Last Lecture: TCP

1. *Multiplexing and Demultiplexing ✔*
2. *Byte-stream service ✔*
   - Stream of bytes sent and received, not stream of packets
3. *Reliable data transfer ✔*
   - A combination of go-back-N and selective repeat, and performance tuning heuristics
4. *Connection management*
   - Connection establishment and tear down
5. *Flow control*
   - Prevent sender from overflowing receiver
6. *Congestion control* (later)
This Lecture: TCP

1. *Multiplexing and Demultiplexing*

2. *Byte-stream service*
   - Stream of bytes sent and received, not stream of packets

3. *Reliable data transfer*
   - A combination of go-back-N and selective repeat, and performance tuning heuristics

4. *Connection management ✔*
   - Connection establishment and tear down

5. *Flow control ✔*
   - Prevent sender from overflowing receiver

6. *Congestion control* (later)
TCP Segment Structure

<table>
<thead>
<tr>
<th>Source port #</th>
<th>Dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgement number</td>
<td></td>
</tr>
<tr>
<td>Receive window</td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td></td>
</tr>
<tr>
<td>Urg data pointer</td>
<td></td>
</tr>
<tr>
<td>Options (variable length)</td>
<td></td>
</tr>
<tr>
<td>Application data (variable length)</td>
<td></td>
</tr>
</tbody>
</table>

- **URG**: urgent data (generally not used)
- **ACK**: ACK # valid
- **PSH**: push data now (generally not used)
- **RST, SYN, FIN**: connection estab (setup, teardown commands)
- **Internet checksum** (as in UDP)

- **Counting by bytes of data (not segments!)**
- **# bytes rcvr willing to accept**
TCP Options

Options is a list of options, in one of two formats:
- (kind) [1 byte]
- (kind, length, data) [1 byte, 1 byte, N bytes]
  - length counts all bytes in the option

List of common options:

<table>
<thead>
<tr>
<th>Kind</th>
<th>Length</th>
<th>Meaning</th>
<th>RFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>End of option list</td>
<td>793</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>No Operation, for padding</td>
<td>793</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>MSS</td>
<td>793</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Window Scale</td>
<td>1323</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>SACK permitted</td>
<td>2018</td>
</tr>
<tr>
<td>5</td>
<td>N</td>
<td>SACK</td>
<td>2018</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>Timestamp option</td>
<td>1323</td>
</tr>
</tbody>
</table>
4. TCP Connection Management

- **Connection establishment**
  - Allow each end to know the other exists
    - Trigger allocation of transport entity resources
    - Buffer
    - Timers (if any), ...
  - Set up optional parameters
    - Max segment size (MSS)
    - Initial Sequence Numbers (ISN)
    - Window size, ...

- **Connection termination**
  - Tell the other end you’re done
  - Clean up after yourself (e.g., wait for delayed duplicates to die)
Three-way handshake to establish connection

1. Host A sends a **SYN** (open) to the host B
2. Host B returns a SYN acknowledgment (**SYN ACK**)
3. Host A sends an **ACK** to acknowledge the SYN ACK

Each side tells its ISN to the other side.
Step 1: A’s Initial SYN Segment

Flags: SYN, FIN, RST, PSH, URG, ACK

A’s port | B’s port
----------|----------

A’s Initial Sequence Number

Acknowledgment

24 | 0 | Flags
Advertised window

Checksum

Urgent pointer

Options (variable) (MSS here)

A tells B it wants to open a connection…
Step 2: B’s SYN/ACK Segment

Flags: SYN, FIN, RST, PSH, URG, ACK

B’s port | A’s port
---|---

B’s Initial Sequence Number | A’s ISN plus 1

20 | 0 | Flags | 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 Acknowledges the SYN/ACK

A tells B it is okay to start sending

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

Flags: SYN, FIN, RST, PSH, URG, ACK

<table>
<thead>
<tr>
<th>A’s port</th>
<th>B’s port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td>B’s ISN plus 1</td>
</tr>
<tr>
<td>20</td>
<td>0 Flags</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
</tbody>
</table>
Timeout for SYN Retransmission

On BSD and the likes:

- 6 seconds after the first SYN
- 24 seconds after the second SYN
- 48 seconds after the third SYN
- give up

- Most Berkeley-derived OS have an upper limit of 75 seconds
SYN Loss and Web Download

- 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...
  - The 6 seconds of delay may be very long
  - The user may get impatient
  - ... and click the hyperlink again, or click “reload”
- “Reload” triggers an “abort” of the “connect”
  - Browser creates a new socket and does a “connect”
  - Essentially, forces a faster send of a new SYN packet!
  - Sometimes very effective, and the page comes fast
Tips and Tricks

