(Possible) Transport Protocol Functions

**Multiplexing/demultiplexing for multiple applications.**
- “Port” abstraction abstracts OS notions of “process”

**Connection establishment.**
- Logical end-to-end connection
- Connection state to optimize performance

**Error control.**
- Hide unreliability of the network layer from applications
- Many types of errors: corruption, loss, duplication, reordering.

**End-to-end flow control.**
- Avoid flooding the receiver

**[Congestion control.]**
- Avoid flooding the network
Outline

Connection establishment
  • Reminder

Error control, Flow control
  • Stop & Wait vs. sliding window (conceptual and TCP)
  • Ack flavors, windows, timeouts, sequence numbers

Connection teardown

Next Lecture – Wireless/Mobility

Monday – TCP again
  • Congestion control – you will not address in Project 3
Transmission Control Protocol (TCP)

Reliable bi-directional byte stream

Connections established & torn down
- Analogy: setting up & terminating phone call

Multiplexing/ demultiplexing
- Ports at both ends

Error control
- Users see correct, ordered byte sequences

End-end flow control
- Avoid overwhelming machines at each end

Congestion avoidance
- Avoid creating traffic jams within network
Establishing Connection

Three-Way Handshake

- Each side notifies other of starting sequence number it will use for sending
- Each side acknowledges other's sequence number
  - SYN-ACK: Acknowledge sequence number + 1
- "Piggy-back" second SYN with first ACK
Error Control – Threats

Network may corrupt frames

- Despite link-level checksum
- Despite switch/router memory ECC
- Example
  - Store packet headers in separate memory from packet bodies
  - Maintain association between header #343 and body #343
    - Most of the time...

Packet-sequencing issues

- Network may duplicate packets (really?)
- Network may re-order packets (why?)
- Network may lose packets (often, actually)
Error Control

Segment corruption problems
- Add end-to-end checksum to TCP segments
- Computed at sender
- Checked at receiver

Packet sequencing problems
- Include sequence number in each segment
  - Byte number of 1st data byte in segment
- Duplicate: ignore
- Reordered: re-reorder or drop
- Lost: retransmit
Error Control

Lost segments detected by sender.
- Receiver won't ACK a lost segment
- Sender can use timeout to detect lack of acknowledgment
- Setting timeout requires estimate of round-trip time

Retransmission requires sender to keep copy of data.
- Local copy is discarded when ACK is received
Error Control Algorithms

Use two basic techniques:

- Acknowledgements (ACKs)
- Timeouts

Two examples:

- Stop-and-wait
- Sliding window
Stop-and-Wait

Receiver: send an acknowledge (ACK) back to the sender upon receiving a packet (frame)

Sender: excepting first packet, send next packet only upon receiving the ACK for the current packet
**What Can Go Wrong?**

1. **Frame lost - resend it on timeout**
   - Sender: Frame
   - Receiver: Timeout
   - Sender: ACK
   - Receiver: Frame

2. **ACK lost - resend packet**
   - Sender: Frame
   - Receiver: Timeout
   - Sender: ACK
   - Receiver: Frame
   - Receiver: ACK
   - Receiver: Frame

3. **ACK delayed – resend packet**
   - Sender: Frame
   - Receiver: Timeout
   - Sender: ACK
   - Receiver: Frame
   - Receiver: ACK
   - Receiver: new frame
   - Receiver: stale ACK

Receiver must be able to detect this is duplicate, not the next packet.

Sender must be able to detect when an ACK is for an old data packet.
Stop & Wait Sequence Numbers

Need a way to detect stale packets
  • Stale data at receiver
  • Stale ACK at sender

TFTP stop&wait sequence numbers are conservative
  • Each packet, ACK is tagged with file position
  • This is overkill
    • Bounding packet lifetime in network allows smaller sequence numbers
    • Special case: point-to-point link, 1-bit sequences numbers
Stop-and-Wait Disadvantage

May lead to inefficient link utilization

Example

- One-way propagation = 15 ms
- Throughput = 100 Mbps
- Packet size = 1000 bytes: transmit = \( \frac{8 \times 1000}{10^8} = 0.08 \text{ms} \)
- Neglect queue delay: Latency = approx. 15 ms; RTT = 30 ms
Stop-and-Wait Disadvantage (cont'd)

Send a message every 30 ms

- Throughput = \( \frac{8 \times 1000}{0.03} = 0.2666 \text{ Mbps} \)

Thus, the protocol uses less than 0.3% of the link capacity!
Solution

Don’t wait for the ACK of the previous packet before sending the next packet!
Sliding Window Protocol: Sender

