Transport Protocols

Readings: 5.1, & 5.2
UDP - User Datagram Protocol
TCP - Transmission Control Protocol
Goals for Today’s Lecture

• Principles underlying transport-layer services
  – (De)multiplexing
  – Detecting corruption
  – Reliable delivery
  – Flow control

• (Primary) Transport-layer protocols in the Internet
  – User Datagram Protocol (UDP)
  – Transmission Control Protocol (TCP)
Layer Encapsulation in

User A

Application

App-to-app channels

Host-to-host connectivity

Link hardware

User B

Get index.html

Connection ID

Source/Destination

Link Address

2/23/14
IP Suite In Action: Where We Are!!

Host
- HTTP
  - TCP
    - IP
      - Ethernet interface

Router
- IP packet

Transport

Network
- IP, ICMP

Media
- Ethernet, TR
- DHCP, ARP

Application
- SNMP, DNS

2/23/14

mytransport

CS125 - myrouting
IP Suite In Action: End Hosts vs. Routers

HTTP message

TCP segment/data gram

mytransport
End-to-End Protocols

• Working our way up and down Protocol Stack
  – Directly connected networks
  – Switching, Routing and Forwarding
  – Internetworking: IP, Bridges, End-to-End
    • Move from host-to-host
    • To Process-to-process
  – Applications: SNMP, DNS (Top down)
• Transport next layer
Roles of Layers

• Application layer
  – Communication for specific applications
  – E.g., HyperText Transfer Protocol (HTTP), File Transfer Protocol (FTP), Network News Transfer Protocol (NNTP), DNS, SNMP

• Transport layer
  – Communication between processes (e.g., socket)
  – Relies on network layer and serves the application layer
  – E.g., TCP and UDP

• Network layer
  – Logical communication between nodes
  – Hides details of the link technology
  – E.g., IP
Transport Protocols

- Provide *logical communication* between application processes running on different hosts
- Run on end hosts
  - Sender: breaks application messages into *segments*, and passes to network layer
  - Receiver: reassembles segments into messages, passes to application layer
- Multiple transport protocol available to applications
  - Internet: TCP and UDP — best known
End-to-End Protocols

• Underlying best-effort network (IP)
  – Drops messages
  – re-orders messages … different routes, costs
  – delivers duplicate copies of a given message
  – limits messages to some finite size - MTU
  – delivers messages after an arbitrarily long delay

• Common end-to-end services (desired by application)
  – guarantee message delivery
  – deliver messages in the same order they are sent
  – deliver at most one copy of each message
  – support arbitrarily large messages
  – support synchronization
  – allow the receiver to flow control the sender
  – support multiple application processes on each host
  – SECURITY
Internet Transport Protocols

- Datagram messaging service (UDP)
  - No-frills extension of “best-effort” IP
- Reliable, in-order delivery (TCP)
  - Connection set-up
  - Discarding of corrupted packets
  - Retransmission of lost packets
  - Flow control
  - Congestion control (next lecture)
- Other services not available
  - Delay guarantees
  - Bandwidth guarantees
  - Security
Multiplexing and Demultiplexing

- Host receives IP datagrams
  - Each datagram has source and destination IP address,
  - Each datagram carries one transport-layer segment
  - Each segment has source and destination port number
- Host uses IP addresses and port numbers to direct the segment to appropriate socket

TCP/UDP segment format:

- 32 bits
- source port #
- dest port #
- other header fields
- application data (message)
UDP - Unreliable Message Delivery Service

- Lightweight communication between processes
  - Avoid overhead and delays of ordered, reliable delivery
  - Send messages to and receive them from a socket
- User Datagram Protocol (UDP)
  - IP plus port numbers to support (de)multiplexing
  - Optional error checking on the packet contents
Simple Demultiplexor (UDP)

- Unreliable and unordered datagram service
- No flow control, No Acks, No msg ordering, No feedback
- Endpoints identified by ports
  - servers have *well-known* ports
  - see `/etc/services` on Unix
- Header format

