CS162
Operating Systems and Systems Programming
Lecture 15

Reliability, Transport Protocols

March 16, 2011
Ion Stoica
http://inst.eecs.berkeley.edu/~cs162

Goals for Today

- Finish e2e argument & fate sharing
- Transport: TCP/UDP
  - Reliability
  - Flow control

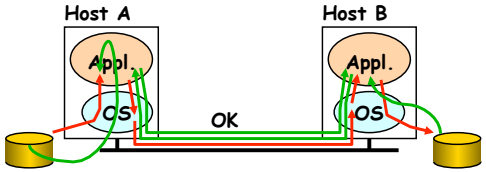
Placing Network Functionality

- Hugely influential paper: "End-to-End Arguments in System Design" by Saltzer, Reed, and Clark ('84)
- "Sacred Text" of the Internet
  - Endless disputes about what it means
  - Everyone cites it as supporting their position

Basic Observation

- Some types of network functionality can only be correctly implemented end-to-end
  - Reliability, security, etc
- Because of this, end hosts:
  - Can satisfy the requirement without network's help
  - Will/**must** do so, since can't ***rely*** on network's help
- Therefore **don't** go out of your way to implement them in the network

Example: Reliable File Transfer



- Solution 1: make each step reliable, and then concatenate them
- Solution 2: end-to-end **check** and try again if necessary

Discussion

- Solution 1 is incomplete
  - What happens if memory is corrupted?
  - Receiver has to do the check anyway!
- Solution 2 is complete
  - Full functionality can be entirely implemented at application layer with no need for reliability from lower layers
- *Is there any need to implement reliability at lower layers?*
  - Well, it could be more efficient

Summary of End-to-End Principle

Implementing this functionality in the network:

- Doesn't reduce host implementation complexity
- Does increase network complexity
- Probably imposes delay and overhead on all applications, even if they don't need functionality
- However, implementing in network can enhance performance in some cases
  - E.g., very losy link

Conservative Interpretation of E2E

- Don't implement a function at the lower levels of the system unless it can be completely implemented at this level
- Unless you can relieve the burden from hosts, don't bother

Moderate Interpretation

- Think twice before implementing functionality in the network
- If hosts can implement functionality correctly, implement it in a lower layer only as a performance enhancement
- But do so only if it does not impose burden on applications that do not require that functionality

Related Notion of *Fate-Sharing*

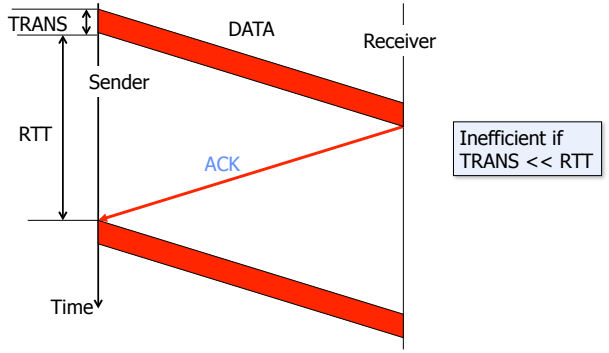
- Idea: when storing state in a distributed system, keep it co-located with the entities that ultimately rely on the state
- Fate-sharing is a technique for dealing with failure
  - Only way that failure can cause loss of the critical state is if the entity that cares about it also fails ...
  - … in which case it doesn't matter
- Often argues for keeping *network state* at end hosts rather than inside routers
  - In keeping with End-to-End principle
  - E.g., packet-switching rather than circuit-switching
  - E.g., NFS file handles, HTTP "cookies"

Reliable Transfer

- Retransmit missing packets
  - Numbering of packets and ACKs
- Do this efficiently
  - Keep transmitting whenever possible
  - Detect missing ACKs and retransmit quickly
- Two schemes
  - Stop & Wait
  - Sliding Window (Go-back-n and Selective Repeat)

