TCP: Reliable, In-Order Delivery

EE 122: Intro to Communication Networks
Fall 2006 (MW 4-5:30 in Donner 155)

Vern Paxson
TAs: Dilip Antony Joseph and Sukun Kim
http://inst.eecs.berkeley.edu/~ee122/
Materials with thanks to Jennifer Rexford, Ion Stoica, and colleagues at Princeton and UC Berkeley

Today’s Lecture

• How does TCP achieve correct operation?
• Reliability in the face of IP’s meager “best effort” service
• 3-way handshake to establish connections
• 3-way or 4-way handshake to terminate conn.
• Retransmission to recover from loss
  – We’ll only look at timeout-based retransmission today
• State diagrams as a tool for understanding complex protocol operation

TCP Service Model

• Reliable, in-order, byte-stream delivery
  – and with good performance
• Challenges - the network can
  – drop packets
  – delay packets
  – deliver packets out-of-order
  – follows from possibility of arbitrary delay
  – replicate packets
  – weird, but it does sometimes happen
  – corrupt packets
  – (What’s missing?)

TCP Support for Reliable Delivery

• Checksum
  – Used to detect corrupted data at the receiver
  – ...leading the receiver to drop the packet
• Sequence numbers
  – Used to detect missing data
  – ... and for putting the data back in order
• Retransmission
  – Sender retransmits lost or corrupted data
  – Timeout based on estimates of round-trip time
  – Fast retransmit algorithm for rapid retransmission

TCP Header

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td>Acknowledgment</td>
</tr>
<tr>
<td>Flags</td>
<td>Advertised window</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options (variable)</td>
<td>Data</td>
</tr>
</tbody>
</table>
These should be familiar:

- Source port
- Destination port
- Sequence number
- Acknowledgment
- HdrLen
- Flags
- Advertised window
- Checksum
- Urgent pointer
- Options (variable)
- Data

Starting sequence number (byte offset) of data carried in this segment:

- Source port
- Destination port
- Sequence number
- Acknowledgment
- HdrLen
- Flags
- Advertised window
- Checksum
- Urgent pointer
- Options (variable)
- Data

Starting sequence number just beyond highest seq. received in order.

If sender sends N in-order bytes starting at seq S then ack for it will be S+N.

Number of 4-byte words in TCP header; 5 = no options

"Must Be Zero" 6 bits reserved

We will get to these shortly
TCP Header

Source port
Destination port
Sequence number
Acknowledgment
HdrLen
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

Buffer space available for receiving data. Used for TCP's sliding window. Interpreted as offset beyond Acknowledgment field's value.

TCP “Stream of Bytes” Service

Host A

Host B

Host A

Host B

TCP Segment

- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1,500 bytes on an Ethernet
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header ≥ 20 bytes long
- TCP segment
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream

TCP “Stream of Bytes” Service

Host A

Host B

Sequence Numbers

Host A

Host B

TCP Header

Source port
Destination port
Sequence number
Acknowledgment
HdrLen
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

TCP Segment

- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1,500 bytes on an Ethernet
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header ≥ 20 bytes long
- TCP segment
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream

TCP “Stream of Bytes” Service

Host A

Host B

Sequence Numbers

Host A

Host B
Initial Sequence Number (ISN)

- Sequence number for the very first byte
  - E.g., Why not just use ISN = 0?
- Practical issue
  - IP addresses and port #s uniquely identify a connection
  - Eventually, though, these port #s do get used again
  - … a chance an old packet is still in flight
  - … and might be associated with new connection
- TCP requires (RFC793) changing ISN over time
  - Set from 32-bit clock that ticks every 4 microseconds
  - … only wraps around once every 4.55 hours
- To establish a connection, hosts exchange ISNs

Connection Establishment: TCP’s Three-Way Handshake

Establishing a TCP Connection

- Three-way handshake to establish connection
  - Host A sends a SYN (open; “synchronize sequence numbers”) to host B
  - Host B returns a SYN acknowledgment (SYN ACK)
  - Host A sends an ACK to acknowledge the SYN ACK

TCP Header

Step 1: A’s Initial SYN Packet

- Flags: SYN
- A’s port
- B’s port
- A’s Initial Sequence Number
- (Irrelevant since ACK not set)
- Advertised window
- Checksum
- Urgent pointer
- Options (variable)

A tells B it wants to open a connection...

Step 2: B’s SYN-ACK Packet

- Flags: SYN, ACK
- B’s port
- A’s port
- B’s Initial Sequence Number
- ACK = A’s ISN plus 1
- Advertised window
- Checksum
- Urgent pointer
- Options (variable)

B tells A it accepts, and is ready to hear the next byte...

… upon receiving this packet, A can start sending data
Step 3: A's ACK of the SYN-ACK

<table>
<thead>
<tr>
<th>Flags:</th>
<th>SYN</th>
<th>FIN</th>
<th>RST</th>
<th>PSH</th>
<th>URG</th>
</tr>
</thead>
<tbody>
<tr>
<td>A's port</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B's port</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A's Initial Sequence Number</td>
<td>B's ISN plus 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20B</td>
<td>Flags</td>
<td>Advertised window</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (available)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A tells B it’s likewise okay to start sending

... upon receiving this packet, B can start sending data

Timing Diagram: 3-Way Handshaking

Active
Open

Server

Client (initiator)

listen()

connect()

SYN, SeqNum = x

SYN + ACK, SeqNum = y, Ack = x + 1

ACK, Ack = y + 1

Passive

Open

What if the SYN Packet Gets Lost?

