Lecture 13 (March 2, 2004)

Outline

Flow Control

TCP Overview

Transport layer functions

- Transport protocol defines the pattern/sequence for end hosts sending packets
- Transport has to deal with a number of things
  - Multiplexing/demultiplexing – ports and message queues
  - Error detection within packets - checksums
  - Reliable, in order delivery of packets - Today
  - Flow control – Today
  - Connection management - TBD
  - Congestion control – TBD
- We're moving toward understanding TCP

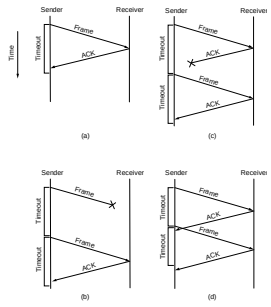
Methods of Reliability

- Packets can be lost and/or corrupted during transmission
  - Bit level errors due to noise
  - Loss due to congestion
- Use checksums to detect bit level errors
  - Internet Checksum is optionally used to detect errors in UDP
    - Uses 16 bits to encode one's complement sum of data + headers
  - When bit level errors are detected, packets are dropped
- Build reliability into the transmission protocol
  - Using *acknowledgements* and *timeouts* to signal lost or corrupt frame

Acknowledgements & Timeouts

- An *acknowledgement* (ACK) is a packet sent by one host in response to a packet it has received
  - Making a packet an ACK is simply a matter of changing a field in the transport header
  - Data can be *piggybacked* in ACKs
- A *timeout* is a signal that an ACK to a packet that was sent has not yet been received within a specified timeframe
  - A timeout triggers a *retransmission* of the original packet from the sender
  - How are timers set?

Acknowledgements & Timeouts



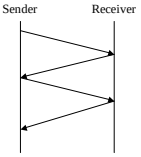
Propagation Delay

- Propagation delay is defined as the delay between transmission and receipt of packets between hosts
- Propagation delay can be used to estimate timeout period
- How can propagation delay be measured?
- What else must be considered in the measurement?

Exponentially weighted moving average RTT estimation

- EWMA was original algorithm for TCP
- Measure **SampleRTT** for each packet/ACK pair
- Compute weighted average of RTT
  - **EstRTT** = α x **EstimatedRTT** + β x **SampleRTT**
  - where $\alpha + \beta = 1$
  - α between 0.8 and 0.9
  - β between 0.1 and 0.2
- Set timeout based on **EstRTT**
  - **TimeOut** = **2** x **EstRTT**

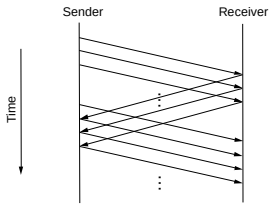
Stop-and-Go Process



- Sender doesn't send next packet until he's sure receiver has last packet
- The packet/Ack sequence enables reliability
- Sequence numbers help avoid problem of duplicate packets
- Problem: keeping the pipe full
- Example
  - 1.5Mbps link x 45ms RTT = 67.5Kb (8KB)
  - 1KB frames imples 1/8th link utilization

Solution: Pipelining via Sliding Window

- Allow multiple outstanding (un-ACKed) frames
- Upper bound on un-ACKed frames, called *window*



Buffering on Sender and Receiver

- Sender needs to buffer data so that if data is lost, it can be resent
- Receiver needs to buffer data so that if data is received out of order, it can be held until all packets are received
  - Flow control
- How can we prevent sender overflowing receiver's buffer?
  - Receiver tells sender its buffer size during connection setup
- How can we insure reliability in pipelined transmissions?
  - Go-Back-N
    - Send all N unACKed packets when a loss is signaled
    - Inefficient
  - Selective repeat
    - Only send specifically unACKed packets
    - A bit trickier to implement

Sliding Window: Sender

- Assign sequence number to each frame (**SeqNum**)
- Maintain three state variables:
  - send window size (**SWS**)
  - last acknowledgment received (**LAR**)
  - last frame sent (**LFS**)
- Maintain invariant: **LFS** - **LAR** <= **SWS**

