CSE 123A
Computer Networks
Fall 2005
Lecture 8
Reliable Communication,
Connections and Flow Control

Some images courtesy David Wetherall and Van Jacobsen
Project #1

- Assignment #1 (solo project)
  - In C
  - Program to generate CRC for bitstream
  - Program to check whether CRC for bitstream is correct
  - Using a binary polynomial that I specify
  - Electronic turnin by Nov 1 at 1pm

- Currently faculty server is hosed, so actual assignment will be posted as soon as its up (tonight or tomorrow morning)
Today’s class

- We begin on the transport layer
  - Builds on the services of the Network layer
  - Communication between processes on hosts
- Principle focus
  - How do we ensure that a message is reliably communicated from one host to another?
- Topics
  - Automatic Repeat reQuest (ARQ)
  - Sliding windows
  - Retransmission timers
  - Connections & Flow control
Thought experiment

- You want to send a long letter to your friend
  - The only medium available to either of you is *postcards*
  - Postcards get lost in the mail, delayed, damaged

- How do you ensure that your friend receives the letter?
Reliable Transmission

- The data networking version of the problem
  - How do we reliably send a message when packets can be lost/corrupted in the network?

- Two options
  - Detect a loss/corruption and retransmit
  - Send data redundantly to tolerate loss/corruption
Automatic Repeat Request (ARQ)

- Acknowledgments (ACKs) and retransmissions after a timeout
- ARQ is generic name for protocols based on this strategy
The Need for Sequence Numbers

- In the case of ACK loss (or poor choice of timeout) the receiver can’t distinguish this message from the next
- Need to understand how many packets can be outstanding and number the packets; here, a single bit will do
Stop-and-Wait

- Only one outstanding packet at a time
- Also called alternating bit protocol in the book
How does receiver recognize a duplicate?

- Sequence # in packet is finite

- How many bits do we need?
  - One bit for stop and wait
  - Won’t send seq #1 until receive ACK for seq #0
  - Only allows one packet in flight
What if packets are delayed?

- Never reuse a seq #?
  Finite… really big #?
- Require in order delivery?
- Prevent very late delivery?
  - TTL: Decrement hop count per packet, discard if exceeded
  - Seq #s not reused within delay bound
What happens if a machine crashes?

- How do we distinguish packets sent before and after reboot? Which seq# to use?

- Solutions
  - Restart sequence # at 0?
  - Assume reboot is greater than max delay bound?
  - Choose seq # at random and hope it works out?
  - Use stable storage (disk) to store recent sequence # and increment high bits of seq # on every boot

- Reality: People don’t worry about this
  - Slow reboots, explicit connection management, tolerant users
Performance Limitations of Stop-and-Wait

- Lousy performance if xmit 1 pkt << prop. delay
  - How bad?
- Want to utilize all available bandwidth
  - Need to keep more data “in flight”
  - How much? Remember the bandwidth-delay product?
- Also limited by quality of timeout (how long?)
Pipelined transmission

- Faster, reliable delivery:
  - Send multiple packets without waiting for the 1st to be ACKed (each with own seq#)
  - Send new packet after each ACK
  - Sender keeps list of unACK’ed packets and resends after timeout
  - Receiver same as stop & wait
- What if packet #2 keeps being lost?
  - Receiver must buffer all packets after 2
  - Potential buffer overflow
- What if sender can send faster than receiver can receive?
Sliding Window

- Single mechanism that supports:
  - Multiple outstanding packets
  - Reliable delivery
  - In-order delivery
  - Flow control

- At the core of all modern ARQ protocols
Sliding Window – Sender

- Window bounds *outstanding* unACKed data
  - Implies need for buffering at sender
- “Last” ACK applies to in-order data
- What to do on a timeout?
  - Go-Back-N: one timer, send all unacknowledged data on timeout
  - Selective Repeat: timer per packet, resend as needed
Sliding Window – Receiver

- Receiver buffers too:
  - data may arrive out-of-order
  - or faster than can be consumed (flow control)

- Receiver ACK choices:
  - Individual, Cumulative (TCP), Selective (newer TCP), Negative
Deciding When to Retransmit

- How do you know when a packet has been lost?
  - Ultimately sender uses timers to decide when to retransmit
- But how long should the timer be?
  - Too long: inefficient (large delays, poor use of bandwidth)
  - Too short: may retransmit unnecessarily (causing extra traffic)
- Right timer is based on the round trip time (RTT)
  - Which can vary greatly (propagation and queuing)
A Simple Network Model

