The Big(ger) Picture

- application
- end-to-end
- IP
- data link/physical

428/598 topics

detailed description of issues here

most coverage until now

performance → congestion control
Where are you?

• Understand how to
  – Build a network on one physical medium
  – Connect networks
  – Implement a reliable byte stream
  – Address network heterogeneity
  – Address global scale

• Final part of class
  – End-to-end issues and common protocols
  – Congestion control: TCP heuristics, switch/router approaches to fairness
  – Performance analysis
End-to-End Protocols

End-to-end Service Model
Protocol Examples
User Datagram Protocol (UDP)
Transmission Control Protocol (TCP)
End-to-End Service Model

• User perspective of network
  – Knowledge of required functionality
  – Implementation is hidden

• Focus
  – Enable communication between applications
  – Translate from host-to-host protocols

• Services
  – Services that cannot be implemented in lower layers (hop-by-hop basis)
  – Avoid duplicate effort
  – Services not needed by all applications
End-to-End Service Model

- Build on “best effort” service provided by network layer (IP)
  - Messages sent from a host are delivered to another host
    - May be lost
    - May be reordered
    - May be delivered multiple times
    - May be limited to a finite size
    - May be delivered after a long delay
End-to-End Service Model

- Support services needed by the application
  - Multiple connections per host
  - Guaranteed delivery
  - Messages delivered in the order they were sent
  - Messages delivered at most once
  - No limit on message size
  - Synchronization between sender and receiver
  - Flow control
End-to-End Service Model

• Challenge
  – Given
    • Less than desirable properties of the underlying network
  – Create
    • High-level services required by applications

• Services
  – Asynchronous demultiplexing service
  – Reliable byte-stream service
User Datagram Protocol (UDP)

- Simple connectionless demultiplexer
  - No handshaking
  - Each segment handled independently

- Service Model
  - Thin veneer over IP services
  - Unreliable unordered datagram service
  - Addresses multiplexing of multiple connections

- Multiplexing
  - 16-bit port numbers
  - Well-known ports

- Checksum
  - Validate header
  - Optional in IPv4
  - Mandatory in IPv6
**User Datagram Protocol (UDP)**

- Why is there a UDP?
  - No connection establishment
    - Low delay
  - Simple
    - No connection state at sender, receiver
  - Small header
  - No congestion control
    - UDP can blast away as fast as desired

- What kind of applications is UDP good for?
  - Streaming multimedia apps
  - Loss tolerant
  - Rate sensitive

- Other UDP uses
  - DNS, SNMP

- Reliable transfer over UDP
  - At application layer
  - Application-specific error recovery
## UDP Header Format

<table>
<thead>
<tr>
<th>Source Port</th>
<th>Destination Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDP Length</td>
<td>UDP Checksum</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>0-7</td>
</tr>
<tr>
<td>Destination Port</td>
<td>8-15</td>
</tr>
<tr>
<td>UDP Length</td>
<td>16-31</td>
</tr>
<tr>
<td>UDP Checksum</td>
<td>32-31</td>
</tr>
</tbody>
</table>
### UDP Header Format

<table>
<thead>
<tr>
<th>Source Port</th>
<th>Destination Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

- 16-bit source and destination ports
**UDP Header Format**

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>Destination Port</td>
<td>UDP Length</td>
<td>UDP Checksum</td>
</tr>
</tbody>
</table>

- Length includes 8-byte header and data
UDP Header Format

- Checksum
- Uses IP checksum algorithm
  - Computed on header, data and pseudo header
UDP Header Format

- Checksum
  - What purpose does the checksum serve?
  - Why is it mandatory when using IPv6?
Transmission Control Protocol (TCP)

- Reliable byte stream
- Service model
  - Multiple connections per host
  - Guaranteed delivery
  - Messages delivered in the order they were sent
  - Messages delivered at most once
  - No limit on message size
  - Synchronization between sender and receiver
  - Flow control
- Multiplexing
  - Equivalent to UDP
- Checksum
  - Equivalent to UDP
  - Mandatory
TCP