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SrcPort</td>
<td>DstPort</td>
<td></td>
</tr>
<tr>
<td>Length</td>
<td>Checksum</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data</td>
</tr>
</tbody>
</table>
```

- Optional checksum
  - pseudo header (Source, Destination, Protocol) + UDP header + data
- Application - accepts full responsibility for all other issues
Why Would Anyone Use UDP?

• Finer control over what data is sent and when
  – As soon as an application process writes into the socket
  – … UDP will package the data and send the packet

• No delay for connection establishment
  – UDP just blasts away without any formal preliminaries
  – … which avoids introducing any unnecessary delays

• No connection state
  – No allocation of buffers, parameters, sequence #s, etc.
  – … making it easier to handle many active clients at once

• Small packet header overhead
  – UDP header is only eight-bytes long
Popular Applications That Use UDP

• Multimedia streaming
  – Retransmitting lost/corrupted packets is not worthwhile ??
  – By the time the packet is retransmitted, it’s too late
  – E.g., telephone calls, video conferencing, gaming

• Simple query protocols like Domain Name System
  – Overhead of connection establishment is overkill
  – Easier to have application retransmit if needed

“Address for www.cnn.com?”

“12.3.4.15”
UDP PseudoHeader

• Issue - Message Correctness
  – Only have ports in the UDP header, yet the message is directed to an IP address
  – UDP wants to make sure that the message is directed to this host and this port

• PseudoHeader
  – Used in calculation of the checksum
  – Includes Source and Destination IP addresses (from IP header)
  – Also the type field
  – Without reading the RFC, not sure if it is prepended or appended
  – Also some books say UDP length field is used twice in the pseudoheader – read RFC if want to know truth
  – UDP checksum, optional in v4, required in v6
Reliable Byte-Stream (TCP)

Outline
- Connection Establishment/Termination
- Sliding Window
- Flow Control
- Adaptive Timeout
- Extensions
TCP Overview

- Connection-oriented
- Byte-stream
  - app writes bytes
  - TCP sends segments
  - app reads bytes
- Full duplex
- Flow control: keep sender from overrunning receiver
- Congestion control: keep sender from overrunning network
Challenges of Reliable Data Transfer

• Over a perfectly reliable channel
  – All of the data arrives in order, just as it was sent
  – Simple: sender sends data, and receiver receives data

• Over a channel with bit errors
  – All of the data arrives in order, but some bits corrupted
  – Receiver detects errors and says “please repeat that”
  – Sender retransmits the data that were corrupted

• Over a lossy channel with bit errors
  – Some data are missing, and some bits are corrupted
  – Receiver detects errors but cannot always detect loss
  – Sender must wait for acknowledgment (“ACK” or “OK”)
  – … and retransmit data after some time if no ACK arrives
TCP Segment

- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes on an Ethernet
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header is typically 20 bytes long
- TCP segment
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream
## Segment Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>SrcPort</td>
<td>0</td>
</tr>
<tr>
<td>DstPort</td>
<td>4</td>
</tr>
<tr>
<td>SequenceNum</td>
<td>10</td>
</tr>
<tr>
<td>Acknowledgment</td>
<td>16</td>
</tr>
<tr>
<td>HdrLen</td>
<td>20</td>
</tr>
<tr>
<td>Flags</td>
<td>24</td>
</tr>
<tr>
<td>AdvertisedWindow</td>
<td>28</td>
</tr>
<tr>
<td>Checksum</td>
<td>32</td>
</tr>
<tr>
<td>UrgPtr</td>
<td>36</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>

---

2/23/14  mytransport
TCP Support for Reliable Delivery

- **Checksum**
  - Used to detect corrupted data at the receiver
