CS 640 Introduction to Computer Networks

Lecture15

CS 640

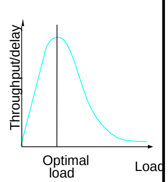
Today's lecture

- Transport layer – TCP

CS 640

Congestion in the Internet

- Checksums are effective for detecting bit errors but they are not the only problem…
- We know that traffic is bursty
  - Statistical multiplexing of ON/OFF sources
  - Heavy-tailed file sizes
  - Routers have limited buffer capacity
  - Packets dropped when buffers full
    - Buffers do protect from short bursts
- Congestion lengthens delays and lowers throughput
  - Standard throughput/load curve



CS 640

How can we deal with congestion?

- Over-provision networks
  - Very expensive
  - Commonly done
    - Networks designed to normally operate at 5-50% capacity
- Call admission control (phone networks)
- Develop protocols to respond to congestion
  - Route away from congestion
    - Good idea – how can we do it?
  - Retransmit in the face of loss
    - This is the state of the art

Congestion Control Basics

- UDP will send packets at any specified rate
  - Does not have mechanisms to handle congestion
- Issues:
  - Detecting congestion
  - Reacting to congestion
  - Avoiding congestion
    - Shaping traffic
    - QoS mechanisms
- Transport protocol will deal with congestion…

Congestion control in the Internet

- TCP implements congestion control
  - Detects congestion through packet losses
  - Reduces rate aggressively in response to congestion
  - Increases rate cautiously to use up available bandwidth
  - Works well for large flows
- Why the Internet doesn't experience congestion collapse
  - Backbones overprovisioned
  - TCP congestion control
  - Sources' rate limited by nearest bottleneck link

Next two lectures

- TCP
  - Introduction
  - Header format
  - Connection establishment and termination
  - Reliability
  - Roundtrip estimation
  - Congestion control (not today)

CS 640

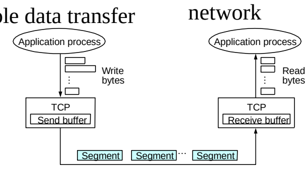
TCP Overview

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

CS 640

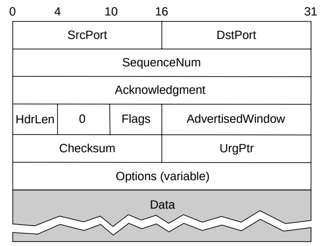
TCP Features

- 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



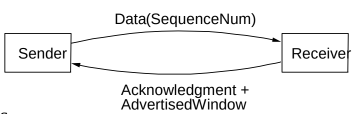
CS 640

Segment Format



Segment Format (cont)

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



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

Sequence Numbers

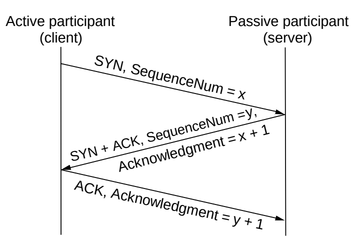
- 32 bit sequence numbers
  - Wrap around supported
- TCP breaks byte stream from application into packets (limited by Max. Segment Size)
- Each byte in the data stream is considered
- Each packet has a sequence number
  - Initial number selected at connection time
  - Subsequent numbers give first data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

Sequence Number Wrap Around

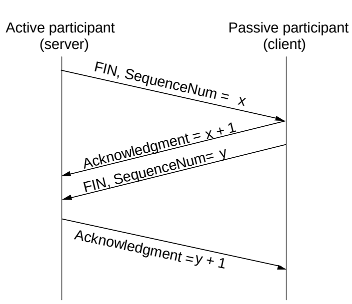
Bandwidth	Time Until Wrap Around
T1 (1.5 Mbps)	6.4 hours
Ethernet (10 Mbps)	57 minutes
T3 (45 Mbps)	13 minutes
FDDI (100 Mbps)	6 minutes
STS-3 (155 Mbps)	4 minutes
STS-12 (622 Mbps)	55 seconds
STS-24 (1.2 Gbps)	28 seconds

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

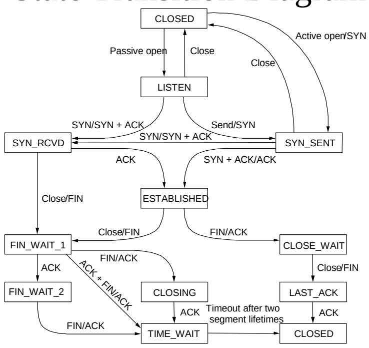
Connection Establishment



Connection Termination



State Transition Diagram



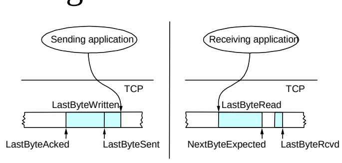
CS 640

Reliability in TCP

- Checksum used to detect bit level errors
- Sequence numbers help detect sequencing errors
  - Duplicates are ignored
  - Out of order 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

CS 640

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`**

CS 640

Flow Control in TCP

- Send buffer size: `MaxSendBuffer`
- Receive buffer size: `MaxRcvBuffer`
- Receiving side
  - `LastByteRcvd` - `LastByteRead` < = `MaxRcvBuffer`
  - `AdvertisedWindow` = `MaxRcvBuffer` - (`NextByteExpected` -1 - `LastByteRead`)
- Sending side
  - `LastByteWritten` - `LastByteAcked` < = `MaxSendBuffer`
  - block sender if (`LastByteWritten` - `LastByteAcked`) + *y* > `MaxSenderBuffer`
  - `LastByteSent` - `LastByteAcked` < = `AdvertisedWindow`
  - `EffectiveWindow` = `AdvertisedWindow` - (`LastByteSent` - `LastByteAcked`)
- Always send ACK in response to arriving data segment
- Persist sending one byte seg. when `AdvertisedWindow = 0`

Keeping the Pipe Full

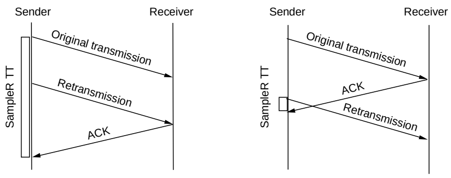
- 16-bit `AdvertisedWindow` controls amount of pipelining
- Assume RTT of 100ms
- Add scaling factor extension to header to enable larger windows

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
  - Try to piggyback ACKs with data
  - Try not to send small packets, sender sends only when it has enough data to fill MSS
    - See Nagle's algorithm
- Acknowledge every other packet
  - Many instances in transmission sequence which require an ACK

Karn/Partridge Algorithm for RTO



- Degenerate cases with for 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

CS 640

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 + δ x Diff`**
- **`Dev = Dev + δ ( |Diff| - Dev)`**
  - Initially settings for **`EstRTT`** and **`Dev`** given
  - δ is a factor between 0 and 1 (typical value is 0.125)

CS 640

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

CS 640