• Connection oriented
  – Explicit setup and teardown required
• Full duplex
  – Data flows in both directions simultaneously
  – Point-to-point connection
• Byte stream abstraction
  – No boundaries in data
  – App writes bytes, TCP send segments, App receives bytes
TCP

• Rate control
  – Flow control to restrict sender rate to something manageable by receiver
  – Congestion control to restrict sender to something manageable by network
  – Both need to handle the presence of other traffic
TCP Outline

- TCP vs. Sliding window on a direct link
- Usage model
- Segment header format and options
- States and state diagram
- Sliding window implementation details
- Flow control issues
- Bit allocation limitations
- Adaptive retransmission algorithms
TCP vs. Direct Link

• Explicit connection setup required
  – Dialup vs. dedicated line

• RTT varies
  – Among peers (host at other end of connection)
  – Over time
  – Requires adaptive approach to retransmission (and window size)

• Packets
  – Delayed
  – Reordered
  – Late
TCP vs. Direct Link

- Peer capabilities vary
  - Minimum link speed on route
  - Buffering capacity at destination
  - Requires adaptive approach to window sizes

- Network capacity varies
  - Other traffic competes for most links
  - Requires global congestion control strategy

- Question
  - Why not implement more functionality (reliability, ordering, congestion control) in IP?
Proposal: Reliable Network Layer

- **Service**
  - High probabilistic guarantee of correct, in order data transmission at the network layer
  - Hop-by hop network layer ACKs
- **Is this sufficient?**
- **No**
  - Routers may crash, buffers may overflow
- **Is it beneficial?**
  - Maybe, depends on link’s error rate
  - Improve performance, not provide correctness
The End-to-End Argument

• Lower layer functions
  – May be redundant or of little value when compared with providing them at that low layer

• Functionality
  – Implemented at a lower layer iff it can be correctly and completely implemented there

• Real constraint
  – Implementing functionality at a lower level should have minimum performance impact on applications that do not use the functionality
End-to-End Argument

- In-order delivery
  - hop-by-hop ordering guarantee is not robust to path changes or multiple paths

- Congestion control
  - Should be stopped at source
  - But network can provide feedback

[Diagram showing network bandwidth and flow with text: green should get 9Mbps, but gets only 5Mbps with hop-by-hop drops]
TCP Usage Model

- **Connection setup**
  - 3-way handshake

- **Data transport**
  - Sender writes data
  - TCP
    - Breaks data into segments
    - Sends each segment over IP
    - Retransmits, reorders and removes duplicates as necessary
  - Receiver reads some data

- **Teardown**
  - 4 step exchange
TCP Connection Establishment

• 3-Way Handshake
  – Sequence Numbers
    • J,K
  – Message Types
    • Synchronize (SYN)
    • Acknowledge (ACK)
  – Passive Open
    • Server listens for connection from client
  – Active Open
    • Client initiates connection to server

Client

Server

- Time flows down

listen

Synchronize (SYN) J

SYN K, acknowledge (ACK) J+1

ACK K+1
TCP Data Transport

- Data broken into segments
  - Limited by maximum segment size (MSS)
  - Defaults to 352 bytes
  - Negotiable during connection setup
  - Typically set to
    - MTU of directly connected network – size of TCP and IP headers

- Three events cause a segment to be sent
  - ≥ MSS bytes of data ready to be sent
  - Explicit PUSH operation by application
  - Periodic timeout
TCP Byte Stream

Application process

Write bytes

TCP

Send buffer

TCP Segment

TCP Segment

TCP Segment

Application process

Read bytes

TCP

Recv buffer
TCP Connection Termination

- Two generals problem
  - Enemy camped in valley
  - Two generals’ hills separated by enemy
  - Communication by unreliable messengers
  - Generals need to agree whether to attack or retreat
Two generals problem

- Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
  - No, even if all messages get through

- No way to be sure last message gets through!
TCP Connection Termination

- **Message Types**
  - Finished (FIN)
  - Acknowledge (ACK)
- **Active Close**
  - Sends no more data
- **Passive close**
  - Accepts no more data