- Advance **LAR** when ACK arrives
- Buffer up to **SWS** frames

Sliding Window: Receiver

- Maintain three state variables
  - receive window size (**RWS**)
  - largest frame acceptable (**LFA**)
  - last frame received (**LFR**)
- Maintain invariant: **LFA** - **LFR** <= **RWS**

- Frame **SeqNum** arrives:
  - if **LFR** < **SeqNum** < = **LFA** ⟶ accept
  - if **SeqNum** < = **LFR** or **SeqNum** > **LFA** ⟶ discarded
- Send *cumulative* ACKs – send ACK for largest frame such that all frames less than this have been received

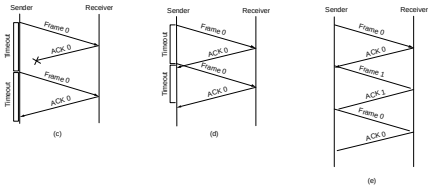
Sequence Number Space

- **SeqNum** field is finite; sequence numbers wrap around
- Sequence number space must be larger then number of outstanding frames
- **SWS** <= **SeqNumSpace** is not sufficient
  - suppose 3-bit **SeqNum** field (0..7)
  - **SWS=RWS=7**
  - sender transmit frames 0..6
  - arrive successfully, but ACKs lost
  - sender retransmits 0..6
  - receiver expecting 7, 0..5, but receives the original incarnation of 0..5
- **SWS** <= **SeqNumSpace/2** is correct rule
- Intuitively, **SeqNum** "slides" between two halves of sequence number space

Another Pipelining Possibility: Concurrent Logical Channels

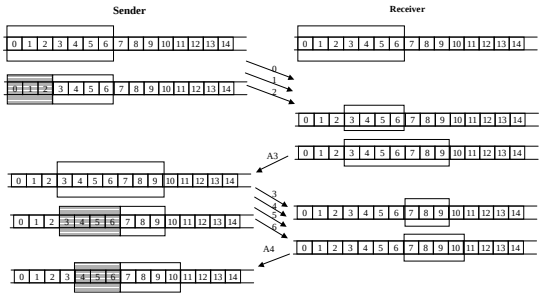
- Multiplex 8 logical channels over a single link
- Run stop-and-wait on each logical channel
- Maintain three state bits per channel
  - channel busy
  - current sequence number out
  - next sequence number in
- Header: 3-bit channel num, 1-bit sequence num
  - 4-bits total
  - same as sliding window protocol
- Separates *reliability* from *order*

Stop & Go sequence numbers



• Simple sequence numbers enable *the client* to discard duplicate copies of the same frame
• Stop & go allows one outstanding frame, requires two distinct sequence numbers

Sliding Window Example



Sliding Window Summary

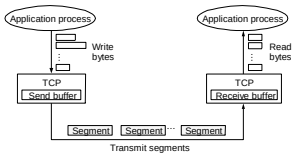
- Sliding window is best known algorithm in networking
- First role is to enable reliable delivery of packets
  - Timeouts and acknowledgements
- Second role is to enable in order delivery of packets
  - Receiver doesn't pass data up to app until it has packets in order
- Third role is to enable flow control
  - Prevents server from overflowing receiver's buffer

TCP Overview

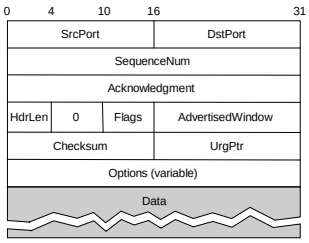
- TCP is the most widely used Internet protocol
  - Web, Peer-to-peer, FTP, telnet, …
- A two way, reliable, byte stream oriented end-to-end protocol
  - Includes flow and congestion control
- Closely tied to the Internet Protocol (IP)
- A focus of intense study for many years
  - Our goal is to understand the RENO version of TCP
    - RENO is most widely used TCP today
    - RFC 2001 (now expired)
    - RENO mainly specifies mechanisms for dealing with congestion

