Networks, Part 2

March 7, 2001

End to End Layer

- Build upon unreliable Network Layer
  - As needed, compensate for latency, ordering, data integrity, routing accuracy, security, ...
- Create Transport Protocols
  - Send messages or streams across a network
  - Sources/Destinations are applications on hosts attached to the network
    - Apps listen for messages on Ports
    - Apps specify a message handler that listens to a port and delivers messages

Generic Message Protocol

- General Send Message interface contains:
  - send (destination address, destination port, reply port, outgoing message)
- General Receive Message Interface (e.g., a call back registered by the application):
  - accept (source address, source reply port, incoming message)

Generic Streaming Protocol

- Connection interface:
  - open (dest_address, dest_port, reply_port)
    - this returns some kind of stream id
  - close (stream_id)
- Write interface:
  - write (stream_id, data)
  - Notice that writes don't require dest. address

Streaming Protocol, example

- Read interface:
  - getting a stream_id can take many forms
    - e.g., request_stream (service_name, stream_name)
  - get_more_stream_data (stream_id)
  - end_of_stream? (stream_id)

Real examples - */IP

- IP is the Network Layer Internet Protocol
- UDP/IP - User Datagram Protocol over IP
- TCP/IP - Transmission Control Protocol over IP
- RTP/IP - Real-Time Protocol over IP

UDP/IP

- Adds the notion of ports
  - Can be used to direct traffic to particular applications
- Adds a level of data integrity (a checksum)
  - This, on top of what the link layer already does
  - Not completely redundant
- No additional notion of reliability

TCP/IP

- Messages arrive in same order as they are sent
- No missing packets
- No duplicate packets
- Some error checking
- Some flow control (to manage congestion)

➡ What many Internet services are based upon

RTP/IP

- Built on top of UDP
- Packets are time-stamped
- No other integrity guarantees

➡ Low overhead, good for streaming

At Least Once Delivery

- At least one copy of a message must be delivered to the receiver
  - Message consists of one or more datagrams
- Datagram Header contains a Datagram ID
- Sender gives Datagram to network layer, waits for ACK, retries after timeout
- What timeout period should be used?
  - Fixed timeout is simple but not optimal
  - Exponential back off is better

At Least Once Delivery, cont.

- Receiver can also use NAKs
  - Assumes datagrams have some kind of sequence number
  - Receiver sends a NAK requesting a resend of missing datagrams (needed to complete a message)
- Still need a timer (at receiver), but now, delay is per message and not per datagram

Two Generals Problem

- No 100% guarantee that a message was delivered
  - Jim N. Gray portrayed problem using a war-like scenario
- How can two generals coordinate an attack if their messaging framework is unreliable?
- There are ways to make the probability of failure vanishingly small

End to End Performance

- Control congestion via flow control
- Lock-step is simple solution, but too expensive
  - Too many round trips; one for each datagram in a message; Time: Avg. round trip time * number of datagrams
- Better to stage/pipeline sending the datagrams and receiving the ACKs
  - Great if network is fairly reliable
  - What happens if receiver cannot accept as quickly as sender can send?

Flow Control

- Sender asks how many datagrams can be sent at a time (window size)
- Sender then asks for permission to send
- Receiver sends ACKs when window is received
- Sender sends another window when given permission to send again
- Time: Time reqd. to send 1 datagram * (number of datagrams -1) + 1 round trip

➡ What window size is optimal?

Flow Control, cont.

- window size = datagram round trip time * bottleneck data rate
  - smaller window size is less efficient
  - larger window size cannot be handled by bottleneck
- In practice, use adaptive flow control
  - Change window size via simple timings
  - Losing packets requires resending a window (expensive)