- **The Morris attack (1985)**
  - Robert H. Morris is the father of the other Morris
  - He worked for Bell Labs, then chief scientist at NSA

- Up to the early 90’s, ISN is chosen sort of like this
  - RFC 793 says: “counter++ every 4μs”, use counter for ISN
  - Berkeley-derived kernels: “counter += C every second, and += D for every new connection”, C&D are constants

- To attack server S who trusts host A (rlogin/rsh)
  - Wait for A to be turned off (or DoS it)
  - Spoof a SYN from A, ignore the SYN/ACK from S
  - Send final ACK from A with correct ISN + 1 (how?)
  - Send commands to server S
Security Issue: SYN Flooding

- The attack:
  - IP-spoof a SYN packet, send it to server.
  - Server sends back SYN-ACK, wait for connection timeout (typically 75 seconds)
  - Thousands of SYN packets can eat up server’s resources and new requests can’t be granted

- No “best” solution
  - Routers can reduce IP-spoofed packets
  - Routers (Cisco & others) have the “TCP intercept” mode
  - SYN cookies, SYN cache, SYN proxying, SYNkill, etc.
  - (Some defenses subject to the attack themselves!!!
History of SYN Flooding

- Discovered in 1994 (Bill Cheswick, Bellovin)
  - “Firewalls and Internet Security: Repelling the Wily Hacker”
  - No countermeasure developed in next 2 years

- Description and exploit tool: *Phrack P48-13* (1996)

- Sep 1996, SYN Flooding attacks seen in the wild
  - CERT Advisory released

- Remedies quickly developed (partial solution)
  - Some made their ways to OS codes
SYN Flooding – Some Technical Details

- Implementation dependent
  - Linux kernel 2.6.10: 1300-byte “sock” structure per SYN
  - Other OS: at least 280 bytes
  - The “backlog” parameter of listen() has an effect on the queue size

- Defenses
  - Avoid IP-spoofing (more later): RFCs 2827, 3013, 3704
  - SYN Cache, SYN Cookies:
    - Drawback: can’t pigging back application data in SYN segment
    - Sometime disabled by default in implementations
    - Most BSD-derived OS implement one of these
    - Linux version > 2.6.5 does too
    - Windows 2K and later does too (modify some registry)
Connection Termination

- **Asymmetric release**: close the connection when one side asks for it
  - Abrupt and may result in data loss

- **Symmetric release**: two separate directions
  - FIN and ACK for each direction
  - Not an easy task.
  - What about a 3-way handshake?
Data Loss in Asymmetric Release

No data are delivered after a disconnect request
Symmetric Release is Hard Too
TCP’s Connection Termination

client

FIN

server

ACK

ACK

FIN

close
close
timed wait
closed
Normal Operations
TCP Allows Half-Close with Shutdown()
Simultaneous Close Allowed

state transitions in simultaneous close
BTW, Simultaneous Open is Possible Too

state transitions in simultaneous open
The Time_Wait (2MSL) state

- **MSL** stands for *Maximum Segment Lifetime*
  - Common implementations are either 30sec, 1min, 2min

- **Purposes of the 2MSL state**
  - Let TCP resend the final ACK if needed (when?)
  - The socket can only be reused after 2MSL (why?)
    - Sometime you can’t bind a server port because of this 2MSL state
    - However, setting socket option SO_REUSEADDR allows us to reuse the port number (violation of RFC)
    - But still, no two identical socket quadruples

- “*Quiet time*” (RFC 793):
  - no connection creation within 2MSL after crashing (why?)
RESET Segments

**RST** is used to

- Reply to connection requests to some port no-one is listening on
  - In UDP, an *ICMP port unreachable* is generated instead
- Reply to connection requests within 2MSL after crashing
- Abort an existing connection

Note: RST has its own sequence number
Crash Recovery

- After restart all state information is lost
- Connection is *half open*
  - Side that did not crash still thinks it is connected
- We should close connections using *keep-alive timer*
  - This is controversial: is TCP or application responsible?
  - *Implementation dependent*
- Crashed side (after reboot) sends *RST i* in response to any *segment i* arriving
- User must decide whether to reconnect
  - Problems with lost or duplicate data
Tips and Tricks

- **TCP Connection Killing**
  - Using RST
  - Using FIN
  - Again, just need to know the right sequence number
5. Flow Control in TCP

- **Flow Control:**
  - Avoid fast sender overflowing slow receiver

- **Basic Mechanism:**
  - Receiver advertises its available window size (FWind)
  - Sender ensures that $\text{LastByteSent} - \text{LastByteAcked} \leq \text{FWind}$
  - FWind is re-advertised in packets flowing back
TCP Flow Control: Sender Side

