CS162
Operating Systems and Systems Programming
Lecture 23

Network Communication Abstractions / Remote Procedure Call

November 21, 2007

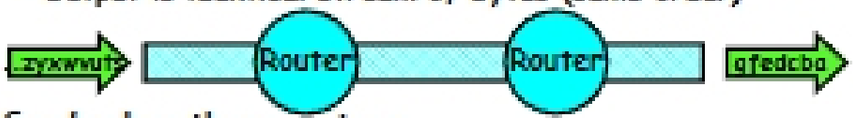
Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs162

Review: Reliable Networking

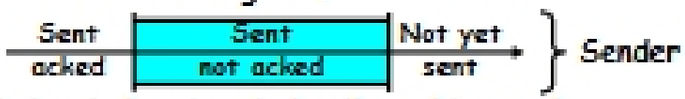
- Layering: building complex services from simpler ones
- Datagram: an independent, self-contained network message whose arrival, arrival time, and content are not guaranteed
- Performance metrics
  - Overhead: CPU time to put packet on wire
  - Throughput: Maximum number of bytes per second
  - Latency: time until first bit of packet arrives at receiver
- Arbitrary Sized messages:
  - Fragment into multiple packets; reassemble at destination
- Ordered messages:
  - Use sequence numbers and reorder at destination
- Reliable messages:
  - Use Acknowledgements
  - Want a window larger than 1 in order to increase throughput

Review: TCP Windows and Sequence Numbers

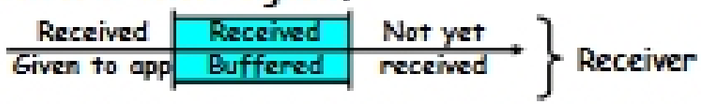
- TCP provides a stream abstraction:
  - Reliable byte stream between two processes on different machines over Internet (read, write, flush)
  - Input is an unbounded stream of bytes
  - Output is identical stream of bytes (same order)



- Sender has three regions:



  - Window (colored region) adjusted by sender
- Receiver has three regions:



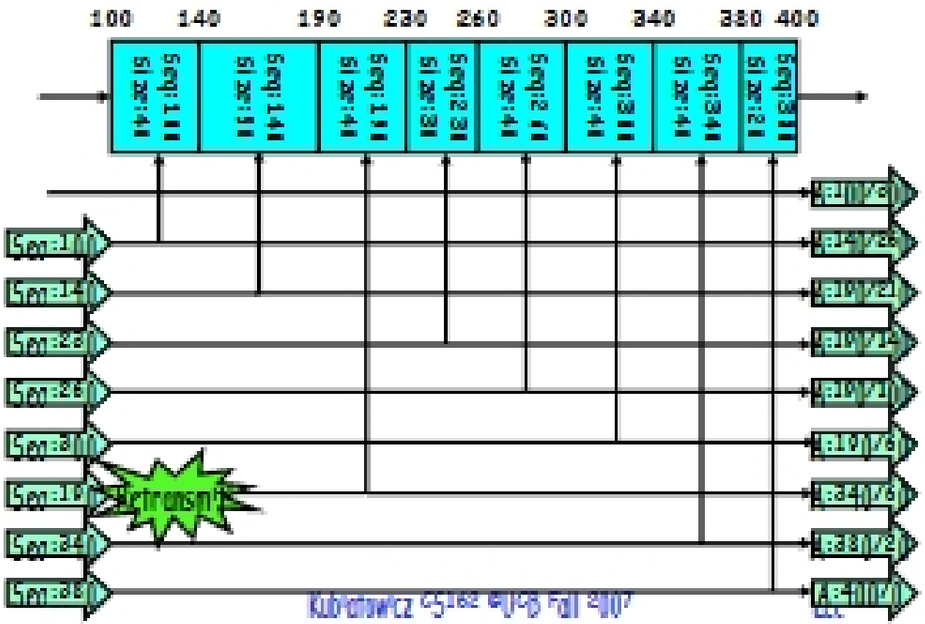
  - Maximum size of window advertised to sender at setup

Goals for Today

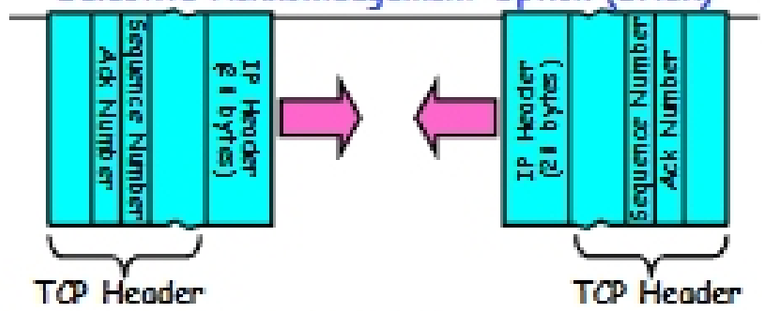
- Finish discussion of TCP
- Messages
  - Send/receive
  - One vs. two-way communication
- Distributed Decision Making
  - Two-phase commit/Byzantine Commit
- Remote Procedure Call

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Window-Based Acknowledgements (TCP)



Selective Acknowledgement Option (SACK)