Each packet has a sequence number
- Assume infinite sequence numbers for simplicity

Sender maintains a window of sequence numbers
- SWS (sender window size) – maximum number of packets that can be sent without receiving an ACK
- LAR (last ACK received)
- LFS (last frame sent)

![Diagram showing acknowledged packets and packets not acknowledged yet with LAR and LFS markers](image-url)
Example

Assume SWS = 3

Note: usually ACK contains the sequence number of the first packet in sequence expected by receiver
Need for Receiver Window

Window size = 3 packets

Sender

Receiver

1 2 3 4

Time
Need for Receiver Window

Window size = 3 packets

Sender

1 2 3 4 5

Receiver

Time
Need for Receiver Window

Window size = 3 packets

Sender

1 2 3 4 5 6 7

Timeout Packet 5

Timeout

Packets Still Arriving

Receiver

Time

Packet 5

Still Arriving
Sliding Window Protocol: Receiver

Receiver maintains a window of sequence numbers

- **RWS** (receiver window size) – maximum number of out-of-sequence packets that can be received
- **LFR** (last frame received) – last frame received in sequence
- **LAF** (last acceptable frame)
- **LAF** = **LFR** ≤ **RWS**

**Note that this window is just for sliding-window**

- TCP “receiver window” has two purposes
- TCP also has a “congestion window”
  - Secret – does not appear in packet header
Sliding Window Protocol: Receiver

Let seqNum be the sequence number of arriving packet

If (seqNum <= LFR) or (seqNum >= LAF)
  • Discard packet

Else
  • Accept packet
  • ACK largest sequence number seqNumToAck, such that all packets with sequence numbers <= seqNumToAck were received

![Diagram of packets in sequence and out-of-sequence]
Window Flow Control: Send Side

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

- **Next to be sent**

**Must retain for possible retransmission**
Packet Reception

What should receiver do?

Receive buffer

- Acked but not delivered to user
- Not yet acked

Window

New
Maximum Window Size

Mechanism for receiver to exert flow control

- Prevent sender from overwhelming receiver's buffering & processing capacity
- Max. transmission rate = window size / round trip time

Receive buffer

- Acked but not delivered to user
- Not yet acked

window
Choices of Ack

**Cumulative ack**
- I have received 17..23
- I have [still] received 17..23

**Selective ack**
- I received 17-23, 25-27

**Negative ack**
- I think I'm missing 24...

Tradeoffs?
Window Size Too Small

Max Throughput = \frac{\text{Window Size}}{\text{Roundtrip Time}}
Adequate Window Size

Max Throughput = \frac{\text{Window Size}}{\text{Roundtrip Time}}
Timeout Value Selection

Long timeout?
Short timeout?
Solution?
Setting Retransmission Timeout (RTO)

Time between sending & resending segment

Challenge

- Too long: Add latency to communication when packets dropped
- Too short: Send too many duplicate packets
- General principle: Must be > 1 Round Trip Time (RTT)
Round-trip Time Estimation

Every Data/Ack pair gives new RTT estimate

Can Get Lots of Short-Term Fluctuations
Original TCP Round-trip Estimator

Round trip times exponentially averaged:

- New RTT = \( \alpha \) (old RTT) + (1 - \( \alpha \)) (new sample)
- Recommended value for \( \alpha \): 0.8 - 0.9
  - 0.875 for most TCP's

Retransmit timer set to \( \beta \) RTT, where \( \beta = 2 \)

- Want to be somewhat conservative about retransmitting
Karn/Partridge Algorithm

- Ignore sample for segment that has been retransmitted
- Use exponential backoff for retransmissions
  - Each time retransmit same segment, double the RTO
  - Based on premise that packet loss is caused by major congestion
Sequence Number Space

Each byte in byte stream is numbered.
- 32 bit value
- Wraps around
- Initial values selected at start up time

TCP breaks byte stream into packets (“segments”)
- Packet size is limited to the Maximum Segment Size

Each segment has a sequence number.
- Indicates where it fits in the byte stream
Finite Length Sequence Number

Sequence number can wrap around

• What is the problem?
• What is the solution?
  • Hint: not “crash the kernel”
  • Not even “crash the connection” or “connection full”
Sequence Numbers

32 Bits, unsigned

Circular Comparison, “b following a”

Why So Big?

