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: Using Acknowledgements

- How to ensure transmission of packets?
  - Detect garbling at receiver via checksum, discard if bad
  - Receiver acknowledges (by sending “ack”) when packet received properly at destination
  - Timeout at sender: if no ack, retransmit
- Some questions:
  - If the sender doesn’t get an ack, does that mean the receiver didn’t get the original message?
    - No
  - What if it ack gets dropped? Or if message gets delayed?
    - Sender doesn’t get ack, retransmits. Receiver gets message twice, acks each.

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
  - 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

Window-Based Acknowledgements (TCP)

- Sequence Number
- Acknowledgment Number
- Size of packets

Selective Acknowledgement

- 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

Congestion Avoidance

- 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
- Specifically TCP solution: “slow start”
  - Start sending slowly
  - If no timeout, slowly increase window size (throughput)
  - Timeout ⇒ congestion, so cut window size in half
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 implementations at this time

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 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();

Administrivia

- Cal Bears Rock!
  - 27 to 3 over Stanford
  - Quite a game: down at Stanford but more Cal fans than
    Stanford Fans
  - Also: Stanford fans don’t seem to understand “the wave”
- My office hours
  - No office hours Thursday (Thanksgiving)
- Project 4 design document
  - Due Monday November 28th
- MIDTERM II: Wednesday November 30th or Monday
  December 5th?
  » 5:30-8:30pm, 10 Evans
  » All material from last midterm and up to previous class
  » Includes virtual memory
- Final Exam
  - December 17th, 12:30 - 3:30, 220 Hearst Gym
- Final Topics: Any suggestions?
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

Using Messages: Send/Receive behavior

- When should \texttt{send(message, mbox)} return?
  - When receiver gets message? (i.e. ack received)
  - When message is safely buffered on destination?
  - Right away, if message is buffered on source node?
- Actually two questions here:
  - When can the sender be sure that receive actually received the message?
  - When can sender reuse the memory containing message?
- Mailbox provides 1-way communication from T1\rightarrow T2
  - T1\rightarrow buffer\rightarrow T2
  - Very similar to producer/consumer
    » Send = V, Receive = P
    » However, can’t tell if sender/receiver is local or not!

Messaging for Producer-Consumer Style

- Using send/receive for producer-consumer style:
  - Producer:
    ```c
    int msg1[1000];
    while(1) {
      prepare message; send(msg1, mbox);  
    }
    ```
  - Consumer:
    ```c
    int buffer[1000];
    while(1) {  
      receive(buffer, mbox); process message;
    }
    ```
- No need for producer/consumer to keep track of space in mailbox: handled by send/receive
  - One of the roles of the window in TCP: window is size of buffer on far end
  - Restricts sender to forward only what will fit in buffer

Messaging for Request/Response communication

- What about two-way communication?
  - Request/Response
    » read a file stored on a remote machine
    » Request a web page from a remote web server
  - Also called: client-server
    » Client ≡ requester, Server ≡ responder
    » Server provides “service” (file storage) to the client
- Example: File service
  - Client: (requesting the file)
    ```c
    char response[1000];
    send("read rutabaga", server_mbox);
    receive(response, client_mbox);
    ```
  - Consumer: (responding with the file)
    ```c
    char command[1000], answer[1000];
    receive(command, server_mbox);
    decode command; read file into answer; send(answer, client_mbox);
    ```
Generals Paradox

- Constraints of problem:
  - Two generals, on separate mountains
  - Can only communicate via messengers
  - Messengers can be captured
- Problem: need to coordinate attack
  - If they attack at different times, they all die
  - If they attack at same time, they win
- Named after Custer, who died at Little Big Horn because he arrived a couple of days too early

Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
- Remarkably, "no", even if all messages get through
  - No way to be sure last message gets through!