Stop & Wait

- Send; wait for ack
- If timeout, retransmit; else repeat



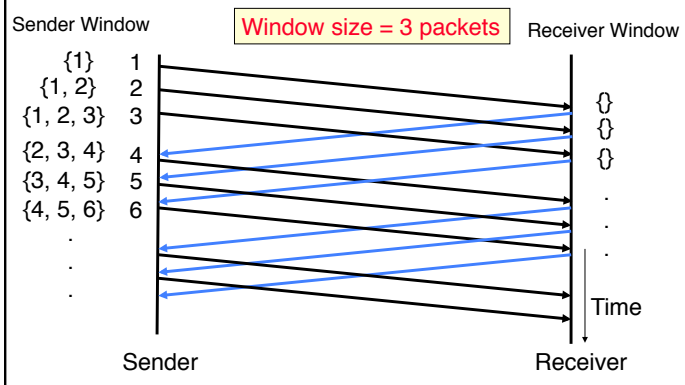
Sliding Window

- *window* = set of adjacent sequence numbers
- The size of the set is the *window size*
- Assume window size is n
- Let A be the last ack'd packet of sender without gap; then window of sender = {A+1, A+2, …, A+n}
- Sender can send packets in its window
- Let B be the last received packet without gap by receiver, then window of receiver = {B+1,…, B+n}
- Receiver can accept out of sequence, if in window

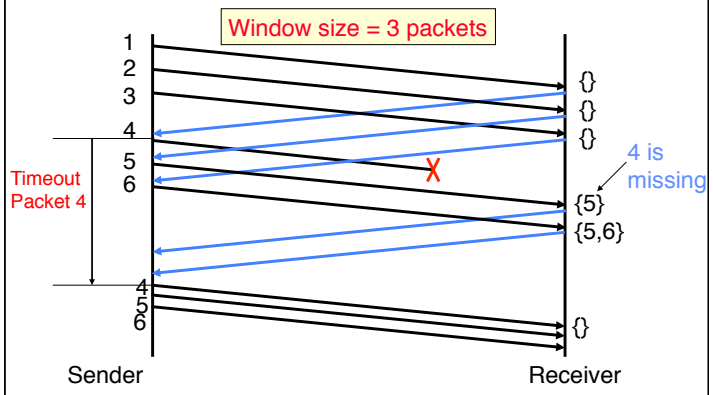
Go-Back-n (GBN)

- Transmit up to *n* unacknowledged packets
- If timeout for ACK(*k*), retransmit *k*, *k+1*, …

GBN Example w/o Errors



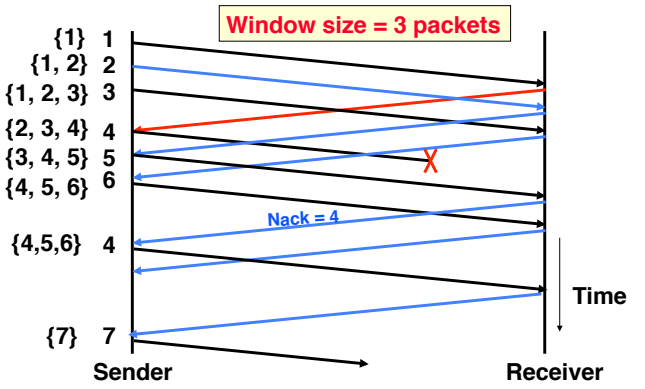
GBN Example with Errors



Selective Repeat (SR)

- Sender: transmit up to *n* unacknowledged packets; assume packet *k* is lost
- Receiver: indicate packet *k* is missing
- Sender: retransmit packet *k*

SR Example with Errors



Observations

- With sliding windows, it is possible to fully utilize a link, provided the window size is large enough. Throughput is ~ (n/RTT)
  - Stop & Wait is like n = 1.
- Sender has to buffer all unacknowledged packets, because they may require retransmission
- Receiver may be able to accept out-of-order packets, but only up to its buffer limits

Motivation for Transport Protocols

- IP provides a weak, but efficient service model (*best-effort*)
  - Packets can be delayed, dropped, reordered, duplicated
  - Packets have limited size (why?)
- IP packets are addressed to a host
  - How to decide which application gets which packets?
- How should hosts send packets into the network?
  - Too fast may overwhelm the network
  - Too slow is not efficient

