CS162
Operating Systems and Systems Programming
Lecture 22

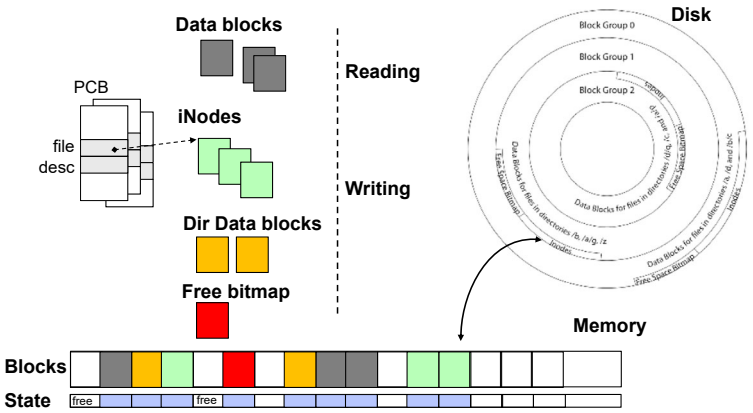
Transactions (Con't), End-to-End Arguments, Distributed Decision Making

November 16th, 2020

Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Recall: File System Buffer Cache

- OS implements a cache of disk blocks for efficient access to data, directories, inodes, freemap



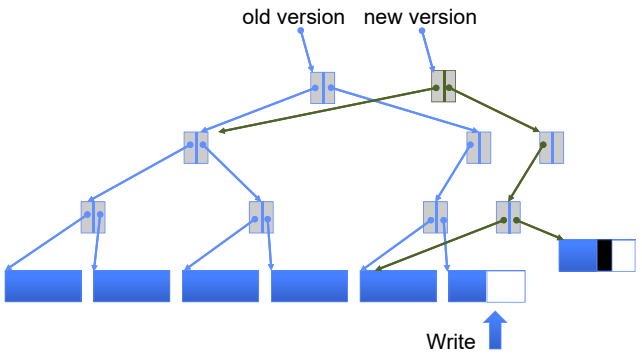
Recall: Important "ilities"

- Availability: the probability that the system can accept and process requests
  - Measured in "nines" of probability: e.g. 99.9% probability is "3-nines of availability"
  - Key idea here is independence of failures
- Durability: the ability of a system to recover data despite faults
  - This idea is fault tolerance applied to data
  - Doesn't necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone
- Reliability: the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition)
  - Usually stronger than simply availability: means that the system is not only "up", but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Must make sure data survives system crashes, disk crashes, other problems

Recall: RAID 6 and other Erasure Codes

- In general: RAIDX is an "erasure code"
  - Must have ability to know which disks are bad
  - Treat missing disk as an "Erasure"
- Today, disks so big that: RAID 5 not sufficient!
  - Time to repair disk sooooo long, another disk might fail in process!
  - "RAID 6" – allow 2 disks in replication stripe to fail
  - Requires more complex erasure code, such as EVENODD code (see readings)
- More general option for general erasure code: Reed-Solomon codes
  - Based on polynomials in GF(2^k) (I.e. k-bit symbols)
  - m data points define a degree m polynomial; encoding is n points on the polynomial
    - » $P(x)=a_0+a_1x^1+\ldots a_{m-1}x^{m-1}$
    - » Coded: P(0),P(1),P(2)….,P(n-1)
  - Any m points can be used to recover the polynomial; $n-m$ failures tolerated
- Erasure codes not just for disk arrays. For example, geographic replication
  - E.g., split data into $m = 4$ chunks, generate $n = 16$ fragments and distribute across the Internet
  - Any 4 fragments can be used to recover the original data --- very durable!

