Basic Observation

• Some types of network functionality can only be correctly implemented end-to-end
  – Reliability, security, etc.

• Because of this, end hosts:
  – Can satisfy the requirement without network’s help
  – Will/must do so, since can’t rely on network’s help

• Therefore don’t go out of your way to implement them in the network

Example: Reliable File Transfer

• Solution 1: make each step reliable, and then concatenate them

• Solution 2: end-to-end check and try again if necessary
Discussion

• Solution 1 is incomplete
  – What happens if memory is corrupted?
  – Receiver has to do the check anyway!

• Solution 2 is complete
  – Full functionality can be entirely implemented at application layer with no need for reliability from lower layers

• Is there any need to implement reliability at lower layers?
  – Well, it could be more efficient

End-to-End Principle

Implementing this functionality in the network:
• Doesn’t reduce host implementation complexity
• Does increase network complexity
• Probably imposes delay and overhead on all applications, even if they don’t need functionality

• However, implementing in network can enhance performance in some cases
  – e.g., very lossy link

Conservative Interpretation of E2E

Don’t implement a function at the lower levels of the system unless it can be completely implemented at this level

Or,
Unless you can relieve the burden from hosts, don’t bother

Moderate Interpretation

• Think twice before implementing functionality in the network

• If hosts can implement functionality correctly, implement it in a lower layer only as a performance enhancement

• But do so only if it does not impose burden on applications that do not require that functionality

• This is the interpretation we are using
Goals of Today’s Lecture

- Reliable Messaging
- RPCs
- Two-Phase Commit

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

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, ACK each

How to Deal with Message Duplication?

- Solution: put sequence number in message to identify re-transmitted packets
  - Receiver checks for duplicate number's; Discard if detected

- Requirements:
  - Sender keeps copy of unACK'd 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 number 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
**Windowing protocol (not quite TCP):**
- Send up to \( N \) packets without ack
  - Allows pipelining of packets
  - Window size (\( N \)) < queue at destination
- 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
  - Receiver gets packets 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)**

- 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'd
    - sent but not ACK'd
    - not yet sent
  - Window (colored region) adjusted by sender

- Receiver has three regions:
  - Sequence regions
    - received and ACK'd (given to application)
    - received but buffered
    - not yet received (or discarded because out of order)
**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”

**Remote Procedure Call (RPC)**

- 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
- Another 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");`

**Goals of Today’s Lecture**

- Reliable Messaging
- RPCs
- Two-Phase Commit

**RPC 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 Information Flow**

Client (caller) → Client Stub → Packet Handler
- Call
- Bundle args
- Send
- Receive
- Unbundle ret vals
- Return

Server (callee) → Server Stub → Packet Handler
- Call
- Bundle ret vals
- Send
- Receive
- Unbundle args
- Return

**RPC Details (1/3)**

- 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

**RPC Details (2/3)**

- Cross-platform issues:
  - What if client/server machines are different architectures/languages?
    - Convert everything to/from some canonical form
    - Tag every item with an indication of how it is encoded (avoids unnecessary conversions)

- 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

**RPC Details (3/3)**

- 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 dist. 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

Problems with RPC: 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

Administrivia

- Midterm 3 coming up on Wed 4/25 6:30-8PM
  - All topics up to and including Lecture 23
    » Focus will be on Lectures 17 – 23 and associated readings, and Projects 3
    » But expect 20-30% questions from materials from Lectures 1-16
  - LKS 245, Hearst Field Annex A1, VLSB 2060, Barrows 20, Wurster 102 (see Piazza for your room assignment)
  - Closed book
  - 2 pages hand-written notes both sides

BREAK
Goals of Today’s Lecture

- TCP flow control
- RPCs
- Two-Phase Commit

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

General’s Paradox

- 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 General’s Paradox (i.e. simultaneous action), let’s solve a related problem

  - Distributed transaction: Two or more machines agree to do something, or not do it, atomically

  - Two-Phase Commit protocol: Developed by Turing award winner Jim Gray (first Berkeley CS PhD, 1969)
Two-Phase Commit Protocol

