Ders konuları

- Connection Oriented Transport : TCP
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management
**TCP: Overview**

- **point-to-point:** one sender, one receiver
- **reliable, in-order byte stream:** no “message boundaries”
- **full duplex data:** bi-directional data flow in same connection
- **MSS: maximum segment size**
- **connection-oriented:** handshaking (exchange of control msgs) init’s sender, receiver state before data exchange
- **flow controlled:** sender will not overwhelm receiver
- **pipelined:** TCP congestion and flow control set window size
- **send & receive buffers**

---

**Ders konuları**

- **Connection Oriented Transport : TCP**
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management
TCP segment structure

- **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)

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
<th>sequence number</th>
<th>acknowledgement number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Options (variable length)**

- **application data (variable length)**

- **Receive window**
- **Urg data pnter**

Counting by bytes of data (not segments!)

# bytes rcvr willing to accept

TCP seq. #’s and ACKs

**Seq. #’s:**
- byte stream “number” of first byte in segment’s data

**ACKs:**
- seq # of next byte expected from other side
- cumulative ACK

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

User types ‘C’

Host A

Seq=42, ACK=79, data = ‘C’

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

Host B

Seq=79, ACK=43, data = ‘C’

Seq=43, ACK=80

time

simple telnet scenario
TCP Round Trip Time and Timeout

**Q:** how to set TCP timeout value?
- longer than RTT
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

**Q:** how to estimate RTT?
- SampleRTT: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- SampleRTT will vary, want estimated RTT “smoother”
  - average several recent measurements, not just current SampleRTT

\[ \text{EstimatedRTT} = (1 - \alpha) \cdot \text{EstimatedRTT} + \alpha \cdot \text{SampleRTT} \]

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: \( \alpha = 0.125 \)
Example RTT estimation:

TCP Round Trip Time and Timeout

Setting the timeout

- EstimatedRTT plus “safety margin”
  - large variation in EstimatedRTT -> larger safety margin
  - first estimate of how much SampleRTT deviates from EstimatedRTT:

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

(typically, $\beta = 0.25$)

Then set timeout interval:

\[
\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT}
\]
Ders konuları

- Connection Oriented Transport : TCP
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management

TCP reliable data transfer

- TCP creates rdt service on top of IP’s unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer

- Retransmissions are triggered by:
  - timeout events
  - duplicate acks
- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control
TCP sender events:

**data rcvd from app:**
- Create segment with seq #
  - seq # is byte-stream number of first data byte in segment
- start timer if not already running (think of timer as for oldest unacked segment)
- expiration interval: TimeOutInterval

**timeout:**
- retransmit segment that caused timeout
- restart timer

**Ack rcvd:**
- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments

---

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

```c
loop (forever) {
    switch(event)
    event: data received from application above
        create TCP segment with sequence number NextSeqNum
        if (timer currently not running)
            start timer
        pass segment to IP
        NextSeqNum = NextSeqNum + length(data)
    event: timer timeout
        retransmit not-yet-acknowledged segment with smallest sequence number
        start timer
    event: ACK received, with ACK field value of y
        if (y > SendBase) {
            SendBase = y
            if (there are currently not-yet-acknowledged segments)
                start timer
        }
} /* end of loop forever */
```

Comment:
- SendBase-1: last cumulatively acked byte
  - Example:
    - SendBase - 1 = 71
    - y = 73, so the rcvr wants 73+
    - y > SendBase, so that new data is acked
TCP: retransmission scenarios

Host A
Seq=92, 8 bytes data
ACK=100

Host B
Seq=92, 8 bytes data
ACK=100

SendBase = 100

lost ACK scenario

A premature timeout

TCP retransmission scenarios (more)

Host A
Seq=92, 8 bytes data
ACK=100

Host B
Seq=92, 8 bytes data
ACK=100

SendBase = 120

Cumulative ACK scenario

SendBase = 120

SendBase = 100

SendBase = 120
### TCP ACK generation [RFC 1122, RFC 2581]

<table>
<thead>
<tr>
<th>Event at Receiver</th>
<th>TCP Receiver action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed</td>
<td>Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>Arrival of in-order segment with expected seq #: One other segment has ACK pending</td>
<td>Immediately send single cumulative ACK, ACKing both in-order segments</td>
</tr>
<tr>
<td>Arrival of out-of-order segment higher-than-expect seq #. Gap detected</td>
<td>Immediately send duplicate ACK, indicating seq# of next expected byte</td>
</tr>
<tr>
<td>Arrival of segment that partially or completely fills gap</td>
<td>Immediate send ACK, provided that segment starts at lower end of gap</td>
</tr>
</tbody>
</table>

### Fast Retransmit

- **Time-out period often relatively long:**
  - long delay before resending lost packet
- **Detect lost segments via duplicate ACKs.**
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.
- **If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:**
  - fast retransmit: resend segment before timer expires
Fast retransmit algorithm:

- Event: ACK received, with ACK field value of \( y \)
  - if (\( y > \text{SendBase} \))
    - \( \text{SendBase} = y \)
    - if (there are currently not-yet-acknowledged segments)
      - start timer
    - else
      - increment count of dup ACKs received for \( y \)
      - if (count of dup ACKs received for \( y = 3 \))
        - resend segment with sequence number \( y \)

Ders konuları

- Connection Oriented Transport: TCP
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management
TCP Flow Control

- **receive side of TCP connection has a receive buffer:**
  - sender won’t overflow receiver’s buffer by transmitting too much, too fast
  - speed-matching service: matching the send rate to the receiving app’s drain rate

- app process may be slow at reading from buffer

TCP Flow control: how it works

- Rcvr advertises spare room by including value of **RcvWindow** in segments
- Sender limits unACKed data to **RcvWindow**
  - guarantees receive buffer doesn’t overflow

(Suppose TCP receiver discards out-of-order segments)
- spare room in buffer

\[
\text{RcvWindow} = \text{RcvBuffer} - (\text{LastByteRcvd} - \text{LastByteRead})
\]
Ders konuları

- Connection Oriented Transport: TCP
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management

TCP Connection Management

**Recall:** TCP sender, receiver establish “connection” before exchanging data segments

- initialize TCP variables:
  - seq. #
  - buffers, flow control info (e.g. RcvWindow)
- client: connection initiator
  ```java
  Socket clientSocket = new Socket("hostname","port number");
  ```
- server: contacted by client
  ```java
  Socket connectionSocket = welcomeSocket.accept();
  ```

**Three way handshake:**

**Step 1:** client host sends TCP SYN segment to server
  - specifies initial seq #
  - no data

**Step 2:** server host receives SYN, replies with SYNACK segment
  - server allocates buffers
  - specifies server initial seq. #

**Step 3:** client receives SYNACK, replies with ACK segment, which may contain data
Closing a connection:

client closes socket:
```java
clientSocket.close();
```

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

**Step 2:** server receives FIN, replies with ACK. Closes connection, sends FIN.

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

**Step 4:** server, receives ACK. Connection closed.

**Note:** with small modification, can handle simultaneous FINs.
TCP Connection Management (cont)

TCP client lifecycle

TCP server lifecycle

Client application initiates a TCP connection

TCP client lifecycle

TCP server lifecycle

Server application creates a failed socket

TCP client lifecycle

TCP server lifecycle