- For sliding window, must have
  - $|\text{Sequence Space}| > |\text{Sending Window}| + |\text{Receiving Window}|$
  - No problem
- Also must guard against stray packets
  - With IP, packets have maximum lifetime of 120s
  - Sequence number would wrap around in this time at 286MB/s
Error Control Summary

Basic mechanisms

- CRC, checksum
- Timeout
- Acknowledgement
- Sequence numbers
- Window

Many variations and details
TCP Flow Control

Recall sliding-window as used for _error_ control
- For window size \( n \), can send up to \( n \) bytes without receiving an acknowledgement
- When the data are acknowledged then the window slides forward

Achieve _flow_ control via dynamically-sized window
- Sender naturally tracks outstanding packets versus max
  - Sending one packet decreases budget by one
- Receiver updates “open window” in every response
  - Packet \( B \rightarrow A \) contains \( \text{Ack}_A \) and \( \text{Window}_A \)
  - Sender can send bytes up through \( (\text{Ack}_A + \text{Window}_A) \)
  - Receiver can increase or decrease window at any time
- Original TCP always sent entire window
  - Congestion control now limits this
**Bidirectional Communication**

Each Side of Connection can Send *and* Receive

**What this Means**

- Maintain different sequence numbers for each direction
- Single segment can contain new data for one direction, plus acknowledgement for other
  - But some contain only data & others only acknowledgement
Ongoing Communication

Bidirectional Communication

- Each side acts as sender & receiver
- Every message contains acknowledgement of received sequence
  - Even if no new data have been received
- Every message advertises window size
  - Size of its receiving window
- Every message contains sent sequence number
  - Even if no new data being sent

When Does Sender Actually Send Message?

- When a maximal-sized segment worth of bytes is available
- When application tells it
  - Set PUSH flag for last segment sent
- When timer expires
Window Flow Control: Send Side

Host A $\rightarrow$ B

<table>
<thead>
<tr>
<th>Source Port</th>
<th>Dest. Port</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequence Number</strong></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>

Host B $\rightarrow$ A

<table>
<thead>
<tr>
<th>Source Port</th>
<th>Dest. Port</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequence Number</strong></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>

App write()
TCP Transmission

Client sends 796 bytes

Client sends 1260 more bytes

Server acknowledges 796 bytes
Tearing Down Connection

Either Side Can Initiate Tear Down

- Send FIN signal
  - “I'm not going to send any more data”

Other Side Can Continue Sending Data

- “Half-open connection”
- I must continue to acknowledge

Acknowledging FIN

- Acknowledge last sequence number + 1
TCP Connection Teardown Example

**Session**
- Echo client on 128.2.222.198, server on 128.2.210.194

**Client FIN**
- SeqC: 1489294581

**Server ACK + FIN**
- Ack: 1489294582 (= SeqC+1)
- SeqS: 1909787689

**Client ACK**
- Ack: 1909787690 (= SeqS+1)
State Diagram: Connection Tear-down

**Active Close**
- **FIN WAIT-1**
  - Send FIN
  - Ack
  - Rcv FIN
  - Send ACK
  - Rcv FIN+ACK
  - Send ACK
  - Rcv ACK of FIN
- **CLOSING**
  - Rcv FIN
  - Send ACK
  - Rcv FIN
- **TIME WAIT**
  - Rcv ACK of FIN
  - Timeout=2ms
  - Delete TCB
- **CLOSED**

**Passive Close**
- **CLOSE WAIT**
  - Send FIN
  - Rcv FIN
  - Send ACK
  - Rcv FIN
  - Rcv FIN+ACK
  - Send ACK
  - Rcv ACK of FIN
- **FIN WAIT**
  - Send FIN
  - Ack
  - Rcv FIN
  - Send ACK
  - Rcv FIN
  - Send ACK
  - Rcv FIN+ACK
  - Send ACK
  - Rcv ACK of FIN
- **CLOSING**
  - Rcv FIN
  - Send ACK
  - Rcv FIN
  - Send ACK
  - Rcv FIN+ACK
  - Send ACK
  - Rcv ACK of FIN
- **TIME WAIT**
  - Rcv ACK of FIN
  - Timeout=2ms
  - Delete TCB
- **CLOSED**
Key TCP Design Decisions

Connection Oriented
- Explicit setup & teardown of connections

Byte-stream oriented
- vs. message-oriented
- Sometimes awkward for application to infer message boundaries

Sliding Window with Cumulative Acknowledgement
- Single acknowledgement covers range of bytes
- Single missing message may trigger series of retransmissions

No Negative Acknowledgements
- Any problem with transmission must be detected by timeout
  - OK for IP to silently drop packets