- Buffers at routers used to absorb bursts when input rate > output
- Loss (drops) occur when sending rate is persistently > drain rate
Effects of Early Retransmissions (early TCP)

What happened

Ideal case
Congestion Collapse

- In the limit, early retransmissions lead to congestion collapse
  - Sending more packets into the network when it is overloaded exacerbates the problem of congestion
  - Network stays busy but very little useful work is being done
- This happened in real life ~1987
  - Led to Van Jacobson’s TCP algorithms, which form the basis of congestion control in the Internet today
    » We’ll cover in depth in a later class
Estimating Round-trip Times (RTTs)

- Idea: Adapt based on recent past measurements

- Simple algorithm:
  - For each packet, note time sent and time ack received
  - Compute RTT samples and average recent samples for timeout

- EstimatedRTT = $\alpha \times \text{EstimatedRTT} + (1 - \alpha) \times \text{SampleRTT}$
- This is an exponentially-weighted moving average that smoothes the samples. Typically, $\alpha = 0.8$ to 0.9.
- Set timeout to small multiple (2) of the estimate to capture variation around mean.
Estimated Retransmit Timer

Actual RTT

Estimated RTT
Karn/Partridge Algorithm

- Problem: RTT for retransmitted packets ambiguous
- Solution: Don’t measure RTT for retransmitted packets and do not relax backed of timeout until valid RTT measurements
Jacobson/Karels Algorithm

- Problem:
  - Variance in RTTs gets large as network gets loaded
  - So an average RTT isn’t a good predictor when we need it most

- Solution: Track variance too.
  - Difference = SampleRTT – EstimatedRTT
  - EstimatedRTT = EstimatedRTT + (δ x Difference)
  - Deviation = Deviation + δ(|Difference| - Deviation)
  - Timeout = μ x EstimatedRTT + φ x Deviation
  - In practice, δ = 1/8, μ = 1 and φ = 4, but timeouts are set as MAX(Timeout, 500ms)

- Key idea: timeout reflects both mean RTT and variance in RTT
  - Small variance: Timeout=RTT
  - Large variance: Timeout dominated by deviation term
Estimate with Mean + Variance

![Graph showing Actual RTT and Estimated RTT](image)
Can we shortcut the timeout?

- Timeout is long in practice
- If packets are usually in order then out-of-order packets imply that a packet was lost
  - **Negative ACK**
    » Receiver requests missing packet
  - **Fast retransmit**
    » Receiver ACKs out-of-order packets with seq# of last *contiguous* packet
    » When sender receives multiple *duplicate* acknowledgements resends missing packet
Fast retransmit
Alternatives to retransmission?

- Redundancy
  - Send additional data to compensate for lost packets
- Why not use retransmission
  - Multicast
    - Lots of receivers
      - If each one ACK/NAK then hard to scale
        - Lots of messages
        - Lots of state
    - Heterogeneous receivers
      - Modem vs 100MBps connected hosts
    - One-way or very long delay channels (spacecraft)
Simplest version

- Send every packet twice
- Must lose both packets in a pair to prevent message from being delivered
**Generalization: Forward Error Correction (FEC)**

- Use erasure codes to redundantly encode $k$ source packets into $k \times m$ encoded packets
  - Reed Solomon Codes
  - Tornado codes
- Multicast/broadcast encoded packets continually
- Any receiver can reconstruct message from any $k$ packets in the set of $k \times m$
Sometimes referred to as a “Digital Fountain”
Pros and Cons of Forward Error Correction

- **Pro**
  - Every packet can be useful for all clients
  - Well suited to multicast situation

- **Con**
  - Sends more data than ideally necessary
  - Need large block sizes for efficiency
Connections & Flow control

- Ok, it's nice that you have a reliable protocol, but there's more to it
  - Some protocol details
    - User Datagram Protocol (UDP)
    - Transmission Control Protocol (TCP)
  - Connection-oriented vs connection-less transport
    - Naming
    - Connection setup
    - Connection teardown
  - Flow control
    - How do we manage buffering at the endpoints?
Naming Processes/Services

- Process here is an abstract term for your Web browser (HTTP), Email servers (SMTP), hostname translation (DNS)
- How do we identify for remote communication?
  - Process id or memory address are OS-specific and transient
- So TCP and UDP use Ports
  - 16-bit integers representing mailboxes that processes “rent”
  - Identify process uniquely as (IP address, protocol, port)
Picking Port Numbers

- We still have the problem of allocating port numbers
  - What port should a Web server use on host X?
  - To what port should you send to contact that Web server?