- Vanilla TCP Acknowledgement
  - Every message encodes Sequence number and Ack
  - Can include data for forward stream and/or ack for reverse stream
- Selective Acknowledgement
  - Acknowledgement information includes not just one number, but rather ranges of received packets
  - Must be specially negotiated at beginning of TCP setup
    - Not widely in use (although in Windows since Windows 98)

Congestion Avoidance

- Congestion
  - How long should timeout be for re-sending messages?
    - Too long→wastes time if message lost
    - Too short→retransmit even though ack will arrive shortly
  - Stability problem: more congestion ⇒ ack is delayed ⇒ unnecessary timeout ⇒ more traffic ⇒ more congestion
    - Closely related to window size at sender: too big means putting too much data into network
- How does the sender's window size get chosen?
  - Must be less than receiver's advertised buffer size
  - Try to match the rate of sending packets with the rate that the slowest link can accommodate
  - Sender uses an adaptive algorithm to decide size of N
    - Goal: fill network between sender and receiver
    - Basic technique: slowly increase size of window until acknowledgements start being delayed/lost
- TCP solution: "slow start" (start sending slowly)
  - If no timeout, slowly increase window size (throughput) by 1 for each ack received
  - Timeout ⇒ congestion, so cut window size in half
  - "*Additive Increase, Multiplicative Decrease*"

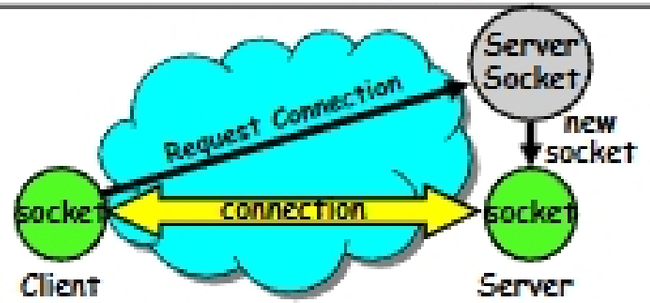
Sequence-Number Initialization

- How do you choose an initial sequence number?
  - When machine boots, ok to start with sequence #0?
    - No: could send two messages with same sequence #!
    - Receiver might end up discarding valid packets, or duplicate ack from original transmission might hide lost packet
  - Also, if it is possible to predict sequence numbers, might be possible for attacker to hijack TCP connection
- Some ways of choosing an initial sequence number:
  - Time to live: each packet has a deadline.
    - If not delivered in X seconds, then is dropped
    - Thus, can re-use sequence numbers if wait for all packets in flight to be delivered or to expire
  - Epoch #: uniquely identifies *which* set of sequence numbers are currently being used
    - Epoch # stored on disk, Put in every message
    - Epoch # incremented on crash and/or when run out of sequence #
  - Pseudo-random increment to previous sequence number
    - Used by several protocol implementations

Use of TCP: Sockets

- Socket: an abstraction of a network I/O queue
  - Embodies one side of a communication channel
    - » Same interface regardless of location of other end
    - » Could be local machine (called "UNIX socket") or remote machine (called "network socket")
  - First introduced in 4.2 BSD UNIX: big innovation at time
    - » Now most operating systems provide some notion of socket
- Using Sockets for Client-Server (C/C++ interface):
  - On server: set up "server-socket"
    - » Create socket, Bind to protocol (TCP), local address, port
    - » Call listen(): tells server socket to accept incoming requests
    - » Perform multiple accept() calls on socket to accept incoming connection request
    - » Each successful accept() returns a new socket for a new connection; can pass this off to handler thread
  - On client:
    - » Create socket, Bind to protocol (TCP), remote address, port
    - » Perform connect() on socket to make connection
    - » If connect() successful, have socket connected to server

Socket Setup (Con't)



- Things to remember:
  - Connection requires 5 values:
    [Src Addr, Src Port, Dst Addr, Dst Port, Protocol]
  - Often, Src Port "randomly" assigned
    - » Done by OS during client socket setup
  - Dst Port often "well known"
    - » 80 (web), 443 (secure web), 25 (sendmail), etc
    - » Well-known ports from 0—1023

Socket Example (Java)

```
server:
      //Makes socket, binds addr/port, calls listen()
      ServerSocket sock = new ServerSocket(6013);
      while(true) {
         Socket client = sock.accept();
         PrintWriter pout = new
           PrintWriter(client.getOutputStream(),true);

         pout.println("Here is data sent to client!");
           …
         client.close();
      }

client:
      // Makes socket, binds addr/port, calls connect()
      Socket sock = new Socket("169.229.60.38",6018);
      BufferedReader bin =
         new BufferedReader(
           new InputStreamReader(sock.getInputStream));
      String line;
      while ((line = bin.readLine())!=null)
         System.out.println(line);
      sock.close();
```


Distributed Applications

- How do you actually program a distributed application?
  - Need to synchronize multiple threads, running on different machines
    - » No shared memory, so cannot use test&set

  - One Abstraction: send/receive messages
    - » Already atomic: no receiver gets portion of a message and two receivers cannot get same message
- Interface:
  - Mailbox (mbox): temporary holding area for messages
    - » Includes both destination location and queue
  - Send(message,mbox)
    - » Send message to remote mailbox identified by mbox
  - Receive(buffer,mbox)
    - » Wait until mbox has message, copy into buffer, and return
    - » If threads sleeping on this mbox, wake up one of them