- Sent but not acked
- Sent and acked
- Not yet sent

FWind
Advertised by receiver

Window slides
As data is ACK’ed

LastByteACKed by receiver

LastByteSent

LastByteWritten
TCP Flow Control: Sender Side

Packet Sent

<table>
<thead>
<tr>
<th>Source Port</th>
<th>Dest. Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence Number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
</tr>
<tr>
<td>HL/Flags</td>
<td>Window</td>
</tr>
<tr>
<td>D. Checksum</td>
<td>Urgent Pointer</td>
</tr>
<tr>
<td>Options..</td>
<td></td>
</tr>
</tbody>
</table>

Packet Received

<table>
<thead>
<tr>
<th>Source Port</th>
<th>Dest. Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence Number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
</tr>
<tr>
<td>HL/Flags</td>
<td>FWind</td>
</tr>
<tr>
<td>D. Checksum</td>
<td>Urgent Pointer</td>
</tr>
<tr>
<td>Options..</td>
<td></td>
</tr>
</tbody>
</table>

App write

acknowledged sent to be sent outside window

Picture taken and modified from Shiv Kalyanaraman’s slides
TCP Flow Control: Receiver Side

Receive buffer

- Acked but not delivered to user
- Not yet acknowledged
- Not yet acknowledged

FWind

Picture taken and modified from Shiv Kalyanaraman’s slides
FWind Size In Practice

- Old implementations’ default: 4KB
- Newer implementations: up to 16KB
- How large should it be, suppose we have plenty of memory and receiver’s CPU is infinitely fast?

Recall the **bandwidth-delay product**:
- RTT x transmission rate
- For T1 link across US: 60ms x 1.544M bps → 11.58 KB
- For T3 link across US: 60ms x 45 Mbps → 337.5 KB
- Note: **337.5 KB >> 16-bit window size ≈ 65KB**
- For OC-12 link across US: 60ms x 622 Mbps → 4.7MB

Solution: use the Window Scale option
Technical Issues with Flow Control

A. **Deadlock**
   - Can deadlock occur with current flow control mechanism?

B. **Performance tuning for interactive data flow**
   - telnet, SSH, Rlogin, ..., 10% of TCP segments (with a few to tens of data bytes per segment)

C. **Performance tuning for bulk data flow**
   - FTP, Email, HTTP, ..., 90% of TCP segments (with hundred of data bytes)
A. Deadlock & TCP Persistence Timer

- To prevent deadlock, *persistence timer* is used to send *window probes*
  - Normal segment with just *one* byte of data (past current window)
  - Host required to respond to data sent past window

- Exponential back-off is used for persistence timer
  - Start with 1.5 seconds
  - Double every time up to 60 seconds
Tips and Tricks

- Talking about interactive data flows: how fast can people type?

- Guinness record is about 190 wpm (Natalie Lantos, 1999)
  - If each word has 5 letters on average, then it is about 950cpm or 15.8 characters per second.
  - If each word has 10 letters on average, then it’s still only about 31 characters per second! (or ~ 1 byte for each 32ms, twice longer than typical local RTT)
B. TCP Interactive Data Flow

- Data might be sent 1 byte at a time
- Heuristics to improve performance for interactive data flow?
  - *Delayed ACK* (200ms, or every other segment)
  - *Nagle Algorithm*: try to delay sending “small” segments until outstanding data is acknowledged or a full-sized segment is available
    - *This algorithm is self-clocking!*
  - In an Ethernet with RTT $\approx 16$ms, would Nagle algorithm have any effect for an interactive data flow?
  - Sometime Nagle needs to be turned off (e.g. for X-server, each mouse movement needs to be reported), using TCP_NODELAY socket option
Nagle’s Algorithm in More Details

Sender does not transmit unless one of the following conditions is true:

- a full-sized segment can be sent
- at least \( \frac{1}{2} \) of the maximum \( F_{\text{Window}} \) which has ever been advertised
- no outstanding unacknowledged data

What are the pros and cons of Nagle’s algorithm?
Silly Window Syndrome

- Receiver advertises small $FWind$ gradually
  - Suppose starting from $FWind=0$, application reads 1 byte of data at a time, slowly
  - Sender then would send a few bytes at a time, wasting lots of header overhead

- Symmetric to Nagle’s algorithm, we can impose the following rule (*David Clark’s algorithm*)
  - receiver should not advertise larger window than the current $FWind$ until $FWind$ can be increased by $\min(MSS, \frac{1}{2} \text{buffer space})$
C. Bulk Data Flow

- Sliding window with scale option
- Delayed ACK also helps