate
TCP Transfer

Connection is bi-directional

- acks can carry response data

\[
\begin{align*}
\text{Seq} &= x + \text{MSS}; \quad \text{Ack} = y + 1 \\
\text{Seq} &= x + 2 \times \text{MSS}; \quad \text{Ack} = y + 1 \\
\text{Seq} &= y + \text{MSS}; \quad \text{Ack} = x + 2 \times \text{MSS} + 1 \\
\text{Seq} &= x + 3 \times \text{MSS}; \quad \text{Ack} = y + \text{MSS} + 1
\end{align*}
\]
TCP Connection Teardown

Symmetric: either side can close connection (or RST!)

Can reclaim connection after 2 MSL

Can reclaim connection right away (must be at least 1MSL after first FIN)
TCP State Transitions
TCP Connection Setup, with States

Active participant (client)
- SYN_SENT
- ESTABLISHED

Passive participant (server)
- LISTEN
- SYN_RCVD
- ESTABLISHED

SYN, SequenceNum = x

SYN + ACK, SequenceNum = y,
Acknowledgment = x + 1

ACK, Acknowledgment = y + 1

+data
TCP Connection Teardown

Web server

<table>
<thead>
<tr>
<th>State</th>
<th>Web browser</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIN_WAIT_1</td>
<td>FIN_WAIT_1</td>
</tr>
<tr>
<td>FIN_WAIT_2</td>
<td>FIN_WAIT_2</td>
</tr>
<tr>
<td>TIME_WAIT</td>
<td>TIME_WAIT</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>CLOSED</td>
<td>CLOSED</td>
</tr>
<tr>
<td>FIN</td>
<td>FIN</td>
</tr>
<tr>
<td>ACK</td>
<td>ACK</td>
</tr>
<tr>
<td>LAST_ACK</td>
<td>LAST_ACK</td>
</tr>
<tr>
<td>CLOSE_WAIT</td>
<td>CLOSE_WAIT</td>
</tr>
<tr>
<td>CLOSED</td>
<td>CLOSED</td>
</tr>
</tbody>
</table>
The TIME_WAIT State

We wait 2MSL (two times the maximum segment lifetime of 60 seconds) before completing the close

Why?

ACK might have been lost and so FIN will be resent
Could interfere with a subsequent connection
TCP Handshake in an Uncooperative Internet

TCP Hijacking
- if seq # is predictable, attacker can insert packets into TCP stream
- many implementations of TCP simply bumped previous seq # by 1
- attacker can learn seq # by setting up a connection

Solution: use random initial sequence #'s
- weak form of authentication
TCP Handshake in an Uncooperative Internet

TCP SYN flood
- server maintains state for every open connection
- if attacker spoofs source addresses, can cause server to open lots of connections
- eventually, server runs out of memory

Malicious attacker

Server

SYN, SequenceNum = x
SYN, p
SYN, q
SYN, r
SYN, s
SYN + ACK, y, x + 1

TCP SYN cookies

Solution: SYN cookies
- Server keeps no state in response to SYN; instead makes client store state
- Server picks return seq # \( y = \odot \) that encrypts \( x \)
- Gets \( \odot + 1 \) from sender; unpacks to yield \( x \)

Can data arrive before ACK?
How can TCP choose segment size?

Pick LAN MTU as segment size?
- LAN MTU can be larger than WAN MTU
- E.g., Gigabit Ethernet jumbo frames

Pick smallest MTU across all networks in Internet?
- Most traffic is local!
  - Local file server, web proxy, DNS cache, ...
  - Increases packet processing overhead

Discover MTU to each destination? (IP DF bit)
Guess?
Layering Revisited

IP layer “transparent” packet delivery
- Implementation decisions affect higher layers (and vice versa)
  - Fragmentation => reassembly overhead
    - path MTU discovery
  - Packet loss => congestion or lossy link?
    - link layer retransmission
  - Reordering => packet loss or multipath?
    - router hardware tries to keep packets in order
  - FIFO vs. active queue management
IP Packet Header Limitations

Fixed size fields in IPv4 packet header
- source/destination address (32 bits)
  - limits to ~ 4B unique public addresses; about 600M allocated
  - NATs map multiple hosts to single public address
- IP ID field (16 bits)
  - limits to 65K fragmented packets at once => 100MB in flight?
  - in practice, fewer than 1% of all packets fragment
- Type of service (8 bits)
  - unused until recently; used to express priorities
- TTL (8 bits)
  - limits max Internet path length to 255; typical max is 30
- Length (16 bits)
  - Much larger than most link layer MTU’s
TCP Packet Header Limitations

Fixed size fields in TCP packet header
- seq #/ack # -- 32 bits (can’t wrap within MSL)
  - T1 ~ 6.4 hours; OC-192 ~ 3.5 seconds
- source/destination port # -- 16 bits
  - limits # of connections between two machines (NATs)
  - ok to give each machine multiple IP addresses
- header length
  - limits # of options
- receive window size -- 16 bits (64KB)
  - rate = window size / delay
  - Ex: 100ms delay => rate ~ 5Mb/sec
  - RFC 1323: receive window scaling
  - Defaults still a performance problem
HTTP on TCP

How do we reduce the # of messages?

Delayed ack: wait for 200ms for reply or another pkt arrival

TCP RST from web server