TCP Features

- Connection-oriented
- Byte-stream
  - app writes bytes
  - TCP sends *segments*
  - app reads bytes
- Reliable data transfer
- Full duplex
- Flow control: keep sender from overrunning receiver
- Congestion control: keep sender from overrunning network

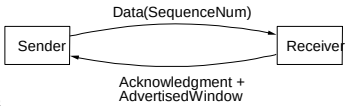


Segment Format



Segment Format (cont)

- Each connection identified with 4-tuple:
  - **`(SrcPort, SrcIPAddr, DsrPort, DstIPAddr)`**
- Sliding window + flow control
  - **`acknowledgment, SequenceNum, AdvertisedWinow`**



- Flags
  - **`SYN, FIN, RESET, PUSH, URG, ACK`**
- Checksum is the same as UDP
  - pseudo header + TCP header + data

Sequence Numbers

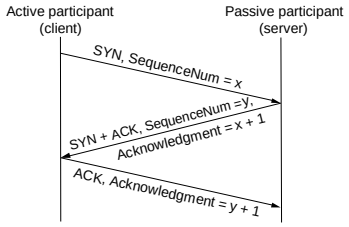
- 32 bit sequence numbers
  - Wrap around supported
- TCP breaks byte stream from application into packets (limited by Max. Segment Size)
- Each byte in the data stream is considered
- Each packet has a sequence number
  - Initial number selected at connection time
  - Subsequent numbers indicate first data byte number in packet
- ACK's indicate ***next byte expected***

Sequence Number Wrap Around

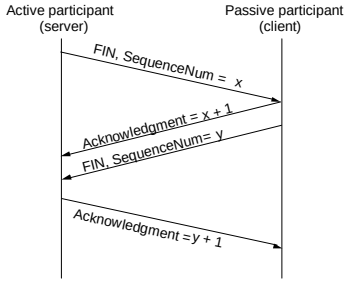
Bandwidth	Time Until Wrap Around
T1 (1.5 Mbps)	6.4 hours
Ethernet (10 Mbps)	57 minutes
T3 (45 Mbps)	13 minutes
FDDI (100 Mbps)	6 minutes
STS-3 (155 Mbps)	4 minutes
STS-12 (622 Mbps)	55 seconds
STS-24 (1.2 Gbps)	28 seconds

• Protect against this by adding a 32-bit timestamp to TCP header

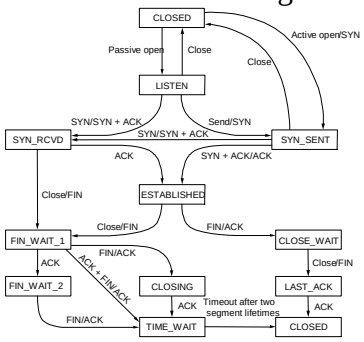
Connection Establishment



Connection Termination



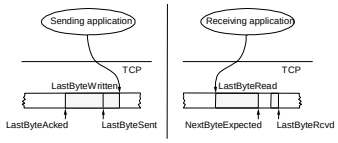
State Transition Diagram



Reliability in TCP

- Checksum used to detect bit level errors
- Sequence numbers used to detect sequencing errors
  - Duplicates are ignored
  - Reordered packets are reordered (or dropped)
  - Lost packets are retransmitted
- Timeouts used to detect lost packets
  - Requires RTO calculation
  - Requires sender to maintain data until it is ACKed

Sliding Window Revisited



- Sending side
  - **`LastByteAcked`** < = **`LastByteSent`**
  - **`LastByteSent`** < = **`LastByteWritten`**
  - buffer bytes between **`LastByteAcked`** and **`LastByteWritten`**
- Receiving side
  - **`LastByteRead`** < **`NextByteExpected`**
  - **`NextByteExpected`** < = **`LastByteRcvd +1`**
  - buffer bytes between **`NextByteRead`** and **`LastByteRcvd`**