- Servers typically bind to “well-known” port numbers
  - e.g., HTTP 80, SMTP 25, DNS 53, … look in /etc/services
  - Ports below 1024 traditionally reserved for “well-known” services

- Clients use OS-assigned temporary (ephemeral) ports
  - Above 1024, recycled by OS when client finished
Transmission Control Protocol (TCP)

- Reliable **bi-directional** bytestream between processes
  - Message boundaries are not preserved
- Connection-oriented
  - Conversation between two endpoints with beginning and end
- Flow control (later)
  - Prevents sender from over-running receiver buffers
- Congestion control (next class)
  - Prevents sender from over-running network buffers
TCP Delivery

Application process

TCP
Send buffer

Write bytes

TCP
Receive buffer

Read bytes

Transmit segments

Segment Segment ⋯ Segment
TCP Header Format

- Ports plus IP addresses identify a connection

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>SrcPort</td>
<td>0-4</td>
</tr>
<tr>
<td>DstPort</td>
<td>4-9</td>
</tr>
<tr>
<td>SequenceNum</td>
<td>9-13</td>
</tr>
<tr>
<td>Acknowledgment</td>
<td>13-17</td>
</tr>
<tr>
<td>HdrLen</td>
<td>Flags</td>
</tr>
<tr>
<td>Checksum</td>
<td>UrgPtr</td>
</tr>
<tr>
<td>Data</td>
<td>25-31</td>
</tr>
</tbody>
</table>
TCP Header Format

- Sequence, Ack numbers used for the sliding window
  - How big a window? Flow control/congestion control determine

```
0  4  10  16  31
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SrcPort</td>
<td>DstPort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- SequenceNum
- Acknowledgment
- HdrLen: 0
- Flags
- AdvertisedWindow
- Checksum
- UrgPtr
- Options (variable)
- Data
TCP Header Format

- Flags may be URG, ACK, PSH, RST, SYN, FIN
Connection Establishment

- Both sender and receiver must be ready before we start to transfer the data
  - Sender and receiver need to agree on a set of parameters
  - Most important: sequence number space in each direction
  - Lots of other parameters: e.g., the Maximum Segment Size

- Handshake protocols: setup state between two oblivious endpoints
Two-way handshake?

Active participant (client)

Passive participant (server)

SYN, SequenceNum = x

SYN, SequenceNum = y

+data

What’s wrong here?
Two-way handshake?

Active participant (client)

Old SYN, SequenceNum = x

New SYN, SequenceNum = q

SYN, SequenceNum = y

+data

Passive participant (server)

Delayed old SYN

Rejected
Three-Way Handshake

- Opens both directions for transfer

Active participant (client)  Passive participant (server)

SYN, SequenceNum = x

SYN + ACK, SequenceNum = y, Acknowledgment = x + 1

ACK, Acknowledgment = y + 1

+data
Some Comments

- We could abbreviate this setup, but it was chosen to be robust, especially against delayed duplicates
  - Three-way handshake from Tomlinson 1975
- Choice of changing initial sequence numbers (ISNs) minimizes the chance of hosts that crash getting confused by a previous incarnation of a connection
- How to choose ISNs?
  - Maximize period between reuse
  - Minimize ability to guess (why?)
TCP State Transitions

CLOSED
- Passive open
- Close

LISTEN
- Send/SYN
- SYN/SYN + ACK
- SYN + ACK/ACK

SYN_RCVD
- ACK
- Close/FIN

SYN_SENT
- SYN/SYN + ACK
- Active open/SYN

ESTABLISHED
- FIN/ACK
- Close/FIN

FIN_WAIT_1
- ACK
- FIN/ACK

FIN_WAIT_2
- ACK

CLOSING
- Timeout after two segment lifetimes

LAST_ACK
- ACK

TIME_WAIT
- ACK

CLOSE_WAIT
- Close/FIN

CLOSED
- ACK
Again, with States

Active participant
(client)

SYN_SENT

ESTABLISHED

Passive participant
(server)

LISTEN

SYN_RCVD

ESTABLISHED

SYN, SequenceNum = x

SYN + ACK, SequenceNum = y,
Acknowledgment = x + 1

ACK, Acknowledgment = y + 1

+data
Connection Teardown

- Orderly release by sender and receiver when done
  - Delivers all pending data and “hangs up”

- Cleans up state in sender and receiver

- TCP provides a “symmetric” close
  - Both sides shutdown independently
TCP Connection Teardown

Web server

FIN_WAIT_1
FIN
FIN_WAIT_2
ACK
TIME_WAIT
...ACK
CLOSED

Web browser

CLOSE_WAIT
LAST_ACK
CLOSED

FIN
The **TIME_WAIT** State

- We wait 2MSL (two times the maximum segment lifetime of 60 seconds) before completing the close
- Why?

- ACK might have been lost and so FIN will be resent
- Could interfere with a subsequent connection

- Real life: Abortive close
  - Some systems don’t wait for 2*MSL, simply send Reset packet (RST)
  - Why? Frees up resources immediately
Flow Control

- Sender must transmit data no faster than it can be consumed by the receiver
  - Receiver might be a slow machine
  - App might consume data slowly

- Implement by adjusting the size of the sliding window used at the sender based on receiver feedback about available buffer space
  - This is the purpose of the Advertised Window field
TCP Header Format

- Advertised window is used for flow control

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>SrcPort</td>
<td>2 bytes</td>
</tr>
<tr>
<td>DstPort</td>
<td>2 bytes</td>
</tr>
<tr>
<td>SequenceNum</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Acknowledgment</td>
<td>4 bytes</td>
</tr>
<tr>
<td>HdrLen</td>
<td>1 byte</td>
</tr>
<tr>
<td>Flags</td>
<td>1 byte</td>
</tr>
<tr>
<td>Checksum</td>
<td>2 bytes</td>
</tr>
<tr>
<td>UrgPtr</td>
<td>2 bytes</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>variable</td>
</tr>
</tbody>
</table>

Options (variable)
Sender and Receiver Buffering

Sending application

TCP

LastByteWritten

LastByteAcked

LastByteSent

Receiving application

TCP

LastByteRead

NextByteExpected

LastByteRcvd

= available buffer

= buffer in use
Example – Exchange of Packets

Receiver has buffer of size 4 and application doesn’t read

T=1
SEQ=1

T=2
ACK=2; WIN=3

T=3
SEQ=2

T=4
ACK=3; WIN=2

T=5
SEQ=3

T=6
ACK=4; WIN=1

Stall due to flow control here

T=5
ACK=5; WIN=0
Example – Buffer at Sender

| T=1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| T=2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| T=3 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| T=4 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| T=5 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| T=6 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

- □ = acked
- □ = sent
- □ = advertised
Lots of icky details

- Window probes
- Silly Window Syndrome
- Nagel’s algorithm
- PAWS
- Etc…

- Steven’s books “TCP/IP Illustrated (vol 1,2)” is a great source of information on this
Example Icky Detail: Advertised Window Deadlock

- If the receiving process does not empty the buffer (e.g., not scheduled), then the sender fills up the receiver’s buffer
  - Advertised Window is 0
  - Effective Window goes to 0 when all data is ACKed
- Problem: When can the sender start sending again?
  - No timeouts because all data is ACKed
  - No packets from receiver with a new Advertised Window because receiver isn’t running
- Solution: Ping with a segment of 1 byte of data
  - Eventually receiver responds with a new Advert. Window
Misc TCP Header fields

- Header length allows for variable length TCP header with options for extensions such as timestamps, selective acknowledgements, etc.
- Checksum protects TCP header and data
- Urgent pointer/data not used in practice

