Transport Layer IV

Dmitri Loguinov
Texas A&M University

March 7, 2019
Chapter 3: Roadmap

3.1 Transport-layer services
3.2 Multiplexing and demultiplexing
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer
3.5 Connection-oriented transport: TCP
   - Segment structure
   - Reliable data transfer
   - Flow control
   - Connection management
3.6 Principles of congestion control
3.7 TCP congestion control
TCP: Overview [RFCs: 793, 1122, 1323, 2001, 2018, 2581, 3390, 5681]

- Point-to-point (unicast):
  - One sender, one receiver
- Reliable, in-order *byte stream*:
  - Packet boundaries are not visible to the application
- 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 (excluding headers)
- Connection-oriented:
  - Handshaking (exchange of control msgs) initializes sender/receiver state before sending data

- Flow controlled:
  - Sender will not overwhelm receiver
Chapter 3: Roadmap

3.1 Transport-layer services
3.2 Multiplexing and demultiplexing
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer
3.5 Connection-oriented transport: TCP
   - Segment structure
   - Reliable data transfer
   - Flow control
   - Connection management
3.6 Principles of congestion control
3.7 TCP congestion control
TCP Segment Structure

- Sequence/ACK numbers
  - Count bytes, not segments
  - ACKs piggybacked on data packets
- Flags (U-A-P-R-S-F)
  - Urgent data (not used)
  - ACK field is valid
  - PUSH (not used)
  - RST (reset connection)
  - SYN (connection request)
  - FIN (connection close)
- Hdr length in DWORDs (4-bit field)
  - Normally 20 bytes, but longer if options are present

```
<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>hdr len</td>
<td>not used</td>
</tr>
<tr>
<td>checksum</td>
<td>Urg data pointer</td>
</tr>
<tr>
<td>Options (variable length)</td>
<td></td>
</tr>
<tr>
<td>application data (variable length)</td>
<td></td>
</tr>
</tbody>
</table>
```
TCP Seq. #’s and ACKs

Seq. #’s:
- Sequence number of the first byte in segment’s data

ACKs:
- Seq # of next byte expected from sender
- Cumulative ACK

Q: how receiver handles out-of-order segments?
A: TCP spec doesn’t say, up to implementor

Simple telnet scenario

User types ‘C’

Host A

Seq=42, ACK=79, data = ‘C’

host ACKs receipt of ‘C’, echoes back ‘C’

host ACKs receipt of echoed ‘C’

Seq=79, ACK=43, data = ‘C’

Seq=43, ACK=80

Host B
**TCP Round Trip Time and Timeout**

**Q:** how to set TCP timeout value (RTO)?
- Want it slightly larger than the next RTT
  - But the RTT varies
- **Too short:** premature timeout
  - Unnecessary retransmissions
- **Too long:** slow reaction to segment loss
  - Protocol may stall often, exhibit low performance

**Idea:** dynamically measure RTT, average these samples, then add safety margin
- **SampleRTT:** measured time from segment transmission until ACK receipt
  - Ignore retransmissions, why?
- **SampleRTT will vary,** want estimated RTT “smoother”
  - Average several recent measurements, not just current SampleRTT
TCP Round Trip Time and Timeout

EstimatedRTT(n) = (1−α)*EstimatedRTT(n-1) + α*SampleRTT(n)

- Exponentially weighted moving average (EWMA)
  - Influence of past sample decreases exponentially fast
  - Typical value: α = 0.125 = 1/8

- Task: derive a non-recursive formula for EstimatedRTT(n)
  - Assume EstimatedRTT(0) = SampleRTT(0)
  - Let Y(n) = EstimatedRTT(n) and y(n) = SampleRTT(n)

\[
Y(n) = (1 − α)^n y(0) + α \sum_{i=0}^{n-1} (1 − α)^i y(n − i)
\]
Example RTT Estimation:

![Graph showing RTT estimation over sample numbers](image)

