Misc.

• Last Time
  – Finished Multicast–RLM, SRM, Application Level Multicast

• Today
  – Start End-to-end protocols
RLM

- Problem: how to do congestion control on a multicast tree
  - Different receivers can receive at different rates
  - Source based solutions are unscalable

- Solution?

- How is SRM problem/solution different?
Application Level Multicast Paper

- Not going to wait for Multicast to be deployed

- Multicast at the application level
  - Less efficient (recall Mbone)
  - But much more easily deployable

- Major effort is in constructing a geographically consistent overlay
Discussion

• Naming: How to obtain globally unique multicast addresses
  - How do we do this in a scalable and distributed fashion?
  - Use DNS?

• Security Nightmare?

• Reliability? congestion control?

• Application Level multicast the solution?

• Multicast Still an open problem – partial solutions deployed
End-to-End (Transport) Protocols

• IP and the network layer provide host-to-host connectivity across a scalable heterogeneous network

• Unfortunately, IP is a best-effort network; it can
  – Drop Messages
  – Reorder messages
  – Duplicate messages
  – Delay messages a long time
  – Limit size of messages

• How do these features compare with the requirements of applications?
  – End-to-End Protocols provide better service models to applications
  – Recall the End-to-End argument, these guys should do all the work!

• How to get from host-to-host to process-to-process communication?
End-to-end Services

• Ideally: transport protocol worries about the end-to-end service provided to the application; *it does not care about the communication path*

• What are common end-to-end services of interest?
  – Allow multiple processes on a host (is this possible with IP?)
  – Guarantee message delivery
  – Guarantee ordered delivery
  – No duplicates
  – Arbitrary size messages
  – Flow control (don't overflow receiver)
  – Congestion control (don't overflow network) – is this a service?
  – other? (QoS, Encryption, Synchronization...)

• Are these needed by all applications?
Discussion

- Why is end-to-end operation different from link-level communication
  - At the link level 2 ends on the link communicate with each other
  - End-to-end, 2 ends of the connection communicate with each other
Transport Protocols

• User Datagram Protocol
  – Basic transport (process to process IP)
  – Many other protocols built on top of it

• Transmission Control Protocol (TCP)
  – Reliable bytestream; many bells and whistles

• Others, including:
  – Realtime Transmission Protocol (RTP/RTCP)
  – Remote Procedure Call (RPC)
  – Stream Control Transmission Protocol (SCTP)

• Multicast Transport Protocols
  – MFTP, PGM, (RLM/SRM?) etc..

• Point – TCP is not the only transport protocol
UDP – User Datagram Protocol

- The simplest end-to-end protocol is to extend IP to recognize multiple processes per host
- UDP provides a simple demultiplexing key to differentiate between processes – no other functionality is supported
  - e.g., when a message arrives, if queues are full it is dropped
  - Why is this interesting?

- What should be used as a demultiplex key
  - How about process id?
- **Port Numbers** are used as a demultiplex key
  - A Port is a logical “mailbox” which is associated with a process

- How does a process know the “key” for a process it wants to communicate with?
  - Well known port numbers for most servers (e.g., http server at port 80; defined in RFC 1700)
  - Otherwise, by out-of-band agreement

- Try to tie this in with socket programming
**UDP**

<table>
<thead>
<tr>
<th>0</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>SrcPort</td>
<td>DstPort</td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td>Length</td>
<td></td>
</tr>
</tbody>
</table>

- **Data**

- **UDP checksum is optional; when used, it checksums**
  - the whole message body (including UDP header)
  - Psuedoheader from IP

- **Recall that IP checksum was on the IP header only**
- **Idea: protection against misrouted datagrams**
Transmission Control Protocol (TCP)

- A reliable connection-oriented service model
  - Reliable: everything gets there exactly one time
  - connection-oriented: in-order delivery of a stream of bytes
  - Full duplex

- Most widely used and most carefully “tuned” transport protocol on the internet

- Like UDP supports multiple processes per host (also using port numbers)

- TCP implements both flow-control and congestion control (will discuss in detail later)
TCP Overview of Operation

- Connection establishment is needed
- Sending process writes some bytes (any number)
- TCP breaks into segments and sends via IP
- Receiving process reads some bytes (any number)
- How big is the segment?
- When does TCP send the segments?
- How to implement Reliability and in-order delivery?
TCP – Overview (cont’d)

- Common choice for Maximum Segment Size (MSS): maximum size that will not cause IP to fragment locally
  - What is this equal to?
- When to send a “segment”?
  1. When there is enough data to send an MSS
  2. If the application demands an immediate send
  3. Set a timer when you send a segment; send again when it fires
- Why three different ways?
- Packet boundaries are not visible to a process
- Reliability? Need some form of ARQ (isn’t it supported at link layer?)
- In-order delivery? Don’t allow a receive until all preceding data has arrived
### TCP

<table>
<thead>
<tr>
<th>SrcPort</th>
<th>DstPort</th>
</tr>
</thead>
<tbody>
<tr>
<td>SequenceNum</td>
<td></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>0</td>
</tr>
<tr>
<td>Checksum</td>
<td>UrgPtr</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>