![Diagram of TCP connection termination process](image)
TCP Segment Header Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>0-7</td>
</tr>
<tr>
<td>Destination Port</td>
<td>8-15</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>16-21</td>
</tr>
<tr>
<td>ACK Sequence Number</td>
<td>22-27</td>
</tr>
<tr>
<td>Header Length</td>
<td>28-29</td>
</tr>
<tr>
<td>Flags</td>
<td>30-31</td>
</tr>
<tr>
<td>Advertised Window</td>
<td></td>
</tr>
<tr>
<td>TCP Checksum</td>
<td></td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td></td>
</tr>
<tr>
<td>Options</td>
<td></td>
</tr>
</tbody>
</table>
**TCP Segment Header Format**

<table>
<thead>
<tr>
<th>Source Port</th>
<th>Destination Port</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequence Number</strong></td>
<td></td>
</tr>
<tr>
<td><strong>ACK Sequence Number</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Header Length</strong></td>
<td><strong>Flags</strong></td>
</tr>
<tr>
<td><strong>TCP Checksum</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Options</strong></td>
<td></td>
</tr>
</tbody>
</table>

- 16-bit source and destination ports
### TCP Segment Header Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>0 - 7</td>
</tr>
<tr>
<td>Destination Port</td>
<td>8 - 15</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>16 - 23</td>
</tr>
<tr>
<td>ACK Sequence Number</td>
<td>24 - 31</td>
</tr>
<tr>
<td>Header Length</td>
<td>0</td>
</tr>
<tr>
<td>Flags</td>
<td>1</td>
</tr>
<tr>
<td>Advertised Window</td>
<td>2</td>
</tr>
<tr>
<td>TCP Checksum</td>
<td>3</td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td>4</td>
</tr>
<tr>
<td>Options</td>
<td>5 - 31</td>
</tr>
</tbody>
</table>

- 32-bit send and ACK sequence numbers
TCP Segment Header Format

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>Destination Port</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence Number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACK Sequence Number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Header Length</td>
<td>Flags</td>
<td>Advertised Window</td>
<td></td>
</tr>
<tr>
<td>TCP Checksum</td>
<td>Urgent Pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 4-bit header length in 4-byte words
  - Minimum 5 bytes
  - Offset to first data byte
### TCP Segment Header Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>0-7</td>
<td>16-bit integer for source port</td>
</tr>
<tr>
<td>Destination Port</td>
<td>8-15</td>
<td>16-bit integer for destination port</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>16-23</td>
<td>32-bit integer for sequence number</td>
</tr>
<tr>
<td>ACK Sequence Number</td>
<td>24-31</td>
<td>32-bit integer for ACK sequence number</td>
</tr>
<tr>
<td>Header Length</td>
<td>0</td>
<td>4-bit integer for header length</td>
</tr>
<tr>
<td>Flags</td>
<td>1</td>
<td>4-bit integer for flags</td>
</tr>
<tr>
<td>Advertised Window</td>
<td>2</td>
<td>32-bit integer for advertised window</td>
</tr>
<tr>
<td>TCP Checksum</td>
<td>3</td>
<td>16-bit integer for TCP checksum</td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td>4</td>
<td>32-bit integer for urgent pointer</td>
</tr>
</tbody>
</table>

- **Reserved**
  - Must be 0
### TCP Segment Header Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>0-7</td>
</tr>
<tr>
<td>Destination Port</td>
<td>8-15</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>16-23</td>
</tr>
<tr>
<td>ACK Sequence Number</td>
<td>24-31</td>
</tr>
<tr>
<td>Header Length</td>
<td>0</td>
</tr>
<tr>
<td>Flags</td>
<td>3-5</td>
</tr>
<tr>
<td>Advertised Window</td>
<td>6-15</td>
</tr>
<tr>
<td>TCP Checksum</td>
<td>16-31</td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td>32</td>
</tr>
</tbody>
</table>

- **6 1-bit flags**
  - **URG**: Contains urgent data
  - **ACK**: Valid ACK seq. number
  - **PSH**: Do not delay data delivery
  - **RST**: Reset connection
  - **SYN**: Synchronize for setup
  - **FIN**: Final segment for teardown
### TCP Segment Header Format