- **RTT (ms)**
- **Sample number**
- **Sampled RTT**
- **Estimated RTT**
TCP Round Trip Time and Timeout

- **Setting the timeout:**
  - EstimatedRTT plus a “safety margin”
    - Larger variation in EstimatedRTT → larger safety margin
  - First estimate how much SampleRTT deviates from EstimatedRTT (typically, $\beta = 0.25$):

\[
\text{DevRTT}(n) = (1-\beta)\times\text{DevRTT}(n-1) \\
+ \beta \times |\text{SampleRTT}(n) - \text{EstimatedRTT}(n)|
\]

Then set retransmission timeout (RTO):

\[
\text{RTO}(n) = \text{EstimatedRTT}(n) + 4\times\text{DevRTT}(n)
\]
Example Timeout Estimation:

![Graph showing example timeout estimation with sample number on the x-axis and RTT (ms) on the y-axis. The graph includes lines for sampled RTT, estimated RTT, and timeout.](image)

- **Sample Number**: 0 to 60
- **RTT (ms)**: 0 to 400

The graph illustrates how timeout estimation changes with varying sample numbers, showing the estimated RTT line closely tracking the sampled RTT line with occasional fluctuations representing the timeout operations.
Chapter 3: Roadmap

3.1 Transport-layer services
3.2 Multiplexing and demultiplexing
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer
3.5 Connection-oriented transport: TCP
   - Segment structure
   - Reliable data transfer
   - Flow control
   - Connection management
3.6 Principles of congestion control
3.7 TCP congestion control
TCP Reliable Data Transfer

- TCP creates rdt service on top of IP’s unreliable service
  - Hybrid of Go-back-N and Selective Repeat
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer
  - For the oldest unACK’ed packet

- Retransmissions are triggered by:
  - Timeout events
  - Duplicate acks
- Initially consider simplified TCP sender:
  - Ignore duplicate acks
  - Ignore flow control, congestion control
NextSeqNum = InitialSeqNum // random for each transfer
SendBase = InitialSeqNum

loop (forever) {
    switch(event) {
        (a) data received from application above (assuming it fits into window):
            create TCP segment with sequence number NextSeqNum
            if (timer currently not running)
                start timer
            pass segment to IP
            NextSeqNum = NextSeqNum + length(data)
        (b) timeout:
            retransmit pending segment with smallest sequence number (i.e., SendBase); restart timer
        (c) ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase = y
                if (there are currently not-yet-acknowledged segments)
                    restart timer with latest RTO
                else cancel timer
            }
    } /* end of loop forever */
TCP Seq. #'S and ACKs

FTP Example:
• Suppose MSS = 1,000 bytes and the sender has a large file to transmit (we ignore seq field in ACKs and ACK field in data pkts)

What is the sender window size?

Host A

seq = 0
seq = 1000
seq = 2000
seq = 3000
seq = 4000
seq = 2000

RTO

Host B

ACK = 1000
ACK = 2000
ACK = 2000
ACK = 2000
ACK = 2000
ACK = 5000
TCP ACK Generation [RFC 1122, RFC 2581]