  - …leading the receiver to drop the packet

- **Sequence numbers**
  - Used to detect missing data
  - ... and for putting the data back in order

- **Retransmission**
  - Sender retransmits lost or corrupted data
  - Timeout based on estimates of round-trip time
  - Fast retransmit algorithm for rapid retransmission
TCP “Stream of Bytes” Service
...Emulated Using TCP “Segments”

Segment sent when:
1. Segment full (Max Segment Size),
2. Not full, but times out, or
3. “Pushed” by application.
Segment Format (cont)

• Each connection identified with 4-tuple:
  - \((\text{SrcPort}, \text{SrcIPAddr}, \text{DsrPort}, \text{DstIPAddr})\)

• Sliding window + flow control
  - acknowledgment, SequenceNum, AdvertisedWindow

• Flags
  - SYN, FIN, RESET, PUSH, URG, ACK

• Checksum
  - pseudo header + TCP header + data
Segment Transmission

- MSS - Maximum Segment Size, ~ MTU (-IP Hdrs)
- Push - Sending application/process “pushes” segment, do not wait for full segment, e.g. telnet
- Timer - periodically fires “keep alive”
Sequence Numbers

Host A

ISN (initial sequence number)

Sequence number = 1st byte

TCP Data

TCP HDR

ACK sequence number = next expected byte

Host B

TCP Data

TCP HDR

2/23/14
Initial Sequence Number (ISN)

• Sequence number for the very first byte
  – E.g., Why not a de facto ISN of 0?
• Practical issue
  – IP addresses and port #s uniquely identify a connection
  – Eventually, though, these port #s do get used again
  – … and there is a chance an old packet is still in flight
  – … and might be associated with the new connection
• So, TCP requires changing the ISN over time
  – Set from a 32-bit clock that ticks every 4 microseconds
  – … which only wraps around once every 4.55 hours!
• But, this means the hosts need to exchange ISNs
TCP Three-Way Handshake
State Transition Diagram

CLOSED
- Passive open
- Close

LISTEN
- Close

SYN_RCVD
- SYN/SYN + ACK
- SYN/SYN + ACK

SYN_SENT
- Send/SYN

ESTABLISHED
- SYN + ACK/ACK
- ACK

FIN_WAIT_1
- Close/FIN
- ACK

FIN_WAIT_2
- FIN/ACK

CLOSING
- FIN/ACK
- Ack
- Timeout after two segment lifetimes

CLOSE_WAIT
- Close/FIN

LAST_ACK
- ACK

TIME_WAIT
- ACK

CLOSED
- Closed
Connection Establishment and Termination

A: Active participant (client)

B: Passive participant (server)

SYN, SequenceNum ≠

SYN+ACK, SequenceNum=y,

Acknowledgment =x+1

ACK, Acknowledgment =y+1
### TCP Header

<table>
<thead>
<tr>
<th>Flags:</th>
<th>SYN</th>
<th>FIN</th>
<th>RST</th>
<th>PSH</th>
<th>URG</th>
<th>ACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source port</td>
<td>Destination port</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>0</td>
<td>Flags</td>
<td>Advertised window</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Flags:**
- SYN
- FIN
- RST
- PSH
- URG
- ACK
Step 1: A’s Initial SYN Packet

<table>
<thead>
<tr>
<th>Flags:</th>
<th>SYN</th>
<th>FIN</th>
<th>RST</th>
<th>PSH</th>
<th>URG</th>
<th>ACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>A’s port</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B’s port</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A’s Initial Sequence Number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Advertised window</td>
<td>20</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flags</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Urgent pointer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A tells B it wants to open a connection…
Step 2: B’s SYN-ACK Packet

<table>
<thead>
<tr>
<th>Flags:</th>
<th>B’s port</th>
<th>A’s port</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYN</td>
<td>B’s Initial Sequence Number</td>
<td></td>
</tr>
<tr>
<td>FIN</td>
<td>A’s ISN plus 1</td>
<td></td>
</tr>
<tr>
<td>RST</td>
<td>20</td>
<td>Flags</td>
</tr>
<tr>
<td>PSH</td>
<td>0</td>
<td>Advertised window</td>
</tr>
