

CS162

Operating Systems and Systems Programming

Lecture 20

Reliability, Transactions Distributed Systems

April 9th, 2018

Profs. Anthony D. Joseph & Jonathan Ragan-Kelley

<http://cs162.eecs.Berkeley.edu>

Important “ilities”

- **Availability:** the probability that the system can accept and process requests
 - Often measured in “nines” of probability. So, a 99.9% probability is considered “3-nines of availability”
 - Key idea here is independence of failures

Availability %	Downtime per year	Downtime per month	Downtime per week
90% (“one nine”)			
99% (“two nines”)			
99.9% (“three nines”)			
99.99% (“four nines”)			
99.999% (“five nines”)			
99.9999% (“six nines”)			

*Google, Amazon, Microsoft cloud target Service Level Agreements (SLAs)

4/9/18

CS162 © UCB Spring 2018

Lec 20.2

Important “ilities”

- **Availability:** the probability that the system can accept and process requests
 - Often measured in “nines” of probability. So, a 99.9% probability is considered “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, etc

4/9/18

CS162 © UCB Spring 2018

Lec 20.3

How to Make File System Durable?

- Disk blocks contain Reed-Solomon error correcting codes (ECC) to deal with small defects in disk drive
 - Can allow recovery of data from small media defects
- Make sure writes survive in short term
 - Either abandon delayed writes or
 - Use special, battery-backed RAM (called non-volatile RAM or **NVRAM**) for dirty blocks in buffer cache
- Make sure that data survives in long term
 - Need to replicate! More than one copy of data!
 - Important element: **independence of failure**
 - » Could put copies on one disk, but if disk head fails...
 - » Could put copies on different disks, but if server fails...
 - » Could put copies on different servers, but if building is struck by lightning....
 - » Could put copies on servers in different continents...

4/9/18

CS162 © UCB Spring 2018

Lec 20.4

RAID: Redundant Arrays of Inexpensive Disks

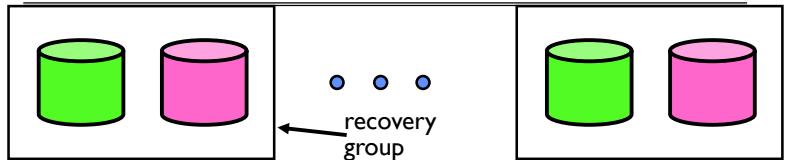
- Invented by David Patterson, Garth A. Gibson, and Randy Katz here at UCB in 1987
- Data stored on multiple disks (redundancy)
- Either in software or hardware
 - In hardware case, done by disk controller; file system may not even know that there is more than one disk in use
- Initially, five levels of RAID (more now)

4/9/18

CSI62 © UCB Spring 2018

Lec 20.5

RAID 1: Disk Mirroring/Shadowing



- Each disk is fully duplicated onto its “shadow”
 - For high I/O rate, high availability environments
 - Most expensive solution: 100% capacity overhead
- Bandwidth sacrificed on write:
 - Logical write = two physical writes
 - Highest bandwidth when disk heads and rotation fully synchronized (hard to do exactly)
- Reads may be optimized
 - Can have two independent reads to same data
- Recovery:
 - Disk failure \Rightarrow replace disk and copy data to new disk
 - **Hot Spare:** idle disk already attached to system to be used for immediate replacement

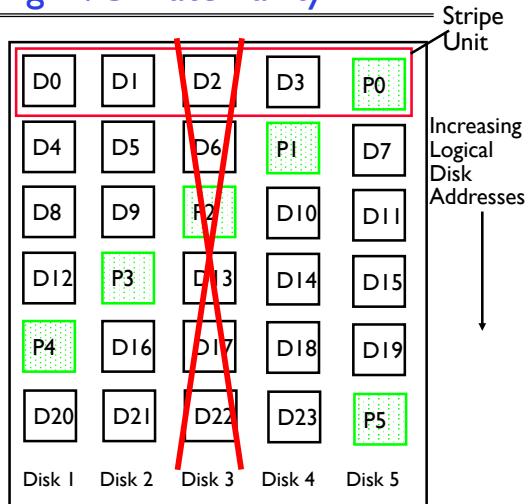
4/9/18

CSI62 © UCB Spring 2018

Lec 20.6

RAID 5+: High I/O Rate Parity

- Data striped across multiple disks
 - Successive blocks stored on successive (non-parity) disks
 - Increased bandwidth over single disk
- Parity block (in green) constructed by XORing data blocks in stripe
 - $P_0 = D_0 \oplus D_1 \oplus D_2 \oplus D_3$
 - Can destroy any one disk and still reconstruct data
 - Suppose Disk 3 fails, then can reconstruct: $D_2 = D_0 \oplus D_1 \oplus D_3 \oplus P_0$
- Can spread information widely across internet for durability
 - Overview now, more later in semester



