Review: Reliable Message Delivery: the Problem

- All physical networks can garble and/or drop packets
  - Physical media: packet not transmitted/received
    » If transmit close to maximum rate, get more throughput – even if some packets get lost
    » If transmit at lowest voltage such that error correction just starts correcting errors, get best power/bit
  - Congestion: no place to put incoming packet
    » Point-to-point network: insufficient queue at switch/router
    » Broadcast link: two host try to use same link
    » In any network: insufficient buffer space at destination
    » Rate mismatch: what if sender send faster than receiver can process?

- Reliable Message Delivery on top of Unreliable Packets
  - Need some way to make sure that packets actually make it to receiver
    » Every packet received at least once
    » Every packet received at most once
  - Can combine with ordering: every packet received by process at destination exactly once and in order

Goals for Today

- Finish Discussion of TCP/IP
- Messages
  - Send/receive
  - One vs. two-way communication
- Distributed Decision Making
  - Two-phase commit/Byzantine Commit
- Remote Procedure Call
- Distributed File Systems (Part I)

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 ack gets dropped? Or if message gets delayed?
    » Sender doesn’t get ack, retransmits. Receiver gets message twice, acks each.
How to deal with message duplication

- **Solution**: put sequence number in message to identify re-transmitted packets
  - Receiver checks for duplicate #’s: Discard if detected
- **Requirements**:
  - Sender keeps copy of unack’ed messages
    - Easy: only need to buffer messages
  - Receiver tracks possible duplicate messages
    - Hard: when ok to forget about received message?

**Alternating-bit protocol**:
- Send one message at a time; don’t send next message until ack received
- Sender keeps last message; receiver tracks sequence # of last message received
- **Pros**: simple, small overhead
- **Con**: Poor performance
  - Wire can hold multiple messages; want to fill up at (wire latency x throughput)
  - Con: doesn’t work if network can delay or duplicate messages arbitrarily

---

Better messaging: Window-based acknowledgements

- **Windowing protocol (not quite TCP)**:
  - Send up to N packets without ack
    - Allows pipelining of packets
  - Each packet has sequence number
    - Receiver acknowledges each packet
    - Ack says “received all packets up to sequence number X”/send more
  - Acks serve dual purpose:
    - Reliability: Confirming packet received
    - Ordering: Packets can be reordered at destination
- **What if packet gets garbled/dropped?**
  - Sender will timeout waiting for ack packet
    - Should receiver discard packets that arrive out of order?
    - Simple, but poor performance
    - Alternative: Keep copy until sender fills in missing pieces?
    - Reduces # of retransmits, but more complex
- **What if ack gets garbled/dropped?**
  - Timeout and resend just the un-acknowledged packets

---

Transmission Control Protocol (TCP)

- **Transmission Control Protocol (TCP)**
  - TCP (IP Protocol 6) layered on top of IP
  - Reliable byte stream between two processes on different machines over Internet (read, write, flush)
- **TCP Details**
  - Fragments byte stream into packets, hands packets to IP
    - IP may also fragment by itself
  - Uses window-based acknowledgement protocol (to minimize state at sender and receiver)
    - “Window” reflects storage at receiver – sender shouldn’t overrun receiver’s buffer space
    - Also, window should reflect speed/capacity of network – sender shouldn’t overload network
  - Automatically retransmits lost packets
  - Adjusts rate of transmission to avoid congestion
    - A “good citizen”

---

TCP Windows and Sequence Numbers

- **Sender has three regions**:  
  - Sequence regions
    - sent and ack’ed
    - Sent and not ack’ed
    - Not yet sent
  - Window (colored region) adjusted by sender
- **Receiver has three regions**:
  - Sequence regions
    - received and ack’ed (given to application)
    - received and buffered
    - not yet received (or discarded because out of order)
Window-Based Acknowledgements (TCP)

- Every message encodes Sequence number and Ack
- Can include data for forward stream and/or ack for reverse stream

Selective Acknowledgement Option (SACK)

- 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)

Administrivia

- Project 4 design document:
  - Extension to Wednesday night
- Final Exam
  - Thursday 12/16, 8:00AM-11:00AM, 10 Evans
  - All material from the course
    - With slightly more focus on second half, but you are still responsible for all the material
  - Two sheets of notes, both sides
  - Will need dumb calculator (No phones, devices with net)
- There is a lecture on Wednesday
  - Including this one, we are down to 4 lectures...!
- Optional Final Lecture: Monday 12/6
  - Send me topics you might want to hear about
  - Won’t be responsible for topics on Final
  - Starting to get interesting suggestions!
  - Examples:
    - Realtime OS, Secure Hardware, Quantum Computing
    - Dragons... Etc.

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
- 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 involves 5 values:
    [ Client Addr, Client Port, Server Addr, Server Port, Protocol ]
  - Often, Client Port “randomly” assigned
    » Done by OS during client socket setup
  - Server Port often “well known”
    » 80 (web), 443 (secure web), 25 (sendmail), etc
    » Well-known ports from 0—1023
  - Note that the uniqueness of the tuple is really about two
    Addr/Port pairs and a protocol

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()
Sockect sock = new Socket("169.229.60.38",6013);
BufferedReader bin =
  new BufferedReader( 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

Using Messages: Send/Receive behavior

• When should 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 receiver actually received the message?
  - When can sender reuse the memory containing message?

• Mailbox provides 1-way communication from T1→T2
  - T1→buffer→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);
  ```
  Server: (responding with the file)
  ```c
  char command[1000], answer[1000];
  receive(command, server_mbox);
  decode command;
  read file into answer;
  send(answer, client_mbox);
  ```
General’s Paradox

- General’s 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

Two-Phase Commit

- Since we can’t solve the General’s Paradox (i.e., simultaneous action), let’s solve a related problem
- Two-Phase Commit: Two machines agree to do something, or not do it, atomically

Two-phase commit example

- Simple Example: A=WellsFargo Bank, B=Bank of America
  - Phase 1: Prepare Phase
    » A writes “Begin transaction” to log
    » A→B: 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
  - Different failure modes 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” to its log,
    » Site A crashes
    » Site B wakes up, checks 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
  - What happens if one or more of the nodes is malicious?
    - Malicious: attempting to compromise the decision making
**Byzantine General’s Problem**

- 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→Read (“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 or 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.

• 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)