- Persistent stable log on each machine: 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 used to guarantee that all machines either commit or don't

2PC Algorithm

- One coordinator
- N workers (replicas)
- High level algorithm description:
  - Coordinator asks all workers if they can commit
  - If all workers reply "VOTE-COMMIT", then coordinator broadcasts "GLOBAL-COMMIT"
    - Otherwise coordinator broadcasts "GLOBAL-ABORT"
  - Workers obey the GLOBAL messages
- Use a persistent, stable log on each machine to keep track of what you are doing
  - If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash

Detailed Algorithm

Coordinator sends VOTE-REQ to all workers

- If receive VOTE-COMMIT from all N workers, send GLOBAL-COMMIT to all workers
- If doesn't receive VOTE-COMMIT from all N workers, send GLOBAL-ABORT to all workers

Worker Algorithm

- Wait for VOTE-REQ from coordinator
  - If ready, send VOTE-COMMIT to coordinator
  - If not ready, send VOTE-ABORT to coordinator
    - And immediately abort
- If receive GLOBAL-COMMIT then commit
  - If receive GLOBAL-ABORT then abort

Failure Free Example Execution

coordinator

worker 1  VOTE-REQ GLOBAL-COMMIT

worker 2

worker 3  VOTE-COMMIT
State Machine of Coordinator

- Coordinator implements simple state machine:

  - **INIT**
    - Send: VOTE-REQ
  - **WAIT**
    - Recv: START
    - Send: VOTE-REQ
    - Recv: VOTE-ABORT
      - Send: GLOBAL-ABORT
    - Recv: all VOTE-COMMIT
      - Send: GLOBAL-COMMIT
  - **ABORT**
  - **COMMIT**

Dealing with Worker Failures

- Failure only affects states in which the coordinator is waiting for messages
- Coordinator only waits for votes in "WAIT" state
- In **WAIT**, if doesn’t receive N votes, it times out and sends GLOBAL-ABORT

Example of Worker Failure

- Coordinator
- **WAIT**
  - ABORT
  - COMMIT
- Worker 1
  - VOTE-REQ
  - VOTE-COMMIT
- Worker 2
  - VOTE-COMMIT
- Worker 3
  - GLOBAL-ABORT
  - Time
Dealing with Coordinator Failure

- Worker waits for VOTE-REQ in INIT
  - Worker can time out and abort (coordinator handles it)
- Worker waits for GLOBAL-* message in READY
  - If coordinator fails, workers must BLOCK waiting for coordinator to recover and send GLOBAL-* message

Example of Coordinator Failure #1

Example of Coordinator Failure #2

Durability

- All nodes use stable storage to store current state
  - stable storage is non-volatile storage (e.g. backed by disk) that guarantees atomic writes.

- Upon recovery, it can restore state and resume:
  - Coordinator aborts in INIT, WAIT, or ABORT
  - Coordinator commits in COMMIT
  - Worker aborts in INIT, ABORT
  - Worker commits in COMMIT
  - Worker asks Coordinator in READY
## Blocking for Coordinator to Recover

A worker waiting for global decision can ask fellow workers about their state:

- If another worker is in **ABORT** or **COMMIT** state then coordinator must have sent **GLOBAL**-*
  - Thus, worker can safely abort or commit, respectively

- If another worker is still in **INIT** state then both workers can decide to abort

- If all workers are in ready, need to **BLOCK** (don’t know if coordinator wanted to abort or commit)

## Distributed Decision Making Discussion (1/2)

- **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

## Distributed Decision Making Discussion (2/2)

- 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

## PAXOS

- **PAXOS**: An alternative used by Google and others that does not have this blocking problem
  - Develop by Leslie Lamport (Turing Award Winner)

- 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 and 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 the following Integrity Constraints apply:
  - 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

Summary

- Remote Procedure Call (RPC): Call procedure on remote machine
  - Provides same interface as procedure
  - Automatic packing/unpacking of args without user programming
- Two-phase commit: distributed decision making
  - First, make sure everyone guarantees 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 ≥ 3f+1