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>Destination Port</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence Number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACK Sequence Number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Header Length</td>
<td>0</td>
<td>Flags</td>
<td>Advertised Window</td>
</tr>
<tr>
<td>TCP Checksum</td>
<td>Urgent Pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **16-bit advertised window**
  - Space remaining in receive window
TCP Segment Header Format

- **16-bit checksum**
  - Uses IP checksum algorithm
  - Computed on header, data and pseudo header
**TCP Segment Header Format**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>0-7</td>
</tr>
<tr>
<td>Destination Port</td>
<td>8-15</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>16-23</td>
</tr>
<tr>
<td>ACK Sequence Number</td>
<td>24-31</td>
</tr>
<tr>
<td>Header Length</td>
<td>0</td>
</tr>
<tr>
<td>Flags</td>
<td>1-3</td>
</tr>
<tr>
<td>Advertised Window</td>
<td>4-15</td>
</tr>
<tr>
<td>TCP Checksum</td>
<td>16-31</td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td>0-31</td>
</tr>
<tr>
<td>Options</td>
<td></td>
</tr>
</tbody>
</table>

- **16-bit urgent data pointer**
  - If URG = 1
  - Index of last byte of urgent data in segment
TCP Options

- Negotiate maximum segment size (MSS)
  - Each host suggests a value
  - Minimum of two values is chosen
  - Prevents IP fragmentation over first and last hops

- Packet timestamp
  - Allows RTT calculation for retransmitted packets
  - Extends sequence number space for identification of stray packets

- Negotiate advertised window granularity
  - Allows larger windows
  - Good for routes with large bandwidth-delay products
## TCP State Descriptions

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSED</td>
<td>Disconnected</td>
</tr>
<tr>
<td>LISTEN</td>
<td>Waiting for incoming connection</td>
</tr>
<tr>
<td>SYN_RCVD</td>
<td>Connection request received</td>
</tr>
<tr>
<td>SYN_SENT</td>
<td>Connection request sent</td>
</tr>
<tr>
<td>ESTABLISHED</td>
<td>Connection ready for data transport</td>
</tr>
<tr>
<td>CLOSE_WAIT</td>
<td>Connection closed by peer</td>
</tr>
<tr>
<td>LAST_ACK</td>
<td>Connection closed by peer, closed locally, await ACK</td>
</tr>
<tr>
<td>FIN_WAIT_1</td>
<td>Connection closed locally</td>
</tr>
<tr>
<td>FIN_WAIT_2</td>
<td>Connection closed locally and ACK’d</td>
</tr>
<tr>
<td>CLOSING</td>
<td>Connection closed by both sides simultaneously</td>
</tr>
<tr>
<td>TIME_WAIT</td>
<td>Wait for network to discard related packets</td>
</tr>
</tbody>
</table>
TCP State Transition Diagram
TCP State Transition Diagram

- Passive open
- SYN/SYN + ACK
- LISTEN
- Event from local application/message sent
- Passive open
- SYN/SYN + ACK
- SYN_SENT
- Active open/SYN
- Close
- Send/ SYN
- SYN/Rcvd
- ACK
- SYN/ SYN + ACK
- ESTABLISHED
- SYN + ACK/ACK
- Close/ACK
- CLOSING
- FIN/ACK
- FIN/ACK
- CLOSED
- Timeout
- FIN/ACK
- FIN + ACK/ACK
- ACK
- TIME_WAIT
- LAST_ACK
- Close/FIN
- ACK
- CLOSE_WAIT
- Close/FIN
- ACK
- CLOSED
- ACK
- FIN/ACK
- FIN_WAIT_2
- ACK
- FIN_WAIT_1
- Close/FIN
- ACK
- Message from receiver/response sent
- Event from local application/message sent
- m local application
TCP State Transition Diagram

- **Questions**
  - **State transitions**
    - Describe the path taken by a server under normal conditions
    - Describe the path taken by a client under normal conditions
    - Describe the path taken assuming the client closes the connection first
  - **TIME_WAIT state**
    - What purpose does this state serve
    - Prove that at least one side of a connection enters this state
    - Explain how both sides might enter this state
TCP State Transition Diagram