<tr>
<td>URG</td>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>ACK</td>
<td>Options (variable)</td>
<td></td>
</tr>
</tbody>
</table>

B tells A it accepts, and is ready to hear the next byte…

… upon receiving this packet, A can start sending data
Step 3: A’s ACK of the SYN-ACK

<table>
<thead>
<tr>
<th>Flags:</th>
<th>SYN</th>
<th>FIN</th>
<th>RST</th>
<th>PSH</th>
<th>URG</th>
<th>ACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B’s ISN plus 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Advertised window</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Urgent pointer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A tells B it wants is okay to start sending...

... upon receiving this packet, B can start sending data
What if the SYN Packet Gets Lost?

- Suppose the SYN packet gets lost
  - Packet is lost inside the network, or
  - Server rejects the packet (e.g., listen queue is full)
- Eventually, no SYN-ACK arrives
  - Sender sets a timer and waits for the SYN-ACK
  - … and retransmits the SYN if needed
- How should the TCP Sender set the timer?
  - Sender has no idea how far away the receiver is
  - Hard to guess a reasonable length of time to wait
  - Some TCPs use a default of 3 or 6 seconds - ATTACK
SYN Loss and Web Downloads

• User clicks on a hypertext link
  – Browser creates a socket and does a “connect”
  – The “connect” triggers the OS to transmit a SYN
• If the SYN is lost…
  – The 3-6 seconds of delay may be very long
  – The user may get impatient
  – … and click the hyperlink again, or click “reload”
• User triggers an “abort” of the “connect”
  – Browser creates a new socket and does a “connect”
  – Essentially, forces a faster send of a new SYN packet!
  – Sometimes very effective, and the page comes fast
TCP Sliding Window
Motivation for Sliding Window

• Stop-and-wait is inefficient
  – Only one TCP segment is “in flight” at a time
  – Especially bad when delay-bandwidth product is high
• Numerical example
  – 1.5 Mbps link with a 45 msec round-trip time (RTT)
    • Delay-bandwidth product is 67.5 Kbits (or 8 KBytes)
  – But, sender can send at most one packet per RTT
    • Assuming a segment size of 1 KB (8 Kbits)
    • … leads to 8 Kbits/segment / 45 msec/segment \(\Rightarrow\) 182 Kbps
    • That’s just one-eighth of the 1.5 Mbps link capacity
TCP Sliding Window
Data Link Versus Transport

• Potentially connects many different hosts
  – need explicit connection establishment and termination
• Potentially different RTT
  – need adaptive timeout mechanism
• Potentially long delay in network
  – need to be prepared for arrival of very old packets
• Potentially different capacity at destination
  – need to accommodate different node capacity
• Potentially different network capacity
  – need to be prepared for network congestion
Sliding Window

- Allow a larger amount of data “in flight”
  - Allow sender to get ahead of the receiver
  - … though not too far ahead
Receiver Buffering

- Window size
  - Amount that can be sent without acknowledgment
  - Receiver needs to be able to store this amount of data
- Receiver advertises the window to the Sender
- Receiver tells the Sender the amount of free space left
  - … and the Sender agrees not to exceed this amount
TCP Header for Receiver Buffering

<table>
<thead>
<tr>
<th>Flags:</th>
<th>SYN</th>
<th>FIN</th>
<th>RST</th>
<th>PSH</th>
<th>URG</th>
<th>ACK</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Sequence number</th>
<th>Acknowledgment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>HdrLen</th>
<th>Flags</th>
<th>Checksum</th>
<th>Urgent pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Options (variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Flags:
- SYN
- FIN
- RST
- PSH
- URG
- ACK

Options (variable)
Sliding Window Revisited