Recall: COW with Smaller-Radix Blocks



- If file represented as a tree of blocks, just need to update the leading fringe

More General Reliability Solutions

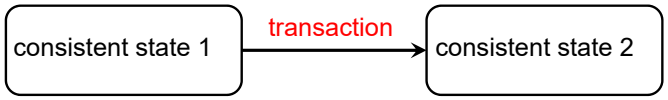
- Use Transactions for atomic updates
  - Ensure that multiple related updates are performed atomically
  - i.e., if a crash occurs in the middle, the state of the systems reflects either all or none of the updates
  - Most modern file systems use transactions internally to update filesystem structures and metadata
  - Many applications implement their own transactions
- Provide Redundancy for media failures
  - Redundant representation on media (Error Correcting Codes)
  - Replication across media (e.g., RAID disk array)

Transactions

- Closely related to critical sections for manipulating shared data structures
- They extend concept of atomic update from memory to stable storage
  - Atomically update multiple persistent data structures
- Many ad-hoc approaches
  - FFS carefully ordered the sequence of updates so that if a crash occurred while manipulating directory or inodes the disk scan on reboot would detect and recover the error (fsck)
  - Applications use temporary files and rename

Key Concept: Transaction

- A *transaction* is an atomic sequence of reads and writes that takes the system from consistent state to another.



- Recall: Code in a critical section appears atomic to other threads
- Transactions extend the concept of atomic updates from *memory* to *persistent storage*

Typical Structure

- Begin a transaction – get transaction id
- Do a bunch of updates
  - If any fail along the way, roll-back
  - Or, if any conflicts with other transactions, roll-back
- Commit the transaction

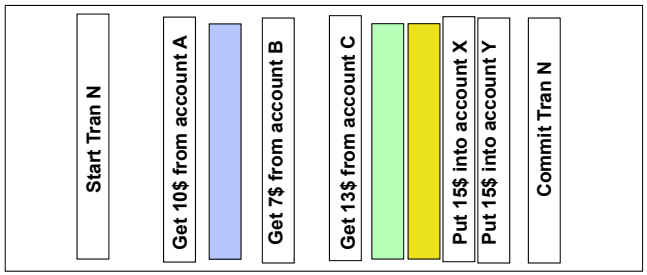
"Classic" Example: Transaction

```
BEGIN;    --BEGIN TRANSACTION
 UPDATE accounts SET balance = balance - 100.00 WHERE
   name = 'Alice';

 UPDATE branches SET balance = balance - 100.00 WHERE
   name = (SELECT branch_name FROM accounts WHERE name
   = 'Alice');

 UPDATE accounts SET balance = balance + 100.00 WHERE
   name = 'Bob';

 UPDATE branches SET balance = balance + 100.00 WHERE
   name = (SELECT branch_name FROM accounts WHERE name
   = 'Bob');
COMMIT;    --COMMIT WORK
```

Transfer $100 from Alice's account to Bob's account

Concept of a log

- One simple action is atomic – write/append a basic item
- Use that to seal the commitment to a whole series of actions



Transactional File Systems

- Better reliability through use of log
  - 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 Systems

- Don't modify data structures on disk directly
- Write each update as transaction recorded in a log
  - Commonly called a journal or intention list
  - Also maintained on disk (allocate blocks for it when formatting)
- Once changes are in the log, they can be safely applied to file system
  - e.g. modify inode pointers and directory mapping
- Garbage collection: once a change is applied, remove its entry from the log

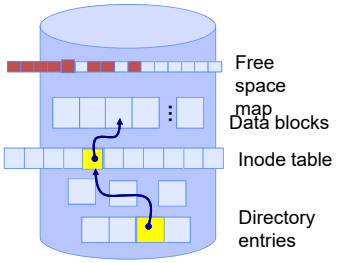
- Linux took original FFS-like file system (ext2) and added a journal to get ext3!
  - Some options: whether or not to write all data to journal or just metadata

- Other examples: NTFS, Apple HFS+, Linux XFS, JFS, ext4

Creating a File (No Journaling Yet)

- Find free data block(s)
- Find free inode entry
- Find dirent insertion point

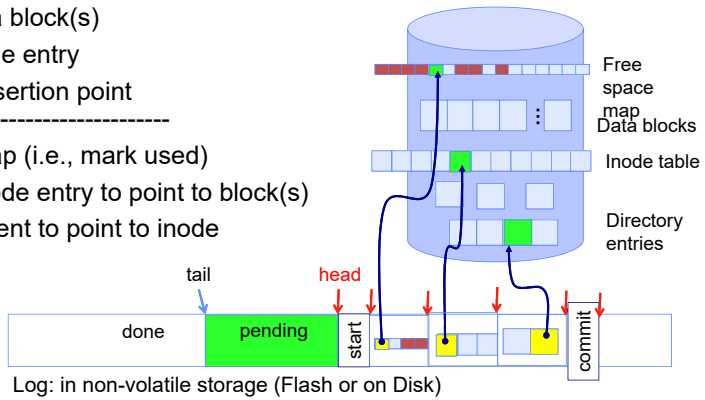
- Write map (i.e., mark used)
- Write inode entry to point to block(s)
- Write dirent to point to inode