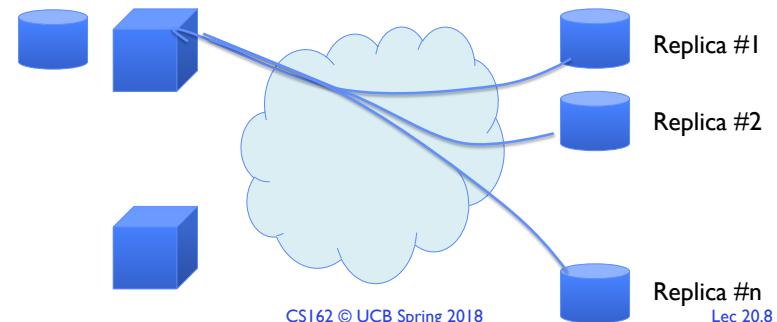
4/9/18

CSI62 © UCB Spring 2018

Lec 20.7

Higher Durability/Reliability through Geographic Replication

- Highly durable – hard to destroy all copies
- Highly available for reads – read any copy
- Low availability for writes
 - Can't write if any one replica is not up
 - Or – need relaxed consistency model
- Reliability? – availability, security, durability, fault-tolerance



4/9/18

CSI62 © UCB Spring 2018

Lec 20.8

File System Reliability

- What can happen if disk loses power or software crashes?
 - Some operations in progress may complete
 - Some operations in progress may be lost
 - Overwrite of a block may only partially complete
- Having RAID doesn't necessarily protect against all such failures
 - No protection against writing bad state
 - What if one disk of RAID group not written?
- File system needs durability (as a minimum!)
 - Data previously stored can be retrieved (maybe after some recovery step), regardless of failure

4/9/18

CS162 © UCB Spring 2018

Lec 20.9

Threats to Reliability

- Interrupted Operation
 - Crash or power failure in the middle of a series of related updates may leave stored data in an *inconsistent state*
 - Example: Transfer funds from one bank account to another
 - What if transfer is interrupted after withdrawal and before deposit?
- Loss of stored data
 - Failure of non-volatile storage media may cause previously stored data to disappear or be corrupted

4/9/18

CS162 © UCB Spring 2018

Lec 20.11

Storage Reliability Problem

- Single logical file operation can involve updates to multiple physical disk blocks
 - inode, indirect block, data block, bitmap, ...
 - With sector remapping, single update to physical disk block can require multiple (even lower level) updates to sectors
- At a physical level, operations complete one at a time
 - Want concurrent operations for performance
- How do we guarantee consistency regardless of when crash occurs?

4/9/18

CS162 © UCB Spring 2018

Lec 20.10

Reliability Approach #1: Careful Ordering

- Sequence operations in a specific order
 - Careful design to allow sequence to be interrupted safely
- Post-crash recovery
 - Read data structures to see if there were any operations in progress
 - Clean up/finish as needed
- Approach taken by
 - FAT and FFS (**fsck**) to protect filesystem structure/metadata
 - Many app-level recovery schemes (e.g., Word, emacs autosaves)

4/9/18

CS162 © UCB Spring 2018

Lec 20.12

FFS: Create a File

Normal operation:

- Allocate data block
- Write data block
- Allocate inode
- Write inode block
- Update bitmap of free blocks and inodes
- Update directory with file name → inode number
- Update modify time for directory

Recovery:

- Scan inode table
- If any unlinked files (not in any directory), delete or put in lost & found dir
- Compare free block bitmap against inode trees
- Scan directories for missing update/access times

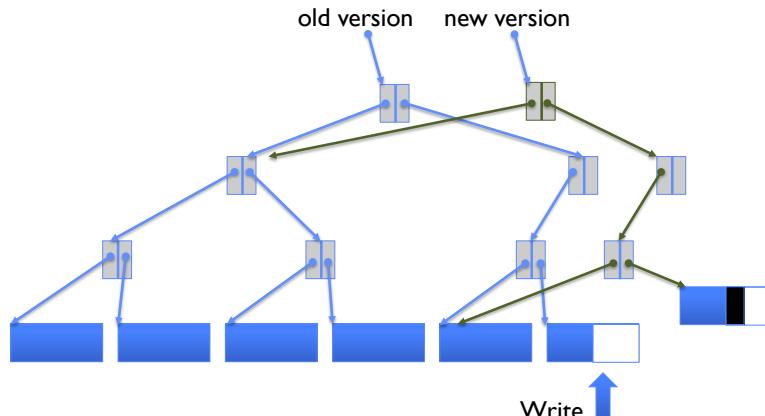
Time proportional to disk size

4/9/18

CS162 © UCB Spring 2018

Lec 20.13

COW with Smaller-Radix Blocks



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

4/9/18

CS162 © UCB Spring 2018

Lec 20.15

Reliability Approach #2: Copy on Write File Layout

- To update file system, write a new version of the file system containing the update
 - Never update in place
 - Reuse existing unchanged disk blocks
- Seems expensive! But
 - Updates can be batched
 - Almost all disk writes can occur in parallel
- Approach taken in network file server appliances
 - NetApp's Write Anywhere File Layout (WAFL)
 - ZFS (Sun/Oracle) and OpenZFS