- Receiver immediately ACKs the base of its window in all cases except Nagle’s algorithm:
  - For in-order arrival of packets, send ACKs for every pair of segments; if second segment of a pair not received in 500ms, ACK the first one alone

```
seq = 0
seq = 1000
seq = 2000
seq = 3000
seq = 4000
RTO
seq = 2000
```

```
seq = 0
seq = 1000
seq = 2000
seq = 2000
seq = 3000
500 ms
```

```
ACK = 2000
ACK = 2000
ACK = 2000
ACK = 2000
RTO
ACK = 5000
```

```
seq = 0
seq = 1000
seq = 2000
seq = 2000
seq = 3000
```

```
ACK = 1000
ACK = 1000
ACK = 1000
ACK = 1000
```

```
Fast Retransmit

- Time-out period often relatively long
  - Especially in the beginning of transfer (3 seconds in RFC 1122)
- Idea: infer loss via duplicate ACKs
  - Sender often sends many segments back-to-back
  - If a segment is lost, there will be many duplicate ACKs
- If sender receives 3 duplicate ACKs for its base, it assumes this packet was lost
  - Fast Retransmit: resend the base segment immediately (i.e., without waiting for RTO)
- Note that reordering may trigger unnecessary retransmission
  - To combat this problem, modern routers avoid load-balancing packets of same flow along multiple paths
(c) event: ACK received, with ACK field value of y
   if (y > SendBase) {
       SendBase = y; dupACK = 0;
       if (SendBase != NextSeqNum)
           restart timer with latest RTO;
       else
           cancel timer; // last pkt in window
   }
else if (y == SendBase) {
    dupACK++;
    if (dupACK == 3)
      { resend segment with sequence y; restart timer}
}

a duplicate ACK for already ACKed segment

Fast Retransmit Algorithm:

Fast Retransmit Algorithm: a duplicate ACK for already ACKed segment
Chapter 3: Roadmap

3.1 Transport-layer services
3.2 Multiplexing and demultiplexing
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer
3.5 Connection-oriented transport: TCP
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management
3.6 Principles of congestion control
3.7 TCP congestion control
TCP Flow Control

- Assume packets received without loss, but the application does not call recv()
  - How to prevent sender from overflowing TCP buffer?
- Speed-matching service: sender rate to suit the receiving app’s ability to process incoming data

Flow control
Sender won’t overflow receiver buffer by transmitting too much, too fast

![Diagram of TCP flow control](image-url)
TCP Flow Control: How It Works

• **Spare room in buffer**
  
  \[ RcvWin = RcvBuffer - (LastByteReceivedInOrder - LastByteDelivered) \]

  - last ACK-1 went to application

• **Combining both constraints (sender, receiver):**
  
  \[ seq < \min(sndBase+sndWin, ACK+RcvWin) \]

• Receiver advertises spare room by including value of \( RcvWin \) in segments

• Sender enforces
  
  \[ seq < ACK + RcvWin \]
  
  - Guarantees receiver buffer doesn’t overflow
Chapter 3: Roadmap

3.1 Transport-layer services
3.2 Multiplexing and demultiplexing
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer
3.5 Connection-oriented transport: TCP
   - Segment structure
   - Reliable data transfer
   - Flow control
   - Connection management
3.6 Principles of congestion control
3.7 TCP congestion control
TCP Connection Management

- Purpose of connection establishment:
  - Exchange initial seq #s
  - Exchange flow control info (i.e., RcvWin)
  - Negotiate options (SACK, large windows, etc.)

Three way handshake:

- **Step 1:** client sends TCP SYN to server
  - Specifies initial seq # X and buffer size RcvWin
  - No data, ACK = 0

- **Step 2:** server gets SYN, replies with SYN+ACK
  - Sends server initial seq # Y and buffer size RcvWin
  - No data, ACK = X+1

- **Step 3:** client receives SYN+ACK, replies with ACK segment
  - Seq = X+1, ACK = Y+1
  - May contain regular data, but many servers will break

- **Step 4:** regular packets
  - Seq = X+1, ACK = Y+1
TCP Connection Management (Cont.)

Closing a connection:

- Closing a socket:
  ```
  closesocket(sock);
  ```

**Step 1:** originator end
system sends TCP FIN control segment to server

**Step 2:** other side
receives FIN, replies with ACK. Connection in “closing” state, sends FIN

TCP initiates a close when it has all ACKs for the transmitted data
**TCP Connection Management (Cont.)**

**Step 3:** originator receives FIN, replies with ACK
- Enters “timed wait” - will respond with ACK to received FINs

**Step 4:** other side receives ACK; its connection considered closed

**Step 5:** after a timeout (TIME_WAIT state lasts 240 seconds), originator’s connection is closed as well

*birectional transfer means both sides must agree to close*