Creating a File (With Journaling)

- Find free data block(s)
- Find free inode entry
- Find dirent insertion point

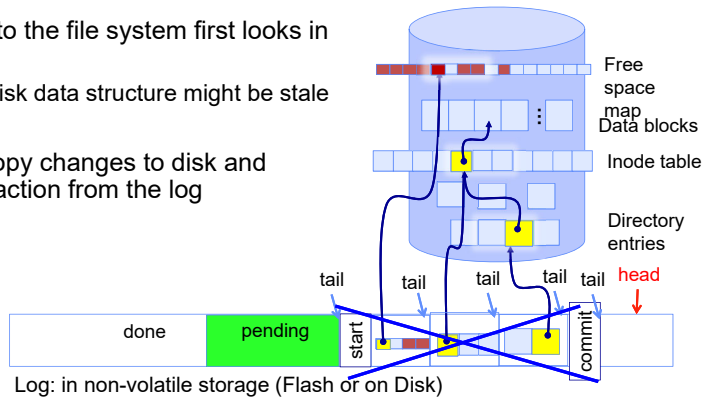
- [log] Write map (i.e., mark used)
- [log] Write inode entry to point to block(s)
- [log] Write dirent to point to inode



Log: in non-volatile storage (Flash or on Disk)

After Commit, Eventually Replay Transaction

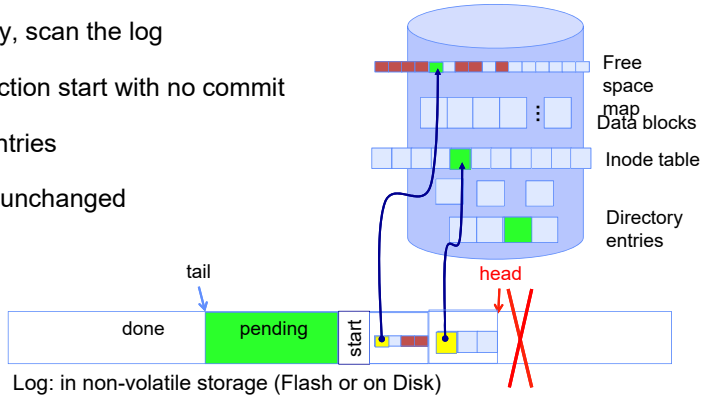
- All accesses to the file system first looks in the log
  - Actual on-disk data structure might be stale
- Eventually, copy changes to disk and discard transaction from the log



Log: in non-volatile storage (Flash or on Disk)

Crash Recovery: Discard Partial Transactions

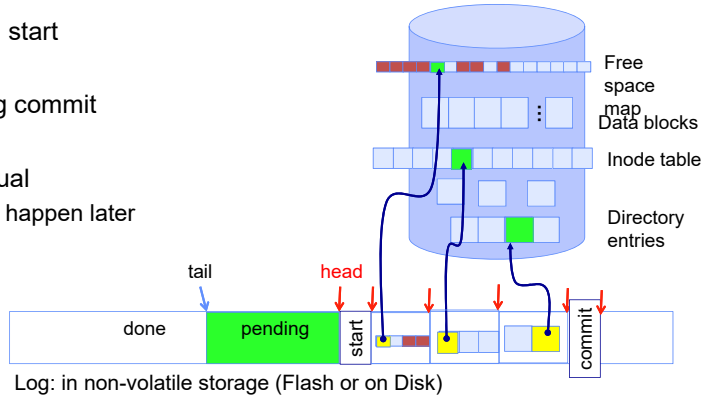
- Upon recovery, scan the log
- Detect transaction start with no commit
- Discard log entries
- Disk remains unchanged



Log: in non-volatile storage (Flash or on Disk)

Crash Recovery: Keep Complete Transactions

- Scan log, find start
- Find matching commit
- Redo it as usual
  - Or just let it happen later



Log: in non-volatile storage (Flash or on Disk)

Journaling Summary

Why go through all this trouble?

