Recall: Use of Erasure Coding in general:
- Centralized vs Distributed Systems
  - Centralized System: System in which major functions are performed by a single physical computer
  - Distributed System: physically separate computers working together on some task
  - Later models: peer-to-peer wide-spread collaboration

Operating Systems and Server Systems Programming
- Easier to add power incrementally
- Users can have complete control over some components
- Collaboration: much easier for users to collaborate through network resources (such as network file systems)

Fraction Blocks Lost
- Distributed Systems: Reality
  - Reality has been disappointing
  - A distributed system is one where you do not know how many machines are in the system or what the other machines are doing
  - Worse reliability, worse performance:
    - Can lose data if any machine crashes
    - If any machine crashes, other machines may continue to deliver adequate service
    - How can one scale?
  - Corollary of Lamport’s quote: “A distributed system is one where you can’t even perform a protocol correctly
    - Can an application be made secure in the face of other failures?

Networking Definitions
- WAN: a network over a larger area
- Local Area Network (LAN): a network over a smaller area
- Physical Medium: copper, fiber optic, wireless, etc.
- Media Access Control (MAC) protocols
- TCP/IP stack

What Is a Protocol?
- A protocol is an agreement on how to communicate, including:
  - Format, order messages are sent and received
  - Interaction of processes
  - Security: what a communication means
  - Session: what is happening on the network
- The OSI Model
  - Application layer
  - Presentation layer
  - Session layer
  - Transport layer
  - Network layer
  - Data link layer
  - Physical layer
- The Internet Hourglass
  - There is just one network-layer protocol, IP.
  - The “narrow waist” facilitates interoperability.

Examples of Protocols in Human Interactions
- Telephone
  - Dial phone number, hear ringing
  - Call “Hi, this is [me].”
  - “Hi, how do you think...”
  - “Hi, how do you think...”

Global Communication: The Problem
- Many different applications
  - Email, web, FTP, etc.
- Many different network styles and technologies
  - Wireless vs. wired, etc.
- How do we organize this?
- Re-implement every application for every technology?

Implications of Hourglass
- Single Internet-layer module (IP)
  - Allows arbitrary networks to interoperate
  - Any network technology that supports IP can even perform a protocol correctly
  - Can you trust the other members of a distributed application enough to
  - Even perform a protocol correctly?

Drawbacks of Layering
- Layer N may duplicate layer N-1 functionality
- E.g., some recovery to retransmit lost data
- Layers may need some information
- E.g., timestamps, maximum transmission unit size
- Layering can hurt performance
- E.g., hiding details about what is really going on
- Some layers are not always cleanly separated
- Inter-layer dependencies for performance reasons
- Some dependencies in standards (header checksums)
- Headers start to get really big
- Sometimes header bytes >> actual content

Centralized vs Peer-to-Peer
- Centralized System: System in which major functions are performed by a single physical computer
- Peer-to-Peer Model
  - Easier to add power incrementally
  - Users can have complete control over some components
  - Collaboration: much easier for users to collaborate through network resources (such as network file systems)

Distributed Systems: Goals/Requirements
- Transparency: the ability of the system to mask its complexity
- Better reliability through use of log
  - layer N may duplicate layer N-1 functionality
  - Data transfer for data for reliability
- Fault-tolerance can be accomplished with NVRAM
- Distributed systems may not be updated immediately, data
- Distribution between “Log Structured” and “Journalled”
  - In Log Structured filesystem, data stays in log form
  - In Journalled filesystem, Log used for recovery
  - Journaling File System
  - Applications updates in system metadata using transactions (using logs, etc.)
  - Updates to non-directory files (i.e., user stuff) can be done in parallel
  - Log structured file systems

Solution: Intermediate Layers
- Introduce intermediate layers that provide sets of abstractions for various network functionality & technologies
  - A new vendor-independent API
  - Vendors can implement API
  - What would be easy in a centralized system becomes a lot more difficult
  - Can you trust the other members of a distributed application enough to
  - Even perform a protocol correctly?

The Hourglass Model
- The narrow waist facilitating interoperability.
Administrivia

- Last Midterm: 5/2
- Can have 4 handwritten sheets of notes – both sides
- Focus on material from lectures 17-24, but any topic fair game!
- Don’t forget the story/group evaluation!
- Very important to help you understand your group dynamics

Optional HW2
- Will give you a chance to try out using the language “Go” to build a two-phase commit protocol
- You will be testing it out for next term
- Not sure that we will be giving out points for it. Stay tuned!

