Recall: Use of Erasure Coding in general: High Durability/overhead ratio!

- Use of Erasure Coding: Exploit law of large numbers for durability!
  - Assuming independent failures
  - Using, for instance, a Reed-Solomon code
- 6 month repair, FBLPY with 4x increase in total size of data:
  - Replication (4 copies): 0.03 (i.e. 3% blocks lost / year)
  - Fragmentation (16 of 64 fragments needed): 10^{-35} (i.e. 10^{-33} lost / year)

Recall: Transactional File Systems

- Better reliability through use of log
  - All changes are treated as transactions
  - A transaction is committed once it is written to the log
    - Data forced to disk for reliability
    - Process can be accelerated with NVRAM
  - Although File system may not be updated immediately, data preserved in the log
- Difference between “Log Structured” and “Journaled”
  - In a Log Structured filesystem, data stays in log form
  - In a Journaled filesystem, Log used for recovery

Journaling File System

- Applies updates to system metadata using transactions (using logs, etc.)
- Updates to non-directory files (i.e., user stuff) can be done in place (without logs), full logging optional
  - Ex: NTFS, Apple HFS+, Linux XFS, JFS, ext3, ext4
- Full Logging File System
  - All updates to disk are done in transactions

Societal Scale Information Systems

- The world is a large distributed system
  - Microprocessors in everything
  - Vast infrastructure behind them
  - Internet Connectivity
  - Databases
  - Information Collection
  - Remote Storage
  - Online Games
  - Commerce
  - MEMS for Sensor Nets
Centralized vs Distributed Systems

- **Centralized System**: System in which major functions are performed by a single physical computer
  - Originally, everything on single computer
  - Later: client/server model
- **Distributed System**: physically separate computers working together on some task
  - Early model: multiple servers working together
    » Probably in the same room or building
    » Often called a “cluster”
  - Later models: peer-to-peer/wide-spread collaboration

Distributed Systems: Motivation/Issues/Promise

- Why do we want distributed systems?
  - Cheaper and easier to build lots of simple computers
  - 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)

- The *promise* of distributed systems:
  - *Higher availability*: one machine goes down, use another
  - *Better durability*: store data in multiple locations
  - *More security*: each piece easier to make secure

Distributed Systems: Reality

- Reality has been disappointing
  - *Worse availability*: depend on every machine being up
    » Lamport: “A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.”
  - *Worse reliability*: can lose data if any machine crashes
  - *Worse security*: anyone in world can break into system
- Coordination is more difficult
  - Must coordinate multiple copies of shared state information (using only a network)
  - What would be easy in a centralized system becomes a lot more difficult
- *Trust/Security/Privacy/Denial of Service*
  - Many new variants of problems arise as a result of distribution
  - Can you trust the other members of a distributed application enough to even perform a protocol correctly?
  - Corollary of Lamport’s quote: “A distributed system is one where you can’t do work because some computer you didn’t even know existed is successfully coordinating an attack on my system!”

Distributed Systems: Goals/Requirements

- **Transparency**: the ability of the system to mask its complexity behind a simple interface
  - Possible transparencies:
    - *Location*: Can’t tell where resources are located
    - *Migration*: Resources may move without the user knowing
    - *Replication*: Can’t tell how many copies of resource exist
    - *Concurrency*: Can’t tell how many users there are
    - *Parallelism*: System may speed up large jobs by splitting them into smaller pieces
    - *Fault Tolerance*: System may hide various things that go wrong
  - Transparency and collaboration require some way for different processors to communicate with one another
Networking Definitions

- **Network**: physical connection that allows two computers to communicate
- **Packet**: unit of transfer, sequence of bits carried over the network
  - Network carries packets from one CPU to another
  - Destination gets interrupt when packet arrives
- **Protocol**: agreement between two parties as to how information is to be transmitted

What Is A Protocol?

- A protocol is an agreement on how to communicate, including:
  - **Syntax**: how a communication is specified & structured
    - Format, order messages are sent and received
  - **Semantics**: what a communication means
    - Actions taken when transmitting, receiving, or when a timer expires
- Described formally by a state machine
  - Often represented as a message transaction diagram
  - Can be a partitioned state machine: two parties synchronizing duplicate sub-state machines between them
  - Stability in the face of failures!

Examples of Protocols in Human Interactions

- **Telephone**
  1. (Pick up / open up the phone)
  2. Listen for a dial tone / see that you have service
  3. Dial
  4. Should hear ringing …
  5. Caller: “Hi, it’s John…..”
  6. Or: “Hi, it’s me” (← what’s that about?)
  7. Caller: “Hey, do you think … blah blah blah …” pause
  8. Callee: “Hello?”
  9. Callee: “Yeah, blah blah blah …” pause
  10. Caller: Bye
  11. Call: Bye
  12. Hang up

Global Communication: The Problem

- Many different applications
  - email, web, P2P, etc.
- Many different network styles and technologies
  - Wireless vs. wired vs. optical, etc.
- How do we organize this mess?
  - Re-implement every application for every technology?
- No! But how does the Internet design avoid this?
Solution: Intermediate Layers

- Introduce intermediate layers that provide set of abstractions for various network functionality & technologies
  - A new app/media implemented only once
  - Variation on “add another level of indirection”
- Goal: Reliable communication channels on which to build distributed applications

The Internet Hourglass

- There is just one network-layer protocol, IP. The “narrow waist” facilitates interoperability.