Establishment under normal conditions

CLOSED

LISTEN

SYN_RCVD

SYN_SENT

SYN/SYN + ACK

SYN/SYN + ACK

ACK

ACK

FIN/SYN

Send/SYN

Close

Close

FIN/SYN + ACK

SYN + ACK/ACK

CLOSE_WAIT

LAST_ACK

TIME_WAIT

CLOSING

FIN_WAIT_2

FIN_WAIT_1

Establishment under normal conditions

Active open/SYN

Timeout

Send/SYN

ACK

Timeout

ACK
TCP State Transition Diagram

Lost ACK from receiver?

- Passive open
- SYN/SYN + ACK
- ACTIVE open SYN
- Send/SYN
- SYN/SYN + ACK
- SYN_RCVD
- Close/FIN
- Passive open
- SYN/SYN + ACK
- LISTEN
- SYN_SENT
- SYN/SYN + ACK
- ESTABLISHED
- FIN/ACK
- SYN/ACK
- FIN_WAIT_1
- FIN/ACK
- FIN_WAIT_2
- FIN/ACK
- CLOSING
- FIN/ACK
- TIME_WAIT
- FIN/ACK
- CLOSE_WAIT
- Close/FIN
- LAST_ACK
- ACK
- Timeout
- CLOSED
TCP State Transition Diagram

Timeouts?

CLOSED
- Passive open
- Close

LISTEN
- SYN/SYN + ACK
- Send/SYN

SYN_RCVD
- Close/FIN
- ACK
- SYN/SYN + ACK

FIN_WAIT_1
- ACK
- FIN + ACK/ACK

FIN_WAIT_2
- FIN/ACK

CLOSING
- ACK

TIME_WAIT
- Timeout

CLOSE_WAIT
- Close/FIN

LAST_ACK
- ACK

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

If no response after multiple tries, return to CLOSED

Send/SYN
If no response after multiple tries, return to CLOSED

ACK

CS/ECE 438
© Robin Kravets & Matt Caesar, UIUC - Spring 2009 50
TCP State Transition Diagram

One side closes first
TCP TIME_WAIT State

• Problem
  – What happens if a segment from an old connection arrives at a new connection?

• Maximum Segment Lifetime
  – Max time an old segment can live in the Internet

• TIME_WAIT State
  – Connection remains in this state from two times the maximum segment lifetime
TCP State Transition Diagram

Both sides close at the same time
TCP State Transition Diagram

FIN_ACK received (rare)
TCP Sliding Window Protocol

- Sequence numbers
  - Indices into byte stream
- Initial Sequence Number
  - Why not just use 0?
- ACK sequence number
  - Actually next byte expected as opposed to last byte received
TCP Sliding Window Protocol

• Advertised window
  – Enables dynamic receive window size

• Receive buffers
  – Data ready for delivery to application until requested
  – Out-of-order data to maximum buffer capacity

• Sender buffers
  – Unacknowledged data
  – Unsent data out to maximum buffer capacity
TCP Sliding Window Protocol – Sender Side

- \( \text{LastByteAcked} \leq \text{LastByteSent} \)
- \( \text{LastByteSent} \leq \text{LastByteWritten} \)
- Buffer bytes between \( \text{LastByteAcked} \) and \( \text{LastByteWritten} \)

![Diagram of sliding window protocol]

- Maximum buffer size
- Advertised window
- First unacknowledged byte
- Data available, but outside window
- Last byte sent
TCP Sliding Window Protocol – Receiver Side

- \( \text{LastByteRead} < \text{NextByteExpected} \)
- \( \text{NextByteExpected} \leq \text{LastByteRcvd} + 1 \)
- Buffer bytes between \( \text{NextByteRead} \) and \( \text{LastByteRcvd} \)

![Diagram showing TCP sliding window protocol with last byte read, next byte expected, advertised window, and buffered, out-of-order data.]