- Source port and Destination port identify processes
  - Along with source/destination IP addresses form a unique connection identifier – true/false?
- TCP runs a sliding window algorithm
  - Acknowledgements used to ack received segments
  - Sequence number of the first byte in the segment
  - Advertised window is the size of the window at the receiver (flow control)
- Checksum is identical to UDP
Connection Establishment

- The sequence number is the number of the byte received last + 1
- Initially randomly picked
- Note Duplex operation
- Normal operation occurs within the established state
- Why timeout state?
- Track connection establishment and teardown
TCP “Established” Operation

- Strategy – Sliding window ARQ
  - Use ACKs and Sequence numbers
    - ACK sets a flag bit to say that the ACK field is valid
  - Flow control using advertised window
  - Congestion control
TCP's sliding window is a hybrid of Selective repeat and Go-Back-N

- Like selective repeat, buffer segments that arrive out of order are accepted
- Like Go-Back-N, uses cumulative ACKs

Sender window size obtained by explicit feedback
- What size is it?
- Later, we will update to incorporate congestion
Flow Control

- A LOT of details in TCP operation/optimization; we will just sample a few
- Flow control different from Congestion control
  - Sometimes confused because their support is integrated
- Advertised window is: \(\text{MaxRcvBuffer} - (\text{LastByteRcvd} - \text{LastByteRead})\)
- Sender sends if the packets it has already sent out are less than the advertised window)
  - Effective Send Window = Advertised Window - (Last Byte Sent - Last Byte ACK’d)
  - Why this conservative?
Some Problems

• Zero Window Advertisement
  – How does the sender learn that the receivers window opened up?
    ✴ Possibility: have receiver send a segment when window opens up
    ✴ Might not have data available; would like smart sender dumb receiver
  – Actual Solution
    ✴ Periodic ping with a 1 byte segment; ACK reports new window size
    ✴ *persist timer* fires the send of the probe; exponentially backed off

• Very late packets (session reincarnation)
  – Highly delayed packet arrives from a previous connection
  – receiver responds with an ACK with RST (reset) set
  – Same is true if the packet arrives to the port with a wrong source address (not matching peer ip or port)
Performance Problems/Tuning

- **Performance**
  - Maximize throughput (send as much as possible in each segment?)
  - Minimize latency (send data as soon as it arrives?)

- **Silly Window Syndrome**
  - Receiver window full
  - Receiver reads one byte
  - Sender sends 1 byte
  - Receiver reads 1 byte
  - sender sends 1 byte, etc..
  - “High overhead tinygrams” – wasting TCP/IP headers (40 bytes) for 1 byte data

- **What can we do to solve SWS?**
Addressing Silly Window Syndrome

- Would like to delay sending until we have enough to send; but cannot delay too much

- Receiver side solutions
  - Avoid advertising small increments in advertised window
  - Delay acknowledgements:
    * ACKs can be delayed but by no more than 0.5 seconds
    * In a stream of full size segments, every other segment must be ACK’d

- Sender side
  - Nagle’s algorithm: avoid sending small segments of data until everything sent before them has been ACK’d, or we have a full segment
  - Use Nagle’s algorithm but provide a way to disable it (TCP_NODELAY)
Problem – Gigabit Networks and TCP Wraparound

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>32-bit Wraparound Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 (1.5Mbps)</td>
<td>6.4hrs</td>
</tr>
<tr>
<td>Ethernet (10Mbps)</td>
<td>57min</td>
</tr>
<tr>
<td>T3 (45Mbps)</td>
<td>13 min</td>
</tr>
<tr>
<td>FDDI (100Mbps)</td>
<td>6 min</td>
</tr>
<tr>
<td>STS-3 (155Mbps)</td>
<td>4 min</td>
</tr>
<tr>
<td>STS-12 (622Mbps)</td>
<td>55 sec</td>
</tr>
<tr>
<td>STS-24 (2.4Gbps)</td>
<td>28 sec</td>
</tr>
</tbody>
</table>

- TCP assumes a packet cannot live more than Maximum Segment Life (MSL = 120s)
- If a TCP connection is using an STS-12+ connection, wraparound is faster than MSL
  - Cannot distinguish a delayed segment from a new segment
- Currently not a problem
  - TCP connections do not run at those speeds
  - IETF big fat pipe extension
Problem – Advertised Window Field

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

- Table shows delay bandwidth for RTT=100ms connection
- Advertised window is 16-bits; maximum buffer size is 64Kbyte
- To keep the pipe full, the advertised window field should be bigger than the delay bandwidth product (why?)
- Also addressed by “big fat pipe” TCP extension
Big Fat Pipe Extension

- One more problem with TCP is the granularity of the timer (originally 0.5 seconds)
- Extension addresses wraparound as well as timer granularity
- Implemented using the optional portion of the TCP header
- Stores timestamps in outgoing segments, timestamp included in reply
  - System time can be used to provide fine grained estimates
  - Timestamp effectively extends the sequence number solving the wraparound problem
- Extension also allows the peers to agree on a scaling factor for the advertised window
Revisiting Sliding Window

- End-to-end delay is highly variable
  - Consider, a cross-continent TCP connection vs. a TCP connection on the same LAN
  - How should the timeout value be picked?

