Transport Layer, Part 2
Transmission Control Protocol

These slides are created by Dr. Huang of George Mason University. Students registered in Dr. Huang's courses at GMU can make a single machine readable copy and print a single copy of each slide for their own reference as long as the slide contains the copyright statement, and the GMU facilities are not used to produce the paper copies. Permission for any other use, either in machine-readable or printed form, must be obtained form the author in writing.

Introduction

- TCP provides
  - reliable, stream delivery service
  - full-duplex (bidirectional) communication
  - flow/error control
  - congestion control
  - connection establishment and destruction
- TCP does not include application program interfaces (API).
  - API is provided by operating systems.

Stream Delivery Service

- Data from an application are treated as a stream of bytes
- The stream is divided into **segments** for delivery.
  - this division is up to TCP; application data boundaries are ignored
- Conceptually, every byte has a sequence #.
- Only the sequence # of the first byte of the segment is sent.
- Window sizes are also in bytes (as opposed to the # of frames).

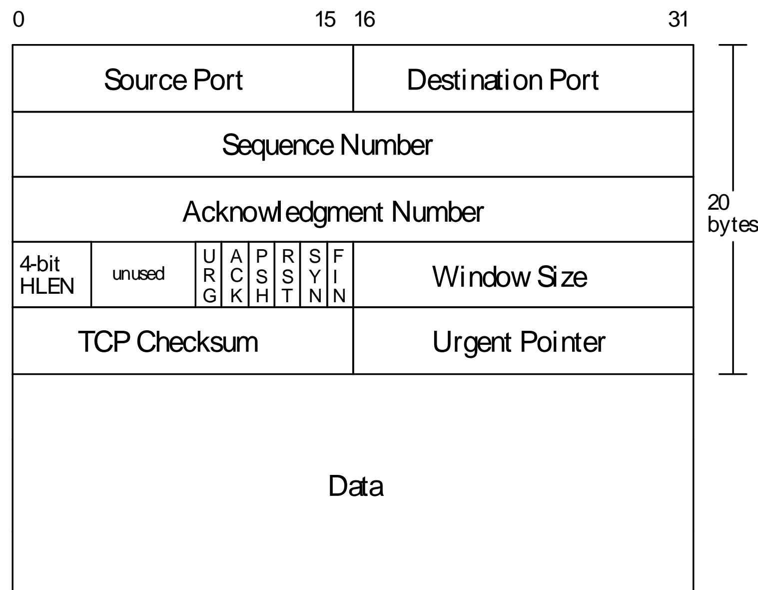
Forced Delivery

- Imagine that you telnet to `site.gmu.edu` and issue an `ls` command.
- While you wait for responses, your TCP module considers three bytes ('l', 's', and return) too small a segment and decides to wait for additional bytes before sending a segment.
- Applications sometimes need a way to force TCP to transmit data immediately.
- This is achieved by setting a PSH bit to 1 in TCP header

Urgent Data

- TCP also provides a way to urgently a part of the data stream to the remote process, regardless the position of that part of data in the stream.
- This is achieved by encapsulating the urgent data in a segment whose URG bit is set to 1.
- Also, a Urgent Pointer in TCP header allows you to specify where the urgent data ends.
- Applications ?

TCP Segment Format



The Concept of Connections

- Unlike UDP, TCP uses **connections**, not ports, to identify message queues
  - A connection is identified by a pair of endpoints.
  - An **endpoint** is an (IP address, port) pair.
- Examples: three connections involving `site.gmu.edu` (129.174.40.83):
  - (18.26.0.36, 53) ↔ (129.174.40.83, **25**)
  - (128.9.4.33, 2000) ↔ (129.174.40.83, 1069)
  - (196.83.4.22, 64) ↔ (129.174.40.83, **25**)

Connection Establishment

- Naïve solution: one party sends a connection request message, and the other either accepts or rejects the request.
- However, consider the "storage" property of the Internet.
  - each router uses buffers to store packets waiting for transmission
  - during congestion, a packet could be buffered in the network for a prolonged period of time
  - it looks as if the packet is temporarily stored in the network

A Horrible Scenario

- Image you establish a connection to a bank server to transfer 1 million dollars to a friend.
- The connection messages are "stored" in the Internet.
- Your machine times out and retransmits.
- The new set of messages arrive at the bank server before the original, probably because they follow different paths; 1 million transferred.
- Later, the original request arrives at the server, a second million transferred.