- Updates atomic, even if we crash:
  - Update either gets fully applied or discarded
  - All physical operations *treated as a logical unit*

Isn't this expensive?

- Yes! We're now writing all data twice (once to log, once to actual data blocks in target file)
- Modern filesystems journal metadata updates only
  - Record modifications to file system data structures
  - But apply updates to a file's contents directly

Recall: CS 162 Collaboration Policy

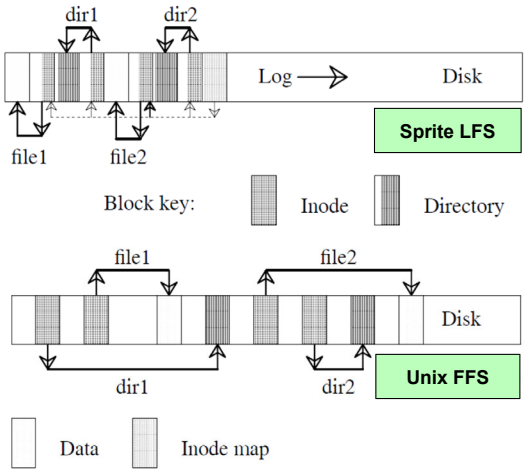
Explaining a concept to someone in another group
Discussing algorithms/testing strategies with other groups
Discussing debugging approaches with other groups
Searching online for generic algorithms (e.g., hash table)

Sharing code or test cases with another group
Copying OR reading another group's code or test cases
Copying OR reading online code or test cases from prior years
Helping someone in another group to debug their code

- We compare all project submissions against prior year submissions and online solutions and will take actions (described on the course overview page) against offenders
- Don't put a friend in a bad position by asking for help that they shouldn't give!

The Log Structured File System (LFS)

- Log Structured File System:
  - The LOG IS the storage
- Log: One continuous sequence of blocks that wrap around whole disk
  - Inodes put into log when changed and point to new data in the log
- Simple example:
  - Create two new files:
    - » dir1/file1 and dir2/file2
    - » Must write new data blocks for files and for new information in directories
  - LFS writes everything sequentially
  - Unix FFS requires 10 non-sequential writes (inodes written twice for ease of recovery)
- Paper on resources page!

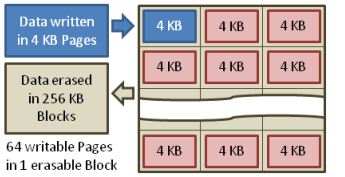


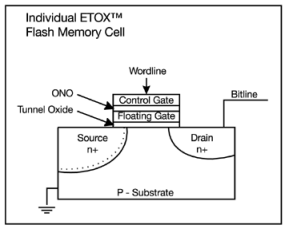
Going Further – Log Structured File Systems

- The log IS what is recorded on disk
  - File system operations logically replay log to get result
  - Create data structures to make this fast
  - On recovery, replay the log
- Index (inodes) and directories are written into the log too
- Large, important portion of the log is cached in memory
  - Relies on Buffer Cache to make reading fast
- Do everything in bulk: log is collection of large segments
- Each segment contains a summary of all the operations within the segment
  - Fast to determine if segment is relevant or not
- Free space is approached as continual cleaning process of segments
  - Detect what is live or not within a segment
  - Copy live portion to new segment being formed (replay)
  - Garbage collection entire segment
  - No bit map

What about Flash Filesystems?