End-To-End Argument

- Hugely influential paper: “End-to-End Arguments in System Design” by Saltzer, Reed, and Clark (’74)
- “Sacred Text” of the Internet
- Endless disputes about what it means
- Everyone cites it as supporting their position

- Simple Message: Domain level HMV will only be correctly implemented end-to-end
- Reliability, security, etc.
- Because of this, and hosts:
- Can satisfy the requirement without network’s help
- Will must do, so can’t rely on network’s help
- Therefore don’t go out of your way to implement them in the network

Implementation complexity in the network:

- Doesn’t redude 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 noisy link

- Can have 3 handwritten sheets of notes – both sides
- Need to synchronize multiple threads, running on different machines

- What happens if memory is corrupted?
- No shared memory, so cannot use test&set

- Receiver has to do the check anyway!

- 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
  - Can satisfying the requirement without network’s help
  - Well, it could be more efficient

Moderate Interpretation

- Think twice before implementing functionality in the network
- It might be able to 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 don’t require that functionality
- This is the interpretation we are using

General’s Paradox

- Can messages ever on unreliable network be used to guarantee two entities do something simultaneously?
- Remarkably, “no”, even if messages get through

General’s Paradox (cont’d)

- Can messages ever on unreliable network be used to guarantee two entities do something simultaneously?

General’s Paradox (cont’d)

- No constraints on time, just that it will eventually happen

General’s Paradox (cont’d)

- How do we make sure that decisions cannot get forgotten?

Distributed Consensus Making

- Consensus problem
  - All nodes propose a value
  - Some nodes might crash and stop responding
  - Eventually, all remaining nodes decide on the same value from set of proposed values

- Distributed Decision Making
  - Choose between “true” and “false”

- Or choose between “commit” and “rollback”

- Equally important (but often forgotten), make it durable!

- How do we make sure that decisions cannot get forgotten?

- This is the “D” of “ACID” in a regular database

- In a global-scale system?

- What about erasure coding or massive replication?

Distributed Applications

- One Abstraction: send/receive messages
- Applications cannot generate messages

- Interface:
  - Sends
  - Receives
  - Mailbox (mbox)
  - Includes both destination location and queue

- Mailbox: handled by send/receive

- In real life, use radio for simultaneous (out of band)

- All nodes propose a value
  - Choose between “true” and “false”

- Some nodes might crash and stop responding

- Choose between “commit” and “abort”

- Name a file to some other machine
  - No shared memory, so cannot use test&set

- Namely after Choo, who died at Little Big Horn because he arrived a couple of days too early

- No way to be sure last message gets though

- Use radio, use random numbers for simultaneous (out of band) communication

- So clearly, we need something other than simultaneously!

Messaging for Request/Response Communication

- Request/Response
  - Send an HTTP request to a remote machine
  - A web page from a remote server web server
  - A chat window from a remote chat server

- Server provides “lease” (or lease time) to the client

Distributed Consensus Making

- Consensus problem
  - All nodes propose a value
  - Some nodes might crash and stop responding

- Eventually, all remaining nodes decide on the same value from set of proposed values

- Distributed Decision Making
  - Choose between “true” and “false”

- Or choose between “commit” and “rollback”

- Equally important (but often forgotten), make it durable!

- How do we make sure that decisions cannot get forgotten?

- This is the “D” of “ACID” in a regular database

- In a global-scale system?

- What about erasure coding or massive replication?

- Life Business applications...
2PC Algorithm

- One coordinator
- N workers (replicas)

High level algorithm description:
- Coordinator asks all workers if they can commit
- If no worker replies "YES COMMIT", then coordinator aborts