• Suppose the SYN packet gets lost
  – Packet is lost inside the network, or:
    – Server discards the packet (e.g., listen queue is full)

• Eventually, no SYN-ACK arrives
  – Sender sets a timer and waits for the SYN-ACK
  – … and retransmits the SYN if needed

• How should the TCP sender set the timer?
  – Sender has no idea how far away the receiver is
  – Hard to guess a reasonable length of time to wait
  – SHOULD (RFCs 1122 & 2988) use default of 3 seconds
    • Other implementations instead use 6 seconds

SYN Loss and Web Downloads

• User clicks on a hypertext link
  – Browser creates a socket and does a "connect"
  – The "connect" triggers the OS to transmit a SYN

• If the SYN is lost...
  – 3-6 seconds of delay: can be very long
  – User may become impatient
  – … and click the hyperlink again, or click "reload"

• User triggers an “abort” of the “connect”
  – Browser creates a new socket and another “connect”
  – Essentially, forces a faster send of a new SYN packet!
  – Sometimes very effective, and the page comes quickly

5 Minute Break

Tearing Down the Connection

Questions Before We Proceed?
Normal Termination, One Side At A Time

- Finish (FIN) to close and receive remaining bytes
  - FIN occupies one octet in the sequence space
- Other host ack’s the octet to confirm
- Closes A’s side of the connection, but not B’s
  - Until B likewise sends a FIN
  - Which A then acks

Normal Termination, Both Together

- Same as before, but B sets FIN with their ack of A’s FIN

Sending/Receiving the FIN Packet

- Sending a FIN: close()
  - Process has finished sending data via the socket
  - Process calls “close()” to close the socket
  - Once TCP has sent all of the outstanding bytes...
  - ... then TCP sends a FIN
    - Even if bytes not yet ack’d
    - Because FIN has seqno beyond all the bytes ...
    - ... and thus won’t be ack’d until all bytes are delivered

- Receiving a FIN: EOF
  - Process is reading data from the socket
  - Eventually, the attempt to read returns an EOF
  - All bytes prior to sender-calling close() have been delivered

Abrupt Termination

- A sends a RESET (RST) to B
  - E.g., because app. process on A crashed
- That’s it
  - B does not ack the RST
  - Thus, RST is not delivered reliably
  - And: any data in flight is lost
  - But: if B sends anything more, will elicit another RST

Reliability: TCP Retransmission

- Reasons for Retransmission
  - Packet lost
  - ACK lost
  - Early timeout

Packet
Packet
Packet
Packet
ACK
ACK
ACK
ACK
How Long Should Sender Wait?

- Sender sets a timeout to wait for an ACK
  - Too short: wasted retransmissions
  - Too long: excessive delays when packet lost
- TCP sets retransmission timeout (RTO) as function of RTT
  - Expect ACK to arrive an RTT after data sent
  - ... plus slop to allow for variations (e.g., queuing, MAC)
- But: how does the sender know the RTT?
- And: what’s a good estimate for “slop”?

Jacobson/Karels Algorithm

- Compute “slop” in terms of observed variability
  - One solution: use standard deviation (requires expensive square root computation)
  - Use mean deviation instead
  
  \[
  \text{Difference} = \text{SampleRTT} - \text{EstimatedRTT} \\
  \text{Deviation} = \text{Deviation} + \delta \times (| \text{Difference} | - \text{Deviation}) \\
  \text{RTO} = \mu \times \text{EstimatedRTT} + \phi \times \text{Deviation}
  \]
  
  \[\delta = 1/4 \text{ (again, for one measurement per flight)}\]
  \[\mu = 1\]
  \[\phi = 4\]
  
  - Implementations often use a coarse-grained (500 msec) timer, so resulting value is large

Karn/Partridge Algorithm

- Measure SampleRTT only for original transmissions
  - Once a segment has been retransmitted, do not use it for any further measurements
- Also, employ exponential backoff
  - Every time RTO timer expires, set RTO ← 2·RTO
  - (Up to maximum ≥ 60 sec)
  - Every time new measurement comes in (= successful original transmission), collapse RTO back to computed value

Problem: Ambiguous Measurement

- How to differentiate between the real ACK, and ACK of the retransmitted packet?

State Diagrams

- For complicated protocols, operation depends critically on current mode of operation
- Important tool for capture this: state diagram
- At any given time, protocol endpoint is in a particular state
  - Dictates its current behavior
- Endpoint transitions to other states on events
  - Interaction with lower layer
    - Reception of certain types of packets
    - Interaction with upper layer
    - New data arrives to send, or received data is consumed
  - Timers
Summary

- Reliable, in-order, byte-stream delivery
  - Sequence numbers
  - Acknowledgments
  - 3-way handshake to establish
  - 3-way or 4-way handshake to terminate
  - Timer-based retransmission
  - State diagram to keep it all straight

- What's missing?
  - Performance

- Next lecture
  - Congestion control