Two-Phase Commit

- Since we can't solve the Generals Paradox (i.e. simultaneous action), let's solve a related problem
- Two-Phase Commit protocol does this
  - Use a persistent, stable log on each machine to keep track of whether commit has happened
    - If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash
  - Prepare Phase:
    - The global coordinator requests that all participants will promise to commit or rollback the transaction
    - Participants record promise in log, then acknowledge
    - If anyone votes to abort, coordinator writes "Abort" in its log and tells everyone to abort; each records "Abort" in log
  - Commit Phase:
    - After all participants respond that they are prepared, then the coordinator writes "Commit" to its log
    - Then asks all nodes to commit; they respond with ack
    - After receive acks, coordinator writes "Got Commit" to log
  - Log can be used to complete this process such that all machines either commit or don't commit

Two Phase Commit Example

- Simple Example: A ≡ ATM machine, B ≡ The Bank
- Phase 1: Prepare Phase
  - A writes "Begin transaction" to log
  - B→A: OK to transfer funds to me?
  - Not enough funds: B→A: transaction aborted; A writes "Abort" to log
  - Enough funds:
    - B: Write new account balance & promise to commit to log
    - B→A: OK, I can commit
  - Phase 2: A can decide for both whether they will commit
    - A: write new account balance to log
    - Write "Commit" to log
    - Send message to B that commit occurred; wait for ack
    - Write "Got Commit" to log

What if B crashes at beginning?
- Wakes up, does nothing; A will timeout, abort and retry

What if A crashes at beginning of phase 2?
- Wakes up, sees that there is a transaction in progress; sends "Abort" to B

What if B crashes at beginning of phase 2?
- B comes back up, looks at log; when A sends it "Commit" message, it will say, "oh, ok, commit"

Distributed Decision Making Discussion

- Why is distributed decision making desirable?
  - Fault Tolerance!
  - A group of machines can come to a decision even if one or more of them fail during the process
    - Simple failure mode called "failstop" (different modes later)
  - After decision made, result recorded in multiple places
- Undesirable feature of Two-Phase Commit: Blocking
  - One machine can be stalled until another site recovers:
    - Site B writes "prepared to commit" record to its log, sends a "yes" vote to the coordinator (site A) and crashes
    - Site A crashes
    - Site B wakes up, check its log, and realizes that it has voted "yes" on the update. It sends a message to site A asking what happened. At this point, B cannot decide to abort, because update may have committed
    - B is blocked until A comes back
  - A blocked site holds resources (locks on updated items, pages pinned in memory, etc) until learns fate of update
- Alternative: There are alternatives such as "Three Phase Commit" which don't have this blocking problem
- What happens if one or more of the nodes is malicious?
  - Malicious: attempting to compromise the decision making
Byzantine General's Problem (n players):
- One General
- n-1 Lieutenants
- Some number of these (f) can be insane or malicious

The commanding general must send an order to his n-1 lieutenants such that:
- IC1: All loyal lieutenants obey the same order
- IC2: If the commanding general is loyal, then all loyal lieutenants obey the order he sends

Impossibility Results:
- Cannot solve Byzantine General's Problem with n=3 because one malicious player can mess up things
- With f faults, need n > 3f to solve problem

Various algorithms exist to solve problem
- Original algorithm has #messages exponential in n
- Newer algorithms have message complexity $O(n^2)$
  » One from MIT, for instance (Castro and Liskov, 1999)

Use of BFT (Byzantine Fault Tolerance) algorithm
- Allow multiple machines to make a coordinated decision even if some subset of them (< n/3) are malicious

Remote Procedure Call
- Raw messaging is a bit too low-level for programming
  - Must wrap up information into message at source
  - Must decide what to do with message at destination
  - May need to sit and wait for multiple messages to arrive
- Better option: Remote Procedure Call (RPC)
  - Calls a procedure on a remote machine
    - Client calls: `remoteFileSystem.ReadFile("rutabaga");`
    - Translated automatically into call on server: `fileSys.Read("rutabaga");`
- Implementation:
  - Request-response message passing (under covers!)
  - "Stub" provides glue on client/server
    » Client stub is responsible for "marshalling" arguments and "unmarshalling" the return values
    » Server-side stub is responsible for "unmarshalling" arguments and "marshalling" the return values.
- Marshalling involves (depending on system)
  - Converting values to a canonical form, serializing objects, copying arguments passed by reference, etc.
**RPC Details**

- Equivalence with regular procedure call
  - Parameters ⇔ Request Message
  - Result ⇔ Reply message
  - Name of Procedure: Passed in request message
  - Return Address: mbox2 (client return mail box)
- Stub generator: Compiler that generates stubs
  - Input: interface definitions in an “interface definition language (IDL)"
    » Contains, among other things, types of arguments/return
  - Output: stub code in the appropriate source language
    » Code for client to pack message, send it off, wait for result, unpack result and return to caller
    » Code for server to unpack message, call procedure, pack results, send them off
- Cross-platform Issues:
  - What if client/server machines are different architectures or in different languages?
    » Convert everything to/from some canonical form
    » Tag every item with an indication of how it is encoded (avoids unnecessary conversions).