Solution

- Choose the initial sequence numbers (ISN) of a connection based on current time.
- The ISN is included in the connection request message.
- If the request message has to be retransmitted, a new ISN shall be chosen.
- This enables us to recognize obsolete request messages.

Three-Way Hand Shaking

- Machine *A* initiates a connection to machine *B*.
- *A* sends a segment with ISN *x*, chosen according *A*'s clock, and with SYN bit set to 1, this is the **connection request** message.
  - the ACK flag is set to 0
  - this segment is still a regular TCP segment; it can carry data too
  - however, if it does contain data, machine *B* cannot deliver the data to application until the connection is successfully established

- *B* returns a segment whose Ack=*x*+1, Seq=*y*, the ISN of *B* chosen according to *B*'s clock, and SYN=1.
  - this is the **connection acceptance** message.
  - if the first segment from A contains *b* bytes of data, then Ack=*x*+*b*+1.
  - this segment itself can contain data too.

- After receiving the acceptance, *A* considers the connection established.
- *A* then sends *B* a segment with Ack=y+1 and SYN=0.
  - this is the **connection confirmation** message
  - all segments from this point on will have SYN=0
- After receiving the confirmation, *B* considers the connection established.

Visualize It

Example

- A to B: SYN=1, ACK=0, seq=100
- B to A: SYN=1, ACK=1, seq=523, ack=101
  - when A receives this segment, it considers the connection successful
- A to B: SYN=0, ACK=1, seq=101, ack=524, data
  - after B receives this segment, it considers the connection successful
  - connection is now full-duplex

A Second Example

- A to B: SYN=1, ACK=0, seq=**100**

  segment lost due to transmission errors
- A to B: SYN=1, ACK=0, seq=**336**
- B to A: SYN=1, ACK=1, seq=**523**, ack=337

  segment lost due to transmission errors
- B to A: SYN=1, ACK=1, seq=**880**, ack=337
- A to B: SYN=0, ACK=1, seq=337, ack=881, data

Connection Termination

- Consider a connection between *A* and *B*.
- *A* terminates the connection by setting FIN=1 in its last outgoing segment.
  - SYN=0, FIN=1, seq=x, b bytes of data
- *A* considers the connection terminated after receiving the Ack of its last segment.
  - SYN=0, FIN=0, ack=$x+b+1$
- Both SYN and FIN counted as one byte in the sequence space.

Half-Closed Connections

- After *A* terminates the connection, the connection is still considered open by *B* and is said to be **half-close**.
  - when a connection is half-close, data can flow in only one direction (*B* to *A*)
  - *A* must continue receiving data from *B* and returning Acks.

- *B* is responsible for terminating the *B*-to-*A* direction by setting FIN=1 in B's last outgoing segment:
  - SYN=0, FIN=1, seq=*x*, *b* bytes of data
- *B* considers the connection closed after receiving the Ack of its last segment:
  - SYN=0, FIN=0, ack=$x+b+1$
  - the connection is then completely terminated

- It is up to B to decide whether it would terminate the connection upon receiving the FIN from A.
- Either endpoint can close the connection first, regardless who initiates the connection.

Example

- Consider a telnet session where the user logins to server B via his/her PC A.
- The user closes the session by typing "`exit\n`".
- A to B: FIN=1, seq=1060, ack=7777, data="`exit\n`"
- B to A: FIN=1, seq=7777, ack=1066
- A to B: FIN=0, seq=1066, ack=7778

Half-Close Example

- The last segment from client A to database server B is a request message, 10-byte long.
- A to B: FIN=1, seq=2405, ack=8002, 10-byte data
- B to A: FIN=0, seq=8002, ack=2405+10+1=2416, no data

- The server process reads the request and sends a long reply, corresponding to multiple segments.
  – the connection is half close when the reply is in transit
- Consider the last reply segment, assumed 100 bytes.
- B to A: FIN=1, seq=9125, 100-byte data
- A to B: FIN=0, ack=9125+100+1=9226

TCP Acknowledgments

- TCP acknowledges the next byte expected (not the last one received).
- Ack=x indicates *all* bytes up to and including x-1 has been successfully received.
- SYN=1 occupies a “byte” in the sequence number space; so does FIN=1.
- a pure Ack is simply a segment with no data
- A receiver could postpone acks up to 200ms for chances of piggybacking.