Next byte to be read by application

Next byte expected (ACK value)

Buffered, out-of-order data

Maximum buffer size

Advertised window
Flow Control vs. Congestion Control

- **Flow control**
  - Preventing senders from overrunning the capacity of the receivers

- **Congestion control**
  - Preventing too much data from being injected into the network, causing switches or links to become overloaded

- Which one does TCP provide?
- TCP provides both
  - Flow control based on advertised window
  - Congestion control discussed later in class
TCP Flow Control: Receiver

- Receive buffer size
  - \[ \text{Receive buffer size} = \text{MaxRcvBuffer} \]
  - \[ \text{LastByteRcvd} - \text{LastByteRead} \leq \text{MaxRcvBuf} \]

- Advertised window
  - \[ \text{Advertised window} = \text{MaxRcvBuf} - (\text{NextByteExp} - \text{NextByteRead}) \]
  - Shrinks as data arrives and
  - Grows as the application consumes data
TCP Flow Control: Sender

- Send buffer size
  - \[= \text{MaxSendBuffer}\]
  - \[\text{LastByteSent} - \text{LastByteAcked} \leq \text{AdvertWindow}\]
- Effective buffer
  - \[= \text{AdvertWindow} - (\text{LastByteSent} - \text{LastByteAck})\]
  - \[\text{EffectiveWindow} > 0\] to send data

- Relationship between sender and receiver
  - \[\text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer}\]
  - block sender if \((\text{LastByteWritten} - \text{LastByteAcked}) + y > \text{MaxSenderBuffer}\)
TCP Flow Control

• Problem: Slow receiver application
  – Advertised window goes to 0
  – Sender cannot send more data
  – Non-data packets used to update window
  – Receiver may not spontaneously generate update or update may be lost

• Solution
  – Sender periodically sends 1-byte segment, ignoring advertised window of 0
  – Eventually window opens
  – Sender learns of opening from next ACK of 1-byte segment
TCP Flow Control

- Problem: Application delivers tiny pieces of data to TCP
  - Example: telnet in character mode
  - Each piece sent as a segment, returned as ACK
  - Very inefficient

- Solution
  - Delay transmission to accumulate more data
  - Nagle’s algorithm
    - Send first piece of data
    - Accumulate data until first piece ACK’d
    - Send accumulated data and restart accumulation
    - Not ideal for some traffic (e.g., mouse motion)
TCP Flow Control

• Problem: Slow application reads data in tiny pieces
  – Receiver advertises tiny window
  – Sender fills tiny window
  – Known as silly window syndrome

• Solution
  – Advertise window opening only when MSS or ½ of buffer is available
  – Sender delays sending until window is MSS or ½ of receiver’s buffer (estimated)
TCP Bit Allocation Limitations

- Sequence numbers vs. packet lifetime
  - Assumed that IP packets live less than 60 seconds
  - Can we send $2^{32}$ bytes in 60 seconds?
  - Less than an STS-12 line

- Advertised window vs. delay-bandwidth
  - Only 16 bits for advertised window
  - Cross-country RTT = 100 ms
  - Adequate for only 5.24 Mbps!
## TCP Sequence Numbers – 32-bit

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>Speed</th>
<th>Time until wrap around</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1.5 Mbps</td>
<td>6.4 hours</td>
</tr>
<tr>
<td>Ethernet</td>
<td>10 Mbps</td>
<td>57 minutes</td>
</tr>
<tr>
<td>T3</td>
<td>45 Mbps</td>
<td>13 minutes</td>
</tr>
<tr>
<td>FDDI</td>
<td>100 Mbps</td>
<td>6 minutes</td>
</tr>
<tr>
<td>STS-3</td>
<td>155 Mbps</td>
<td>4 minutes</td>
</tr>
<tr>
<td>STS-12</td>
<td>622 Mbps</td>
<td>55 seconds</td>
</tr>
<tr>
<td>STS-24</td>
<td>1.2 Gbps</td>
<td>28 seconds</td>
</tr>
</tbody>
</table>
## TCP Advertised Window – 16-bit