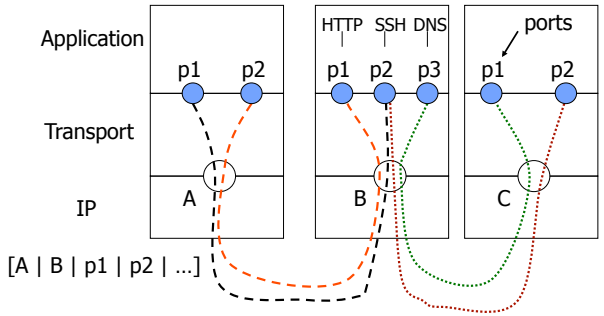
Transport Layer

- Provide a way to decide which packets go to which applications (*multiplexing/demultiplexing*)
- Can
  - Provide reliability, in order delivery, at most once delivery
  - Support messages of arbitrary length
  - Govern when hosts should send data → can implement congestion and flow control

Congestion vs. Flow Control

- Flow Control – avoid overflowing the receiver
- Congestion Control – avoid congesting the network
- What is network congestion?

Transport Layer (cont'd)



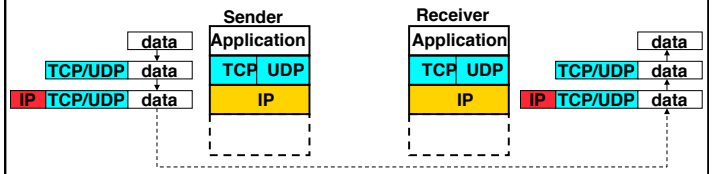
UDP: Not reliable
TCP: Ordered, reliable, well-paced

Ports

- Need to decide which application gets which packets
- Solution: map each socket to a *port*
- Client must know server's port
- Separate 16-bit port address space for UDP and TCP
  - (src_IP, src_port, dst_IP, dst_port) uniquely identifies TCP connection
- *Well known ports* (0-1023): everyone agrees which services run on these ports
  - e.g., ssh:22, http:80
  - On UNIX, must be root to gain access to these ports (why?)
- Ephemeral ports (most 1024-65535): given to clients
  - e.g. chat clients, p2p networks

Headers

- IP header → used for IP routing, fragmentation, error detection
- UDP header → used for multiplexing/demultiplexing, error detection
- TCP header → used for multiplexing/demultiplexing, flow and congestion control



UDP: User (Unreliable) Data Protocol

- Minimalist transport protocol
- Same best-effort service model as IP
- Messages up to 64KB
- Provides multiplexing/demultiplexing to IP
- Does not provide flow and congestion control
- Application examples: video/audio streaming

UDP Service & Header

- Service:
  - Send datagram from (IPa, Port1) to (IPb, Port2)
  - Service is unreliable, but error detection possible
- Header:

0	16	31
Source port	**Destination port**	
UDP length	**UDP checksum**	
Payload (variable)		

•UDP length is UDP packet length (including UDP header and payload, but not IP header)
•Optional UDP checksum is over UDP packet

TCP: Transport Control Protocol

- Reliable, in-order, and at most once delivery
- Stream oriented: messages can be of arbitrary length
- Provides multiplexing/demultiplexing to IP
- Provides congestion control and avoidance
- Application examples: file transfer, chat

TCP Service

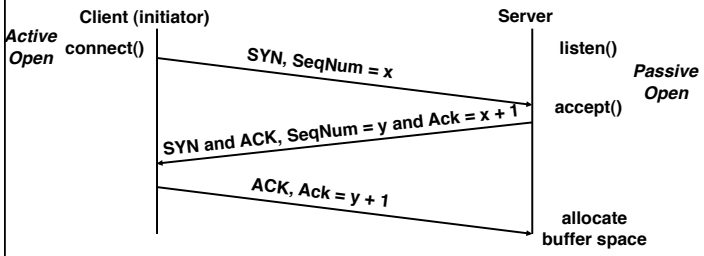
1) Open connection: 3-way handshaking

2) Reliable byte stream transfer from (IPa, TCP_Port1) to (IPb, TCP_Port2)
   - Indication if connection fails: Reset