Exercise

- Assume that all bytes up to 999 have been received and acked. Give the acks in response to segments
  - Seg(1000, 200)
  - Seg(1300, 100)
  - Seg(1400, 300)
  - Seg(1700, 100)
  - Seg(1200, 100)

Exercise

- Give the acks in response to segments
  - Seg(1000, 200, SYN=1)
  - Seg(1300, 100)
  - Seg(1400, 300)
  - Seg(1700, 100, FIN=1)
  - Seg(1201, 99)

TCP Sliding Window Protocol

- The sliding window protocol of TCP must work with dynamic conditions.
  - Unlike the DLL, round-trip times between the two endpoints are unknown a prior.
  - Moreover, round-trip times not fixed, due to fluctuations in background traffic.
- Time-out intervals must constantly adjusted according to present RTT.

RTT and Timeout Interval

- If the timeout interval is too short, unnecessary retransmissions occur
- If the interval is too long, performance suffers when packets are actually lost (waiting for too long to retransmit).
- Recall that round-trip times are not known a prior and could change (perhaps widely) during the lifespan of the connection.
- We need to measure RTT in order to derive timeout intervals.

An Early Solution

- When an outgoing packet is stored in the window, the current time T_1 is recorded.
- When its ack arrives at time T_2, we obtain an RTT sample: New_Sample=$T_2 - T_1$.
- A variable RTT is maintained for each conn.
- When an ack arrives

  RTT = α*RTT + (1- α)*(New_Sample)

  Timeout = β * RTT

  where $\alpha = 0.9$ and $\beta = 2$.

Problem ?

- What is your choice of timeout interval if the past three RTT samples are
  – 10, 9.1, and 10.1 milliseconds

  – 1, 20, and 9 milliseconds

RTT Variances

- Associated with each connection is a second variable, DEV, to estimate the mean deviation of round trip times.

  DIFF = SAMPLE – RTT

  RTT = RTT + δ*DIFF

  DEV= DEV + ρ*(|DIFF| - DEV)

  Timeout = RTT + τ*DEV

  where δ is recommended to be 1/8, ρ 1/4, and τ 4.

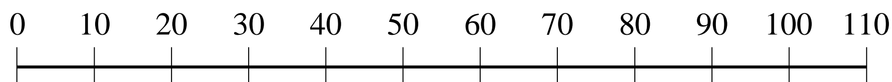
Retransmission Ambiguity Problem

- Consider a packet with seq=*x*, first transmitted at time T_1 and then retransmitted at time T_2.
- An Ack(*x*) arrives at time T_3.
- What is the value of the new RTT sample ?

  $T_3 - T_2$ or $T_3 - T_1$?

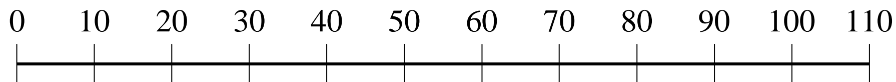
Naïve Solution: discard all samples pertaining to packets that have been retransmitted.

Problem: assuming real RTT=30 ms, and current timeout=20 ms



Kern's Algorithm

- When computing the RTT variable, ignore all round-trip time samples that correspond to retransmitted segments.
- Further, when an ACK that corresponds to a retransmitted packet arrives, double the timeout interval.
- Derive the timeout interval from variable RTT when a valid RTT sample arrives.



Put Them Together

```
When an Ack arrives:
        if (it is ambiguous)
                Timeout = 2*Timesout
        else
                DIFF = SAMPLE – RTT
                RTT = RTT + δ*DIFF
                DEV= DEV + ρ*(|DIFF| - DEV)
                Timeout = RTT + τ*DEV
        endif
```