- Cannot overwrite pages!
  - Must move contents to an erased page
  - Small changes ⇒ lots of rewriting of data/wear out!
- Program/Erase (PE) Wear
  - Permanent damage to gate oxide at each flash cell
  - Caused by high program/erase voltages
  - Issues: trapped charges, premature leakage of charge
  - *Need to balance how frequently cells written: "Wear Leveling"*
- Flash Translation Layer (FTL)
  - Translates between Logical Block Addresses (at OS level) and Physical Flash Page Addresses
  - Manages the wear and erasure state of blocks and pages
  - Tracks which blocks are garbage but not erased
- Management Process (Firmware)
  - Keep freelist full, Manage mapping,
  - Track wear state of pages
  - Copy good pages out of mostly empty blocks before erasure



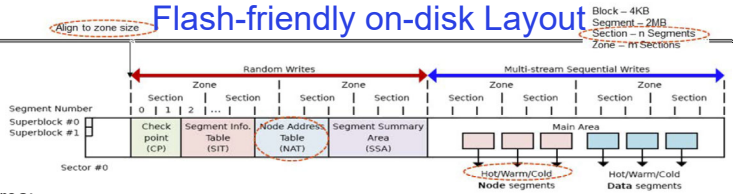


Single FLASH storage bit

Example Use of LFS: F2FS: A Flash File System

- File system used on many mobile devices
  - Including the Pixel 3 from Google
  - Latest version supports block-encryption for security
  - Has been "mainstream" in linux for several years now
- Assumes standard SSD interface
  - With built-in Flash Translation Layer (FTL)
  - Random reads are as fast as sequential reads
  - Random writes are bad for flash storage
    - » Forces FTL to keep moving/coalescing pages and erasing blocks
    - » Sustained write performance degrades/lifetime reduced
- Minimize Writes/updates and otherwise keep writes "sequential"
  - Start with Log-structured file systems/copy-on-write file systems
  - Keep writes as sequential as possible
  - Node Translation Table (NAT) for "logical" to "physical" translation
    - » Independent of FTL
- For more details, check out paper in Readings section of website
  - "F2FS: A New File System for Flash Storage" (from 2015)
  - Design of file system to leverage and optimize NAND flash solutions
  - Comparison with Ext4, Btrfs, Nilfs2, etc

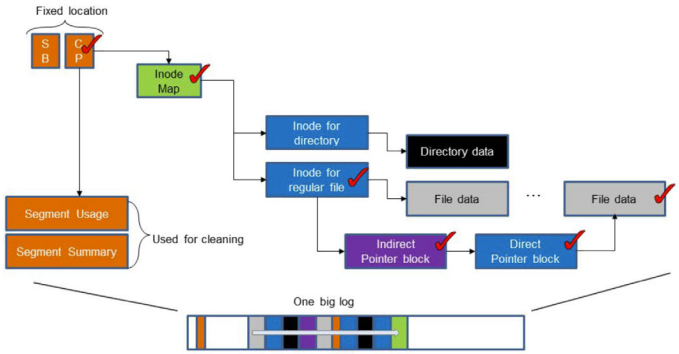
Flash-friendly on-disk Layout



- Main Area:
  - Divided into segments (basic unit of management in F2FS)
  - 4KB Blocks. Each block typed to be *node* or *data*.
- Node Address Table (NAT): *Independent of FTL!*
  - Block address table to locate all "node blocks" in Main Area
- Updates to data sorted by predicted write frequency (Hot/Warm/Cold) to optimize FLASH management
- Checkpoint (CP): Keeps the file system status
  - Bitmaps for valid NAT/SIT sets and Lists of orphan inodes
  - Stores a consistent F2FS status at a given point in time
- Segment Information Table (SIT):
  - Per segment information such as number of valid blocks and the bitmap for the validity of all blocks in the "Main" area
  - Segments used for "garbage collection"
- Segment Summary Area (SSA):
  - Summary representing the owner information of all blocks in the Main area

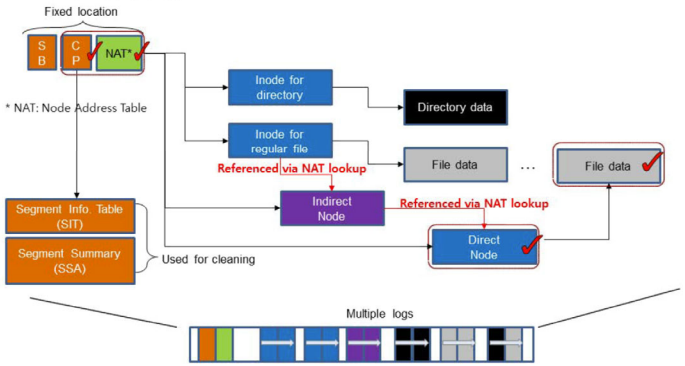
Normal LFS Index Structure: Forces cascading updates when updating data