3) Close (tear-down) connection

Open Connection: 3-Way Handshaking

- Goal: agree on a set of parameters: the start sequence number for each side
  - Starting sequence numbers are random

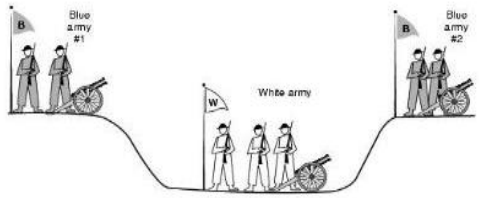


3-Way Handshaking (cont'd)

- Three-way handshake adds 1 RTT delay
- Why?
  - Congestion control: SYN (40 byte) acts as cheap probe
  - Protects against delayed packets from other connection (would confuse receiver)

Close Connection (Two-Army Problem)

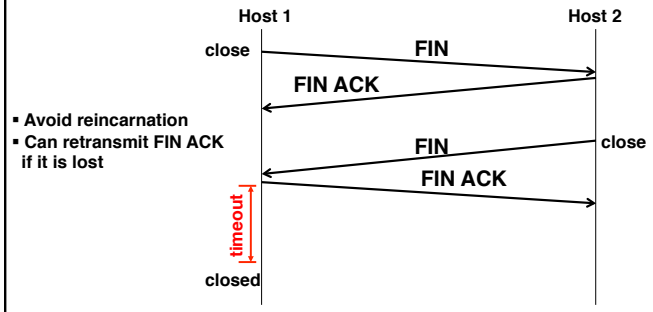
- Goal: both sides agree to close the connection
- Two-army problem:
  - "Two blue armies need to simultaneously attack the white army to win; otherwise they will be defeated. The blue army can communicate only across the area controlled by the white army which can intercept the messengers."



- What is the solution?

Close Connection

- 4-ways tear down connection



TCP Flow Control

- Each byte has a sequence number
- Initial sequence numbers negotiated via SYN/SYN-ACK packets
- ACK contains the sequence number of the next byte expected by the receiver

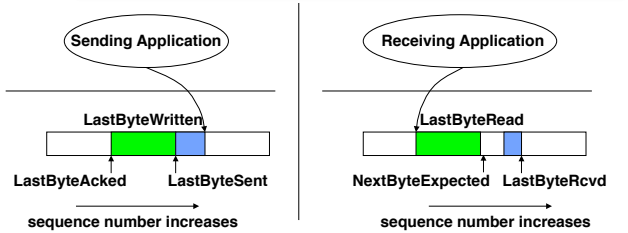
TCP Flow Control

- Receiver window (MaxRcvBuf – maximum buffer size at receiver)

AdvertisedWindow = MaxRcvBuffer – (LastByteRcvd – LastByteRead)

- Sender window (MaxSendBuf – maximum buffer size at sender)

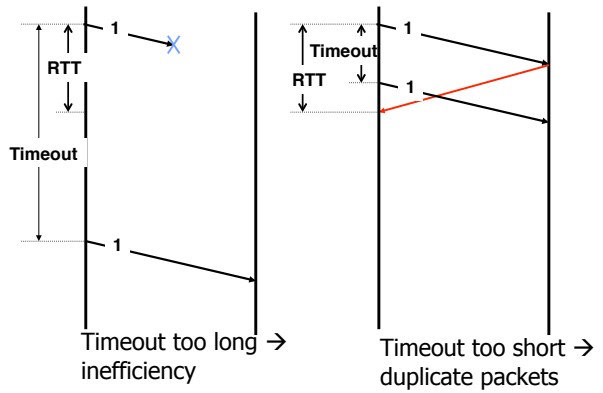
SenderWindow = AdvertisedWindow – (LastByteSent – LastByteAcked)
MaxSendBuffer >= LastByteWritten - LastByteAcked



Retransmission Timeout

- If haven't received ack by timeout, retransmit packet after last acked packet
- How to set timeout?

