CS 640 Introduction to Computer Networks

Lecture25

CS 640

Today's lecture

- Transport layer
  - UDP
  - TCP (except congestion control)

CS 640

Layering and Encapsulation Revisited

- Each layer relies on layers below to provide services in black box fashion
  - Layering makes complex systems easier to understand & specify
  - Makes implementation more flexible
  - Can make implementation bigger and less efficient
  - Layers are implemented by protocols – rules for communication
- Data from applications moves up and down protocol stack
  - Application level data is chopped into packets (segments)
  - Encapsulation deals with attaching headers at layers 2, 3, 4

CS 640

End-to-End Protocols

- Underlying network is *best-effort* so it can:
  - drop messages
  - re-orders messages
  - delivers duplicate copies of a given message
  - deliver messages after an arbitrarily long delay
- Common end-to-end services do:
  - guarantee message delivery
  - deliver messages in the same order they are sent
  - deliver at most one copy of each message
  - support synchronization
  - allow the receiver to flow control the sender
  - support multiple application processes on each host

Basic function of transport layer

- How can processes on different systems get the right messages?
- *Ports* are numeric locators which enable messages to be demultiplexed to proper process.
  - Ports are addresses on individual hosts, not across the Internet
- Ports are established using *well-know* values first
  - Port 80 = http, port 53 = DNS
- Ports are typically implemented as message queues
- Simplest function of the transport layer is multiplexing/ demultiplexing of messages

Other transport layer functions

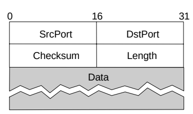
- Connection control
  - Setting up and tearing down communication between processes
- Error detection within packets
  - Checksums
- Reliable, in order delivery of packets
  - Acknowledgement schemes
- Flow control
  - Matching sending and receiving rates between end hosts
- Congestion control
  - Managing congestion in the network

Today's lecture

- Transport layer
  - UDP
  - TCP (except congestion control)

User Datagram Protocol (UDP)

- Unreliable and unordered *datagram* service
- Adds multiplexing/demultiplexing
- Adds reliability through optional checksum
- No flow or congestion control
- Endpoints identified by ports
  - servers have *well-known* ports
  - see **`/etc/services`** on Unix
- Header format



- Optional checksum
  - Computed over pseudo header + UDP header + data

UDP Checksums

- Optional in current Internet
- Covers payload + pseudoheader
- Pseudoheader consists of 3 fields from IP header: protocol number (TCP or UDP), IP src, IP dst and UDP length field
  - Pseudoheader enables verification that message was delivered between correct source and destination.
  - IP dest address was changed during delivery, checksum would reflect this
- UDP uses the same checksum algorithm as IP

UDP in practice

- Minimal requirements make UDP very flexible
  - Any end-to-end protocol can be implemented
    - Remote Procedure Calls (RPC)
    - TCP can be implemented using UDP
- Examples
  - Most commonly used in multimedia applications
    - These are frequently more robust to loss
  - RPCs
  - Many others…

Today's lecture

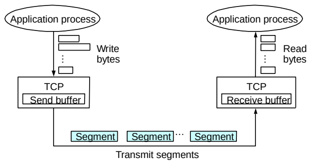
- Transport layer
  - UDP
  - TCP (except congestion control)

TCP Overview

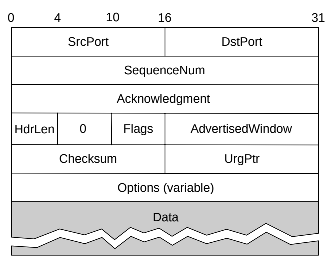
- TCP is the most widely used transport protocol
  - Web, Peer-to-peer, FTP, telnet, …
  - A focus of intense study for many years
- A two way, reliable, byte stream oriented end-to-end protocol
- Closely tied to the Internet Protocol (IP)
- Our goal is to understand the RENO version of TCP (most widely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

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

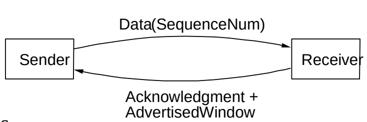


Segment Format



Segment Format (cont)

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



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

Sequence Numbers

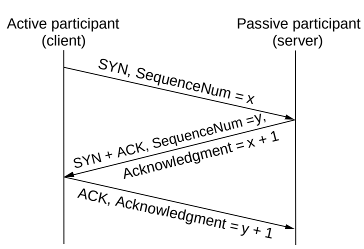
- 32 bit sequence numbers
  - Wrap around supported
- TCP breaks byte stream from application into packets (limited by Max. Segment Size)
- Each byte in the data stream is considered
- Each packet has a sequence number
  - Initial number selected at connection time
  - Subsequent numbers give first data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

Sequence Number Wrap Around

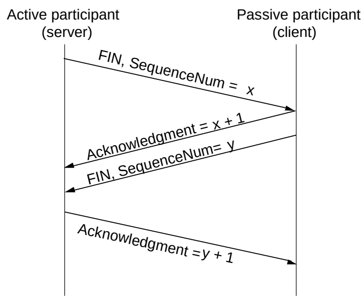
Bandwidth	Time Until Wrap Around
T1 (1.5 Mbps)	6.4 hours
Ethernet (10 Mbps)	57 minutes
T3 (45 Mbps)	13 minutes
FDDI (100 Mbps)	6 minutes
STS-3 (155 Mbps)	4 minutes
STS-12 (622 Mbps)	55 seconds
STS-24 (1.2 Gbps)	28 seconds

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

Connection Establishment

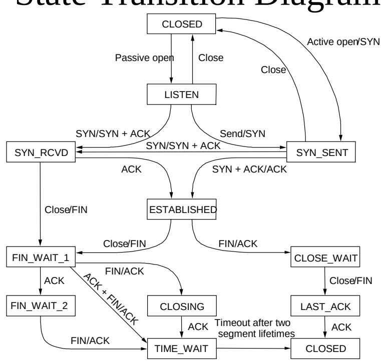


Connection Termination



CS 640

State Transition Diagram



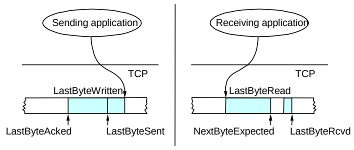
CS 640

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**

CS 640

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

CS 640

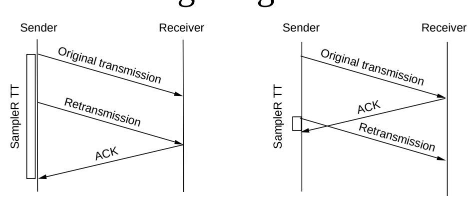
Making TCP More Efficient

- 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

Basic RTT estimation

- Using exponentially weighted moving average
  - **EstRTT=EstRTT+ (1-α)·(SampleRTT-EstRTT)**
  - α set to between `0.8` and `0.9`
- Retransmission timeout set conservatively
  - **RTO=2 · EstRTT**

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

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