- **Update propagation issue: wandering tree**
- **One big log**



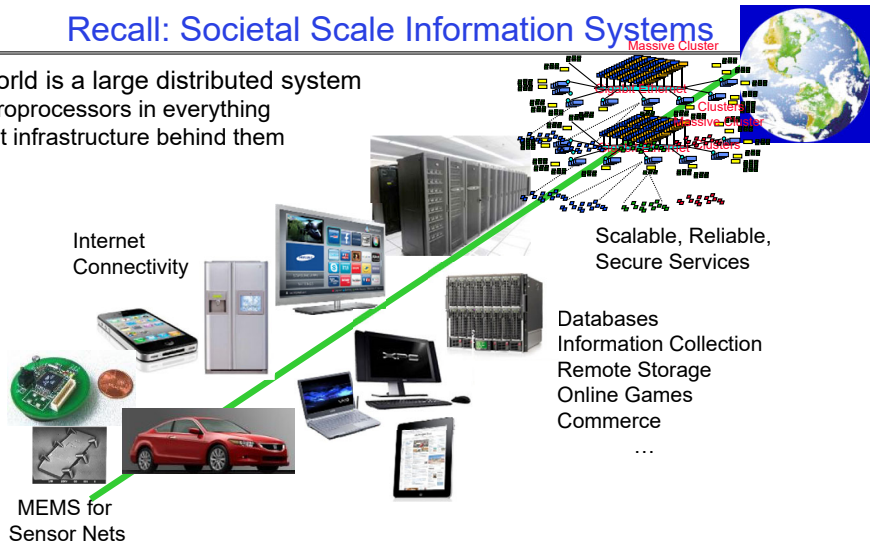
F2FS Index Structure: Indirection and Multi-head logs optimize updates

- **Restrained update propagation: node address translation method**
- **Multi-head log**

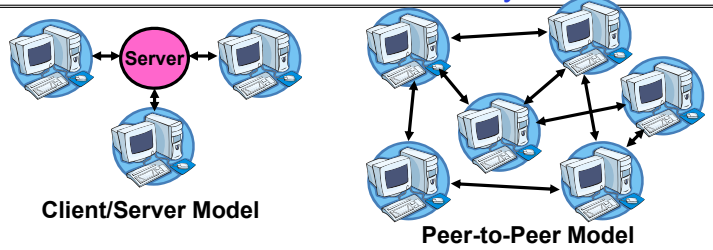


Recall: Societal Scale Information Systems

- The world is a large distributed system
  - Microprocessors in everything
  - Vast infrastructure behind them



Centralized vs Distributed Systems



Client/Server Model

Peer-to-Peer Model

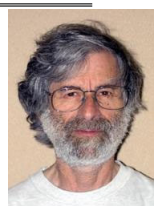
- Centralized System: major functions performed by a single physical computer
  - Originally, everything on single computer
  - Later: client/server model
- Distributed System: physically separate computers working together on task
  - Early model: multiple servers working together
    - » Probably in 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

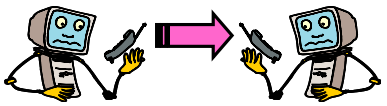
- Reality has been disappointing
  - *Worse availability*: depend on every machine being up
    - » Lamport: "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."
  - *Worse reliability*: can lose data if any machine crashes
  - *Worse security*: anyone in world can break into system
- Coordination is more difficult
  - Must coordinate multiple copies of shared state information
  - 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!"



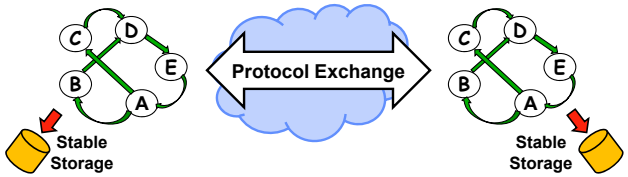
Leslie Lamport

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



How do entities communicate? A Protocol!