**RPC Details (continued)**

- How does client know which mbox to send to?
  - Need to translate name of remote service into network endpoint (Remote machine, port, possibly other info)
    - Binding: the process of converting a user-visible name into a network endpoint
      » This is another word for “naming” at network level
      » Static: fixed at compile time
      » Dynamic: performed at runtime
- Dynamic Binding
  - Most RPC systems use dynamic binding via name service
    » Name service provides dynamic translation of service—mbox
  - Why dynamic binding?
    » Access control: check who is permitted to access service
    » Fail-over: If server fails, use a different one
- What if there are multiple servers?
  - Could give flexibility at binding time
    » Choose unloaded server for each new client
  - Could provide same mbox (router level redirect)
    » Choose unloaded server for each new request
    » Only works if no state carried from one call to next
- What if multiple clients?
  - Pass pointer to client-specific return mbox in request

**Problems with RPC**

- Non-Atomic failures
  - Different failure modes in distributed system than on a single machine
  - Consider many different types of failures
    » User-level bug causes address space to crash
    » Machine failure, kernel bug causes all processes on same machine to fail
    » Some machine is compromised by malicious party
  - Before RPC: whole system would crash/die
  - After RPC: One machine crashes/compromised while others keep working
    » Can easily result in inconsistent view of the world
      » Did my cached data get written back or not?
      » Did server do what I requested or not?
    » Answer? Distributed transactions/Byzantine Commit
- Performance
  - Cost of Procedure call << same-machine RPC << network RPC
  - Means programmers must be aware that RPC is not free
    » Caching can help, but may make failure handling complex

**Cross-Domain Communication/Location Transparency**

- How do address spaces communicate with one another?
  - Shared Memory with Semaphores, monitors, etc...
  - File System
  - Pipes (1-way communication)
  - “Remote” procedure call (2-way communication)
- RPC’s can be used to communicate between address spaces on different machines on the same machine
  - Services can be run wherever it’s most appropriate
  - Access to local and remote services looks the same
- Examples of modern RPC systems:
  - CORBA (Common Object Request Broker Architecture)
  - DCOM (Distributed COM)
  - RMI (Java Remote Method Invocation)
Microkernel operating systems

- Example: split kernel into application-level servers.
  - File system looks remote, even though on same machine

- Why split the OS into separate domains?
  - Fault isolation: bugs are more isolated (build a firewall)
  - Enforces modularity: allows incremental upgrades of pieces of software (client or server)
  - Location transparent: service can be local or remote
    - For example in the X windowing system: Each X client can be on a separate machine from X server; Neither has to run on the machine with the frame buffer.

Monolithic Structure

App  App  App

file system  Windowing
VM  Networking
Threads

Microkernel Structure

App  File  sys  windows
RPC  address  spaces  threads

Conclusion

- TCP: Reliable byte stream between two processes on different machines over Internet (read, write, flush)
  - Uses window-based acknowledgement protocol
  - Congestion-avoidance dynamically adapts sender window to account for congestion in network
- Two-phase commit: distributed decision making
  - First, make sure everyone guarantees that they will commit if asked (prepare)
  - Next, ask everyone to commit
- Byzantine General's Problem: distributed decision making with malicious failures
  - One general, n-1 lieutenants: some number of them may be malicious (often \( f \) of them)
  - All non-malicious lieutenants must come to same decision
  - If general not malicious, lieutenants must follow general
  - Only solvable if \( n \geq 3f+1 \)
- Remote Procedure Call (RPC): Call procedure on remote machine
  - Provides same interface as procedure
  - Automatic packing and unpacking of arguments without user programming (in stub)