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>Speed</th>
<th>Delay x Bandwidth Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1.5 Mbps</td>
<td>18 KB</td>
</tr>
<tr>
<td>Ethernet</td>
<td>10 Mbps</td>
<td>122 KB</td>
</tr>
<tr>
<td>T3</td>
<td>45 Mbps</td>
<td>549 KB</td>
</tr>
<tr>
<td>FDDI</td>
<td>100 Mbps</td>
<td>1.2 MB</td>
</tr>
<tr>
<td>STS-3</td>
<td>155 Mbps</td>
<td>1.8 MB</td>
</tr>
<tr>
<td>STS-12</td>
<td>622 Mbps</td>
<td>7.4 MB</td>
</tr>
<tr>
<td>STS-24</td>
<td>1.2 Gbps</td>
<td>14.8 MB</td>
</tr>
</tbody>
</table>
TCP Round Trip Time and Timeout

- How should TCP set its timeout value?
  - Longer than RTT
    - But RTT varies
  - Too short
    - Premature timeout
    - Unnecessary retransmissions
  - Too long
    - Slow reaction to segment loss

- Estimating RTT
  - SampleRTT
    - Measured time from segment transmission until ACK receipt
    - Will vary
    - Want smoother estimated RTT
  - Average several recent measurements
    - Not just current SampleRTT
TCP Adaptive Retransmission Algorithm - Original

- **Theory**
  - Estimate RTT
  - Multiply by 2 to allow for variations

- **Practice**
  - Use exponential moving average \( \alpha = 0.1 \text{ to } 0.2 \)
  - Estimate = \((\alpha) \times \text{measurement} + (1- \alpha) \times \text{estimate}\)
  - Influence of past sample decreases exponentially fast
TCP Adaptive Retransmission Algorithm - Original

- Problem: What does an ACK really ACK?
  - Was ACK in response to first, second, etc transmission?
TCP Adaptive Retransmission Algorithm – Karn-Partridge

- Algorithm
  - Exclude retransmitted packets from RTT estimate
  - For each retransmission
    - Double RTT estimate
    - Exponential backoff from congestion
TCP Adaptive Retransmission Algorithm – Karn-Partridge

• Problem
  – Still did not handle variations well
  – Did not solve network congestion problems as well as desired
  • At high loads round trip variance is high
Example RTT Estimation

SampleRTT

Estimated RTT

RTT (milliseconds)

SampleRTT

Estimated RTT

time (seconds)

1 50 100

350

300

250

200

150

100

1 100

1 50 100
TCP Adaptive Retransmission Algorithm – Jacobson

• Algorithm
  – Estimate variance of RTT
    • Calculate mean interpacket RTT deviation to approximate variance
    • Use second exponential moving average
    • Dev = (β) * |RTT_Est – Sample| + (1–β) * Dev
    • β = 0.25, A = 0.125 for RTT_est
  – Use variance estimate as component of RTT estimate
    • Next_RTT = RTT_Est + 4 * Dev
  – Protects against high jitter
TCP Adaptive Retransmission Algorithm – Jacobson

- Notes
  - Algorithm is only as good as the granularity of the clock
  - Accurate timeout mechanism is important for congestion control
Evolution of TCP

1975
Three-way handshake
Raymond Tomlinson
In SIGCOMM 75

1974
TCP described by
Vint Cerf and Bob Kahn
In IEEE Trans Comm

1980
1983
BSD Unix 4.2
supports TCP/IP

1984
Nagel’s algorithm
to reduce overhead
of small packets; predicts congestion collapse

1985
1986
Congestion collapse observed

1987
Karn’s algorithm
to better estimate round-trip time

1988
Van Jacobson’s algorithms
congestion avoidance and congestion control
(most implemented in 4.3BSD Tahoe)

1990
4.3BSD Reno
fast retransmit delayed ACK’s
TCP Through the 1990s

1993
TCP Vegas
(Brakmo et al)
delay-based
congestion avoidance

1994
ECN
(Floyd)
Explicit Congestion Notification

1996
SACK TCP
(Floyd et al)
Selective Acknowledgement

1996
Hoe
NewReno startup and loss recovery