Flow Control in TCP

- Send buffer size: **`MaxSendBuffer`**
- Receive buffer size: **`MaxRcvBuffer`**
- Receiving side
  - **`LastByteRcvd`** - **`LastByteRead`** < = **`MaxRcvBuffer`**
  - **`AdvertisedWindow`** = **`MaxRcvBuffer`** - (**`LastByteRcvd`** - **`LastByteRead`**)
- Sending side
  - **`LastByteSent`** - **`LastByteAcked`** < = **`AdvertisedWindow`**
  - **`EffectiveWindow`** = **`AdvertisedWindow`** - (**`LastByteSent`** - **`LastByteAcked`**)
  - **`LastByteWritten`** - **`LastByteAcked`** < = **`MaxSendBuffer`**
  - block sender if (**`LastByteWritten`** - **`LastByteAcked`**) + *y* > **`MaxSenderBuffer`**
- Always send ACK in response to arriving data segment
- Persist sending one byte seg. when **`AdvertisedWindow = 0`**

Keeping the Pipe Full

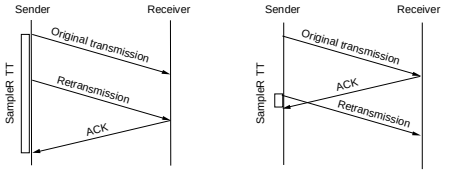
- 16-bit **`AdvertisedWindow`** controls amount of pipelining
- Assume RTT of 100ms
- Add scaling factor extension to header to enable larger windows

Bandwidth	Delay x Bandwidth Product
T1 (1.5 Mbps)	18KB
Ethernet (10 Mbps)	122KB
T3 (45 Mbps)	549KB
FDDI (100 Mbps)	1.2MB
OC-3 (155 Mbps)	1.8MB
OC-12 (622 Mbps)	7.4MB
OC-24 (1.2 Gbps)	14.8MB

Making TCP More Efficient

- Delayed acknowledgements
  - Delay for about 200ms
  - Try to piggyback ACKs with data
- Acknowledge every other packet
  - Many instances in transmission sequence which require an ACK
- Don't forget Nagle's algorithm
  - Can be switched off

Karn/Partridge Algorithm for RTO



- Two degenerate cases with timeouts and RTT measurements
  - Solution: Do not sample RTT when retransmitting
- After each retransmission, set next RTO to be double the value of the last
  - Exponential backoff is well known control theory method
  - Loss is most likely caused by congestion so be careful

Jacobson/ Karels Algorithm

- In late '80s, Internet was suffering from *congestion collapse*
- New Calculations for average RTT – Jacobson '88
- Variance is not considered when setting timeout value
  - If variance is small, we could set RTO = EstRTT
  - If variance is large, we may need to set RTO > 2 x EstRTT
- New algorithm calculates both variance and mean for RTT
- **`Diff`** = **`sampleRTT`** - **`EstRTT`**
- **`EstRTT = EstRTT + ( d x Diff)`**
- **`Dev = Dev + d ( |Diff| - Dev)`**
  - Initially settings for **`EstRTT`** and **`Dev`** will be given to you
  - where d is a factor between 0 and 1
  - typical value is 0.125

Jacobson/ Karels contd.

- **`TimeOut`** = μ x **`EstRTT`** + ϕ x **`Dev`**
  - where $\mu = 1$ and $\phi = 4$
- When variance is small, TimeOut is close to EstRTT
- When variance is large Dev dominates the calculation
- Another benefit of this mechanism is that it is very efficient to implement in code (does not require floating point)
- Notes
  - algorithm only as good as granularity of clock (500ms on Unix)
  - accurate timeout mechanism important to congestion control (later)
- These issues have been studied and dealt with in new RFC's for RTO calculation.
- TCP RENO uses Jacobson/Karels