Detailed Algorithm

<table>
<thead>
<tr>
<th>Coordinator Algorithm</th>
<th>Worker Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordinator sends &quot;VOTE-REQ&quot; to all workers</td>
<td>Worker waits for votes in &quot;WAIT&quot; state</td>
</tr>
<tr>
<td>If Coordinator receives &quot;VOTE-COMMIT&quot; from all workers, sends GLOBAL-COMMIT to all workers</td>
<td>- Finaly, send &quot;VOTE-COMMIT&quot; to all workers</td>
</tr>
<tr>
<td>- But if Coordinator does not receive N votes, it times out</td>
<td>- Finally, send &quot;VOTE-COMMIT&quot; to all workers</td>
</tr>
<tr>
<td>- Worker aborts in &quot;ABORT&quot; state</td>
<td>- And immediately abort</td>
</tr>
</tbody>
</table>

Failure Free Example Execution

- Coordinator initiates the operation
- Worker 1 waits for vote, times out, and aborts
- Worker 2 and Worker 3 send "COMMIT" to the coordinator
- Coordinator sends "GLOBAL-COMMIT" to all workers

Dealing with Worker Failures

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

State Machine of Workers

- Coordinator only waits for votes in "WAIT" state
- If ready, send "VOTE-COMMIT" to all workers
- If a machine crashes, when it wakes up it first checks its log to recover state or world at time of crash

Example of Worker Failure

- Coordinator initiates the operation
- Worker 1 times out and aborts
- Worker 2 and Worker 3 send "COMMIT" to the coordinator
- Coordinator sends "GLOBAL-COMMIT" to all workers

Dealing with Coordinator Failure

- Worker waits for "VOTE-REQ" in INIT state
- Worker can time out and abort (Coordinator handles it)
- Worker waits for "GLOBAL-*" message in READY state then both workers can decide to abort

Failure Free Example Execution

- Coordinator initiates the operation
- Worker 1 waits for vote, times out, and aborts
- Worker 2 and Worker 3 send "COMMIT" to the coordinator
- Coordinator sends "GLOBAL-COMMIT" to all workers

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 "failure" (different modes listed)
- 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 blocked 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

Alternatives to 2PC

- Three-Phase Commit: One more phase, allows nodes to fail or block and still make progress
- Paxos: An alternative used by Google and others that does not have 2PC blocking problem
- Raft: A more simple protocol

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 continue in COMMIT
  - Worker aborts in INIT, ABORT
  - Worker continues 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 the worker is in COMMIT state, then the worker must have sent GLOBAL-COMMIT
- If another worker is still in INIT state then both workers can decide to abort
- If all workers are in ready to BLOCK (don't know if coordinated wanted to abort or commit)

Byzantine General’s Problem

- Byzantine Generals Problem (n players)
  - Some number of these n-1 are 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 same order
Byzantine General's Problem (con't)
• Impossibility Result:
  - Cannot solve Byzantine Generals Problem with n < 3 because one malicious node can block all messages
  - With f faults, need n > 3f + 1 to solve problem
• Various algorithms exist to solve problem
  - Original algorithm: no messages exchanged in 3-turn
  - Newer algorithms have messages exchanged in n-turn
  - One turn: FIFO, for instance (Costas and Lopez, 1998)
• Use of BFT (Byzantine Fault Tolerance) algorithms
  - Allow multiple machines to make a coordinated decision even if some subset of them (say 2f) fail
RPC Implementation
• Request-response message passing (under covers!)
  - "Stub" provides glue on client/server
  - 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.

Remote Procedure Call (RPC)
• RPC Details (1/3)
  - Equivalence with regular procedure call:
    - Parameters: Request Message
    - Result: Reply message
    - Name of Procedure: Passed in request message
    - Return Address: included (client return data)
  - Stub generator: Compiler that generates stubs
    - Interface definitions in an "interface definition language"
    - Contains, among other things, types of arguments
    - Output: stub code in the appropriate source language
    - Code for clients: pack message, send off, wait for result, unpack result, return to caller
    - Code for server: unpack message, call procedure, pack results, send off
• Another option: Remote Procedure Call (RPC)
  - Calls a procedure on a remote machine
  - Communication between a client and server
    - Parameters and result may be complex data structures
    - Parameters can be passed by value or reference
  - Multiple machines can be involved

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

Microservice operating systems
• Example: split microservice into independent servers
  - File system looks remote, even though on same machine
  - Two-phase commit: distributed decision making
    - First, make sure everyone guarantees that they will commit if
      - All nodes agree
    - Next, ask everyone to commit

Problems with RPC: Performance
• Cost of Procedure call = same-machine RPC + network RPC
  - Caching can help, but may make failure handling complex