- **Sending side**
  - \( \text{LastByteAcked} \leq \text{LastByteSent} \)
  - \( \text{LastByteSent} \leq \text{LastByteWritten} \)
  - buffer bytes between \( \text{LastByteAcked} \) and \( \text{LastByteWritten} \)

- **Receiving side**
  - \( \text{LastByteRead} < \text{NextByteExpected} \)
  - \( \text{NextByteExpected} \leq \text{LastByteRcvd} + 1 \)
  - buffer bytes between \( \text{LastByteRead} \) and \( \text{LastByteRcvd} \)
Flow Control

- Send buffer size: \textbf{MaxSendBuffer}
- Receive buffer size: \textbf{MaxRcvBuffer}
- Receiving side
  - $\text{LastByteRcvd} - \text{LastByteRead} \leq \text{MaxRcvBuffer}$
  - $\text{AdvertisedWindow} = \text{MaxRcvBuffer} - (\text{NextByteExpected} - \text{NextByteRead})$
- Sending side
  - $\text{LastByteSent} - \text{LastByteAcked} \leq \text{AdvertisedWindow}$
  - $\text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{LastByteSent} - \text{LastByteAcked})$
  - $\text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer}$
  - block sender if $(\text{LastByteWritten} - \text{LastByteAcked}) + y > \text{MaxSenderBuffer}$
- Always send ACK in response to arriving data segment
- Persist when $\text{AdvertisedWindow} = 0$
Silly Window Syndrome

- How aggressively does sender exploit open window?
- Because Ack indicates message size, a short message continues to always be short, Ack tells sender there is that much space available.

- Receiver-side solutions
  - after advertising zero window, wait for space equal to a maximum segment size (MSS) otherwise many small packets
  - delayed acknowledgements
Nagle’s Algorithm

• How long does sender delay sending data?
  – too long: hurts interactive applications
  – too short: poor network utilization
  – strategies: timer-based vs self-clocking

• When application generates additional data
  – if new data fills a max segment (and window open): send it
  – Else //partially filled buffer
    • if there is unack’ed data in transit: buffer it until ACK arrives
    • else: send it
Protection Against Wrap Around

- 32-bit `SequenceNum`

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>Time Until Wrap Around</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 (1.5 Mbps)</td>
<td>6.4 hours</td>
</tr>
<tr>
<td>Ethernet (10 Mbps)</td>
<td>57 minutes</td>
</tr>
<tr>
<td>T3 (45 Mbps)</td>
<td>13 minutes</td>
</tr>
<tr>
<td>FDDI (100 Mbps)</td>
<td>6 minutes</td>
</tr>
<tr>
<td>STS-3 (155 Mbps)</td>
<td>4 minutes</td>
</tr>
<tr>
<td>STS-12 (622 Mbps)</td>
<td>55 seconds</td>
</tr>
<tr>
<td>STS-24 (1.2 Gbps)</td>
<td>28 seconds</td>
</tr>
<tr>
<td>OC48 (2.4Gbps)</td>
<td>14 seconds</td>
</tr>
</tbody>
</table>
Keeping the Pipe Full

- 16-bit **AdvertisedWindow**

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>Delay x Bandwidth Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 (1.5 Mbps)</td>
<td>18KB</td>
</tr>
<tr>
<td>Ethernet (10 Mbps)</td>
<td>122KB</td>
</tr>
<tr>
<td>T3 (45 Mbps)</td>
<td>549KB</td>
</tr>
<tr>
<td>FDDI (100 Mbps)</td>
<td>1.2MB</td>
</tr>
<tr>
<td>STS-3 (155 Mbps)</td>
<td>1.8MB</td>
</tr>
<tr>
<td>STS-12 (622 Mbps)</td>
<td>7.4MB</td>
</tr>
<tr>
<td>STS-24 (1.2 Gbps)</td>
<td>14.8MB</td>
</tr>
<tr>
<td>OC48 (2.4Gbps)</td>
<td>29.6MB</td>
</tr>
</tbody>
</table>

assuming 100ms RTT
TCP Retransmissions
Automatic Repeat reQuest (ARQ)

- **Automatic Repeat Request**
  - Receiver sends acknowledgment (ACK) when it receives packet
  - Sender waits for ACK and timeouts if it does not arrive within some time period

- **Simplest ARQ protocol**
  - Stop and wait
  - Send a packet, stop and wait until ACK arrives
Reasons for Retransmission

1. Packet lost
2. ACK lost, DUPPLICATE PACKET
3. Early timeout, DUPPLICATE PACKETS
How Long Should Sender Wait?

• Sender sets a timeout to wait for an ACK
  – Too short: wasted retransmissions