- Idea: Make it a function of the measured RTT
  - Difference between send time and ACK receipt (timer granularity hurts)

- Original algorithm
  - Keep an exponential average of the sampled RTT values
  - Pick timeout to be twice the estimated average
Adaptive Retransmission

- Connection can be over a wide array of “network paths”
  - how to configure TCP to work well in such diverse conditions

- An example problem – setting the Retransmission Timeout (RTO)
  - Idea: Make it a function of the measured RTT
  - Original algorithm
    * Measure RTT for every segment/ACK (with delayed ACKs every other segment)
    * Keep weighted average of RTT as:
      · Estimated RTT = $\alpha$ EstimatedRTT + (1-$\alpha$) SampleRTT
      · $\alpha$ between 0.8 and 0.9
      · RTO = 2 x EstimatedRTT
  - Karn/Partridge – ignore retransmitted segments + introduced a Binary Exponential Backoff on a lost packet
Jacobson/Karels Algorithm

• Idea: make timeout reflect variance – not only average RTT

• Algorithm (note, following original paper, not book format):
  – Estimate RTT and RTT “variance” as follows
    \[
    \text{EstimatedRTT} = (1 - \alpha)\text{EstimatedRTT} + \alpha\text{SampleRTT}
    \]
    \[
    \text{Diff} = |\text{SampleRTT} - \text{EstimatedRTT}|
    \]
    \[
    \text{Deviation} = (1 - \beta)\text{Deviation} + \beta(\text{Diff})
    \]
  
• Set timeout to be
  \[
  \text{EstimatedRTT} + \phi\text{Deviation}
  \]

• Typical values for \( \alpha \), \( \beta \) and \( \phi \) are 1/8 and 1/4 and 4 respectively

• What is the intuition, is this better than Karn/Partridge?
Problem: Waiting for timer to recover is expensive

- Would like to avoid timeouts due to a lost or delayed packet

- Send NACKs?
  - Dangerous and not necessarily efficient

- Let sender discover missing packet – how?
  - Implicit feedback: if it receives a number of duplicate acknowledgements
  - Explicit feedback: receiver sends information about what packets it received and what are missing

* Didnt we say that NACK was bad?
• Problem: how to prevent a dynamic set of users who are not aware of each other from overflowing the shared network resources
• Related problem: if the demand exceeds the capacity, how do you share the network fairly?
• Think of collisions in a shared medium: a simple form of congestion
• Two dual approaches
  – Ignore it until there is a problem then take action to control it (conceptual equivalent of contention-based MAC)
  – Pre-allocate resources to avoid problems (conceptual equivalent of what?)
• Useful to think of traffic management in transportation
TCP Congestion Control

- Problem (noticed by van Jacobson in 1988)
  - Hosts sending as fast as they are allowed
  - Congestion happens, packets are dropped
  - Hosts timeout and retransmit packets, increasing congestion

- TCP Congestion Control:
  - Assumes a best-effort network (router queueing is an independent mechanism)
  - Host-based solution: the TCP ends try to figure out what is happening based on the behavior of the network
  - ACKs pace transmission (self-clocking)
    * As ACKs are received, sliding window moves and more packets are sent
Challenges

- How to determine the end to end capacity at the beginning
- How to adapt to dynamic changes
Jacobson’s Solutions

• Additive Increase/Multiplicative decrease of congestion window size
  – Slowly increase sending rate
  – If you run into congestion, backoff aggressively
  – Oscillates around actual capacity of the network

• Implementation: Add a new variable per TCP connection: CongestionWindow

  – Limits how much data the source has in transit (how is this different from advertised window)?

  \[
  \text{MaxWindow} = \min(\text{CongestionWindow, AdvertisedWindow}) \\
  \text{EffectiveWindow} = \text{MaxWindow} - (\text{LastByteSent} - \text{LastByteAcked})
  \]

• How to control window size?
• Every time a “CongestionWindow” worth of data has been successfully sent, increment the window by one packet (Additive Increase)
• Everytime a timeout occurs, divide the congestion window by 2 (multiplicative decrease)
• In practice, increment a little for each ACK
  
  \[
  \text{Increment} = \text{MSS} \cdot \frac{\text{MSS}}{\text{CongestionWindow}} \\
  \text{If} (\text{ACKRecvd}) \\
  \text{CongestionWindow}+ = \text{Increment}
  \]
• Congestion Control mechanism in action: oscillating around the optimal sending rate – can we do better?

• Why not additive increase/additive decrease? or multiplicative increase/multiplicative decrease?

• Problem: Additive increase is too slow to get to capacity when the connection starts
  – How about fast start – send at the maximum possible local rate?