```
0 4 10 16 31
SrcPort       DstPort
SequenceNum
Acknowledgment
HdrLen 0 Flags AdvertisedWindow
Checksum UrgPtr
Options (variable)
Data
```
TCP applications

- HTTP/WWW
- FTP
- SMTP, POP, IMAP (E-mail)

Why is TCP well suited to these applications?
User Datagram Protocol (UDP)

- Provides **unreliable message delivery** between processes
  - Source port filled in by OS as message is sent
  - Destination port identifies UDP delivery queue at endpoint
- Connectionless (no state about who talks to whom)

```
0   16   31
SrcPort  DstPort
Checksum  Length
```

Data
UDP Delivery

Application process

Application process

Application process

Ports

Message Queues

DeMux

Packets arrive

Kernel boundary
UDP Checksum

- UDP includes optional protection against errors
  - Checksum intended as an end-to-end check on delivery
  - So it covers data, UDP header, and IP pseudoheader
Applications for UDP

- Streaming media
- DNS (Domain Name Service)
- NTP (Network Time Protocol)
- Why is UDP appropriate for these?
Summary

- Transport layer allows processes to communicate with stronger guarantees, e.g., reliability
- Reliability mechanisms
  - ARQ
    - Sliding Window + retransmission for efficiency
    - Retransmission timer must be adaptive
  - FEC
    - In restricted settings
- Connection setup/teardown
- Flow control
  - Adjust sliding window to manage receiver buffer
For next time...

- Congestion Control
- Read Ch 6.3-6.4