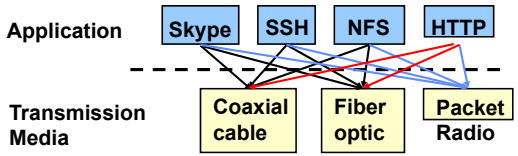


- A protocol is an agreement on how to communicate, including:
  - Syntax: how a communication is specified & structured
    - » Format, order messages are sent and received
  - Semantics: what a communication means
    - » Actions taken when transmitting, receiving, or when a timer expires
- Described 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

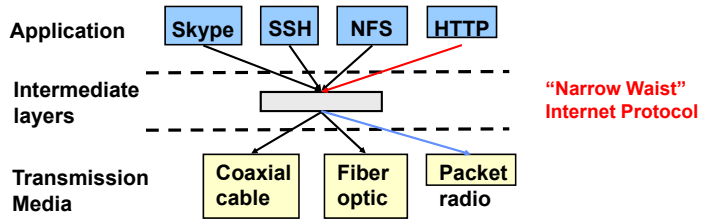
- Telephone
  1. (Pick up / open up the phone)
  2. Listen for a dial tone / see that you have service
  3. Dial
  4. Should hear ringing …
  5. Callee: "Hello?"
  6. Caller: "Hi, it's John…."
     Or: "Hi, it's me" (← what's *that* about?)
  7. Caller: "Hey, do you think … blah blah blah …" **pause**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

Global Communication: The Problem



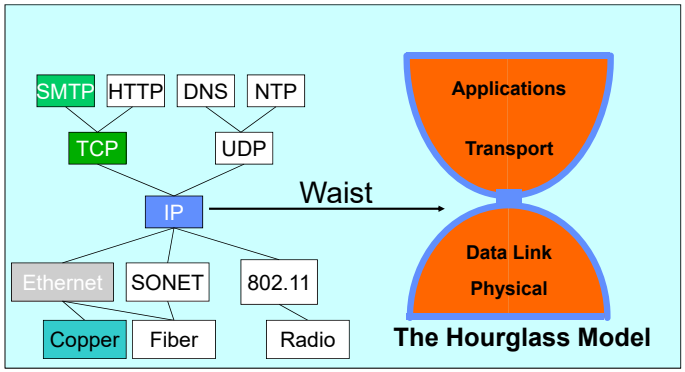
- Many different applications
  - email, web, P2P, etc.
- Many different network styles and technologies
  - Wireless vs. wired vs. optical, etc.
- How do we organize this mess?
  - Re-implement every application for every technology?
- No! But how does the Internet design avoid this?

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

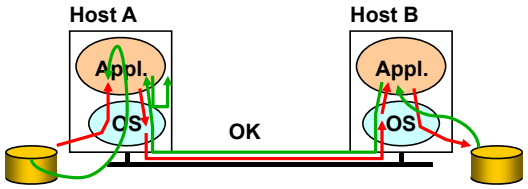
Drawbacks of Layering

- 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

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)
    - » 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:
    int msg1[1000];
    while(1) {
        prepare message;
        send(msg1,mbox);
    }
```

Send Message

```
Consumer:
    int buffer[1000];
    while(1) {
        receive(buffer,mbox);
        process message;
    }
```

Receive 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

```
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);
```

Request File

Get Response

Receive Request

Send Response

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!

Summary (1/2)

- Important system properties
  - Availability: how often is the resource available?
  - Durability: how well is data preserved against faults?
  - Reliability: how often is resource performing correctly?
- RAID: Redundant Arrays of Inexpensive Disks
  - RAID1: mirroring, RAID5: Parity block
- Copy-on-write provides richer function (versions) with much simpler recovery
  - Little performance impact since sequential write to storage device is nearly free
- Use of Log to improve Reliability
  - Journaled file systems such as ext3, NTFS
- Transactions over a log provide a general solution
  - Commit sequence to durable log, then update the disk
  - Log takes precedence over disk
  - Replay committed transactions, discard partials

Summary (2/2)

- 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
- Next Time: Two-phase commit: distributed decision making
  - First, make sure everyone guarantees that they will commit if asked (prepare)
  - Next, ask everyone to commit