Implications of Hourglass

- Single Internet-layer module (IP):
  - Allows arbitrary networks to interoperate
    - Any network technology that supports IP can exchange packets
  - Allows applications to function on all networks
    - Applications that can run on IP can use any network
  - Supports simultaneous innovations above and below IP
    - But changing IP itself, i.e., IPv6, very involved

Drawbacks of Layering

- Layer N may duplicate layer N-1 functionality
  - E.g., error recovery to retransmit lost data
- Layers may need same 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
**Administrivia**

- Last Midterm: 5/2
  - Can have 3 handwritten sheets of notes – both sides
  - Focus on material from lecture 17-24, but all topics fair game!
- Don’t forget to do your group evaluations!
  - Very important to help us understand your group dynamics
- Optional HW4 will come out soon
  - 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 ('84)
- “Sacred Text” of the Internet
  - Endless disputes about what it means
  - Everyone cites it as supporting their position
- Simple Message: 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 complex 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

• Is this still valid?
  – What about Denial of Service?
  – What about Privacy against Intrusion?

  – Perhaps there are things that must be in the network???

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)
    – 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:
  ```c
  Producer:
  int msg1[1000];
  while(1) {
      prepare message;
      send(msg1, mbox);
  }
  
  Consumer:
  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
  – Next time: will discuss fact that this is one of the roles 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
  ```c
  Client: (requesting the file)
  char response[1000];
  send("read rutabaga", server_mbox);
  receive(response, client_mbox);
  
  Server: (responding with the file)
  char command[1000], answer[1000];
  receive(command, server_mbox);
  decode command;
  read file into answer;
  send(answer, client_mbox);
  ```

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 “abort”
• Equally important (but often forgotten!): make it durable!
  – How do we make sure that decisions cannot be forgotten?
    » This is the “D” of “ACID” in a regular database
  – In a global-scale system?
    » What about erasure coding or massive replication?
    » Like Blockchain applications!
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

General's Paradox (con’t)

- 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!
  - In real life, use radio for simultaneous (out of band) communication
  - So, clearly, we need something other than simultaneity!

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

- Two-Phase Commit protocol: Developed by Turing award winner Jim Gray
  - (first Berkeley CS PhD, 1969)
  - Many important DataBase breakthroughs also from Jim Gray

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

Failure Free Example Execution

Coordinator Algorithm
- Coordinator sends VOTE-REQ to all workers
- 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 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
- If receive GLOBAL-COMMIT then commit
- If receive GLOBAL-ABORT then abort

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

State Machine of Coordinator
- Coordinator implements simple state machine:
  - INIT
  - WAIT
  - ABORT
  - COMMIT

- Recv: START Send: VOTE-REQ
- Recv: all VOTE-COMMIT Send: GLOBAL-COMMIT
- Recv: VOTE-ABORT Send: GLOBAL-ABORT
State Machine of Workers

**INIT**
- Receive: VOTE-REQ
- Send: VOTE-ABORT

**READY**
- Receive: VOTE-REQ
- Send: VOTE-COMMIT

**ABORT**
- Receive: GLOBAL-ABORT

**COMMIT**
- Receive: GLOBAL-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: INIT → WAIT → ABORT
- Worker 1: VOTE-REQ
- Worker 2: VOTE-COMMIT
- Worker 3: Global-ABORT

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 abst 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 abst 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

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
  – Develop by Leslie Lamport (Turing Award Winner)
  – No fixed leader, can choose new leader on fly, deal with failure
  – Some think this is extremely complex!
• RAFT: PAXOS alternative from John Osterhout (Stanford)
  – Simpler to describe complete protocol

• 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
Byzantine General's Problem (con't)

- 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 (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");
    ```
**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)
- **Server** (callee)
- **Packet Handler**

**RPC Details (1/3)**

- Equivalence with regular procedure call
  - Parameters $\leftrightarrow$ Request Message
  - Result $\leftrightarrow$ 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

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

  ![Diagram of Monolithic Structure and Microkernel Structure]

  - 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.

Summary (1/2)

- Protocol: Agreement between two parties as to how information is to be transmitted
- E2E argument encourages us to keep Internet communication simple
  - If higher layer can implement functionality correctly, implement it in a lower layer only if:
    - it improves the performance significantly for application that need that functionality, and
    - it does not impose burden on applications that do not require that functionality
- Two-phase commit: distributed decision making
  - First, make sure everyone guarantees that they will commit if asked (prepare)
  - Next, ask everyone to commit

Summary (2/2)

- 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

- BlockChain protocols
  - Could be used for distributed decision making

- 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)