Timing Illustration



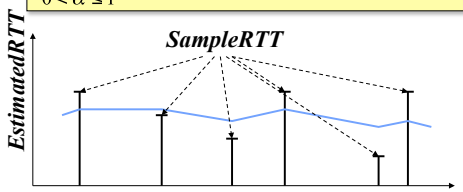
Retransmission Timeout (cont'd)

- If haven't received ack by timeout, retransmit packet after last acked packet
- How to set timeout?
  - Too long: connection has low throughput
  - Too short: retransmit packet that was just delayed
    - » Packet was probably delayed because of congestion
    - » Sending another packet too soon just makes congestion worse
- Solution: make timeout proportional to RTT

RTT Estimation

- Use exponential averaging:

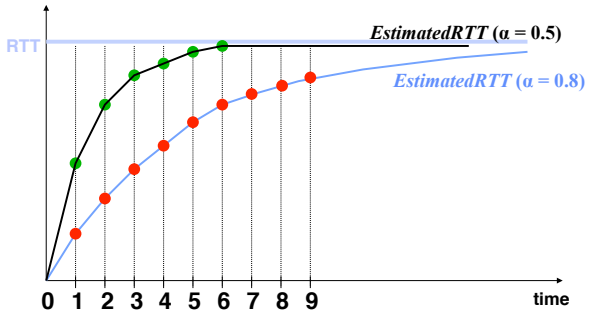
$$SampleRTT = AckRcvdTime - SendPacketTime$$
$$EstimatedRTT = \alpha \times EstimatedRTT + (1-\alpha) \times SampleRTT$$
$$TimeOut = 2 \times EstimatedRTT$$
$$0 < \alpha \leq 1$$



Exponential Averaging Example

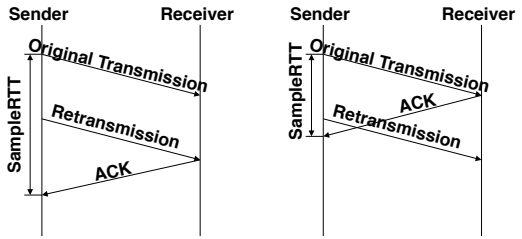
$$EstimatedRTT = \alpha * EstimatedRTT + (1-\alpha) * SampleRTT$$

Assume RTT is constant → *SampleRTT* = RTT



Problem

- How to differentiate between the real ACK, and ACK of the retransmitted packet?



Karn/Partridge Algorithm

- Measure *SampleRTT* only for original transmissions
- Exponential backoff → for each retransmission, double *EstimatedRTT*

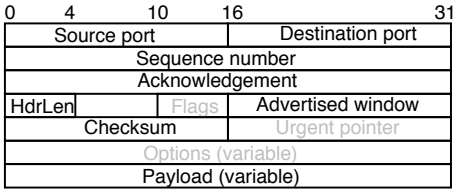
Jacobson/Karels Algorithm

- Problem: exponential average is not enough
  - One solution: use standard deviation (requires expensive square root computation)
  - Use mean deviation instead

$$Difference = SampleRTT - EstimatedRTT$$
$$EstimatedRTT = EstimatedRTT + \delta \times Difference$$
$$Deviation = Deviation + \delta \times (|Difference| - Deviation)$$
$$TimeOut = \mu \times EstimatedRTT + \phi \times Deviation$$
$$0 < \delta \leq 1$$
$$\mu = 1$$
$$\phi = 4$$

TCP Header

- Sequence number, acknowledgement, and advertised window – used by sliding-window based flow control
- HdrLen: TCP header length in 4-byte words
- Checksum: checksum of TCP header + payload



Summary

- Reliable transmission
  - S&W not efficient → Go-Back-n
  - What to ACK? (cumulative, …)
- UDP: Multiplex, detect errors
- TCP: Reliable Byte Stream
  - 3-way handshaking
  - Flow control
  - Timer Value: based on measured RTT