  – Too long: excessive delays when packet lost
• TCP sets timeout as a function of the RTT (Dynamic)
  – Expect ACK to arrive after an RTT
  – … plus a fudge factor to account for queuing
• But, how does the sender know the RTT?
  – Can estimate the RTT by watching the ACKs
  – Smooth estimate: keep a running average of the RTT
    • EstimatedRTT = a * EstimatedRTT + (1 –a ) * SampleRTT
  – Compute timeout: TimeOut = 2 * EstimatedRTT
Example RTT Estimation

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr

2/23/14
A Flaw in This Approach

• An ACK doesn’t really acknowledge a transmission
  – Rather, it acknowledges receipt of the data
• Consider a retransmission of a lost packet
  – If you assume the ACK goes with the 1st transmission
  – … the SampleRTT comes out way too large
• Consider a duplicate packet
  – If you assume the ACK goes with the 2nd transmission
  – … the Sample RTT comes out way too small
• Simple solution in the Karn/Partridge algorithm
  – Only collect samples for segments sent one single time
Yet Another Limitation…

- Does NOT consider variance in the RTT
  - If variance is small, the EstimatedRTT is pretty accurate
  - … but, if variance is large, the estimate is NOT good
- Better to directly consider the variance
  - Consider difference: SampleRTT – EstimatedRTT
  - Boost the estimate based on the difference
- Jacobson/Karels algorithm
  - See Peterson/Davie book for details
History: Adaptive Retransmission (Original Algorithm)

- Measure $\text{SampleRTT}$ for each segment / ACK pair
- Compute weighted average of RTT
  - $\text{EstRTT} = \alpha \times \text{EstRTT} + \beta \times \text{SampleRTT}$
  - where $\alpha + \beta = 1$
  - $\alpha$ between 0.8 and 0.9
  - $\beta$ between 0.1 and 0.2
- Set timeout based on $\text{EstRTT}$
  - $\text{TimeOut} = 2 \times \text{EstRTT}$
History: Karn/Partridge Algorithm

- Do not sample RTT when retransmitting
- Double timeout after each retransmission
History: Jacobson/ Karels Algorithm

- New Calculations for average RTT
- \( \text{Diff} = \text{SampleRTT} - \text{EstRTT} \)
- \( \text{EstRTT} = \text{EstRTT} + (\delta \times \text{Diff}) \)
- \( \text{Dev} = \text{Dev} + \delta (|\text{Diff}| - \text{Dev}) \)
  - where \( \delta \) is a factor between 0 and 1
- Consider variance when setting timeout value
- \( \text{TimeOut} = \mu \times \text{EstRTT} + \phi \times \text{Dev} \)
  - where \( \mu = 1 \) and \( \phi = 4 \)
- Notes
  - algorithm only as good as granularity of clock (500ms on Unix)
  - accurate timeout mechanism important to congestion control (later)
TCP Extensions

• Implemented as header options
• Store timestamp in outgoing segments
• Extend sequence space with 32-bit timestamp (PAWS)
• Shift (scale) advertised window
TCP Issues

- Messages vs Byte Stream
- Record Boundaries…Database Records
- Solution
  - UrgPtr -- Sending app/process can tell receiver that there is urgent data, i.e., record boundary
  - Push -- app/process can tell TCP “send now”…TCP receiver must inform receiving app…force records
TCP Issues

- TCP is reliable byte stream…what about
  - Reliable, Ordered, etc: Request/Reply protocol
  - Unreliable byte stream: video
- TCP - High overhead to setup/close connection
  - Web experience
- TCP - Window Based
  - How about Rate based…
- TCP - Byte based…not message based
State Transition Diagram

- **CLOSED**
  - Passive open
  - Close
- **LISTEN**
  - SYN/SYN + ACK
- **SYN_RCVD**
  - SYN/SYN + ACK
  - ACK
  - Close/FIN
- **SYN_SENT**
  - Send/SYN
  - SYN + ACK/ACK
  - ACK
- **ESTABLISHED**
  - FIN/ACK
  - Close/FIN
- **FIN_WAIT_1**
  - FIN/ACK
  - ACK
  - ACK + FIN/ACK
- **FIN_WAIT_2**
  - FIN/ACK
  - ACK
- **CLOSING**
  - Timeout after two segment lifetimes
  - ACK
- **TIME_WAIT**
  - FIN/ACK
- **CLOSE_WAIT**
  - Close/FIN
- **LAST_ACK**
  - ACK
- **CLOSED**