4/9/18

CS162 © UCB Spring 2018

Lec 20.14

ZFS and OpenZFS

- Variable sized blocks: 512 B – 128 KB
- Symmetric tree
 - Know if it is large or small when we make the copy
- Store version number with pointers
 - Can create new version by adding blocks and new pointers
- Buffers a collection of writes before creating a new version with them
- Free space represented as tree of extents in each block group
 - Delay updates to freespace (in log) and do them all when block group is activated

4/9/18

CS162 © UCB Spring 2018

Lec 20.16

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)

4/9/18

CS162 © UCB Spring 2018

Lec 20.17

Key Concept: Transaction

- An *atomic sequence* of actions (reads/writes) on a storage system (or database)
- That takes it from one *consistent state* to another



4/9/18

CS162 © UCB Spring 2018

Lec 20.19

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

4/9/18

CS162 © UCB Spring 2018

Lec 20.18

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

4/9/18

CS162 © UCB Spring 2018

Lec 20.20

“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

4/9/18

CS162 © UCB Spring 2018

Lec 20.21

Administrivia

- Project 3: initial design docs due this Wednesday, April 11th

4/9/18

CS162 © UCB Spring 2018

Lec 20.23

The ACID properties of Transactions

- **Atomicity:** all actions in the transaction happen, or none happen
- **Consistency:** transactions maintain data integrity, e.g.,
 - Balance cannot be negative
 - Cannot reschedule meeting on February 30
- **Isolation:** execution of one transaction is isolated from that of all others; no problems from concurrency
- **Durability:** if a transaction commits, its effects persist despite crashes

4/9/18

CS162 © UCB Spring 2018

Lec 20.22

Break

4/9/18

CS162 © UCB Spring 2018

Lec 20.24

Transactional File Systems (1/2)

- 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 (improve perf. w/ NVRAM)
 - File system may not be updated immediately, data preserved in the log
- Difference between “Log Structured” and “Journaling”
 - In a Log Structured filesystem, data & metadata stays in log form
 - In a Journaling filesystem, Log used for recovery

4/9/18

CS162 © UCB Spring 2018

Lec 20.25

Transactional File Systems (2/2)

- 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

4/9/18

CS162 © UCB Spring 2018

Lec 20.26

Logging File Systems (1/2)

- Full Logging File System
 - All updates to disk are done in transactions
- Instead of modifying data structures on disk directly, write changes to a journal/log
 - Intention list: set of changes we intend to make
 - Log/Journal is **append-only**
 - Single commit record commits transaction
- Once changes are in log, it is safe to apply changes to data structures on disk
 - Recovery can read log to see what changes were intended
 - Can take time making the changes
 - » As long as new requests consult the log first

4/9/18

CS162 © UCB Spring 2018

Lec 20.27

Logging File Systems (2/2)

- Once changes are copied, safe to remove log
- But, ...
 - If the last atomic action is not done ... poof ... all gone
- Basic assumption:
 - Updates to sectors are atomic and ordered
 - Not necessarily true unless very careful, but key assumption
- Performance
 - Great for random writes: replace with appends to log
 - Impact read performance, but can alleviate this by caching

4/9/18

CS162 © UCB Spring 2018

Lec 20.28

Redo Logging

- Prepare
 - Write all changes (in transaction) to log
- Commit
 - Single disk write to make transaction durable
- Redo
 - Copy changes to disk (idempotent)
- Garbage collection
 - Reclaim space in log
- Recovery
 - Read log
 - Redo any operations for committed transactions
 - Garbage collect log

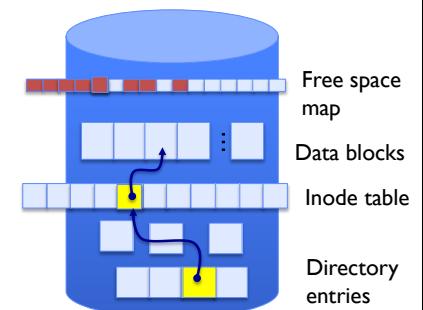
4/9/18

CS162 © UCB Spring 2018

Lec 20.29

Example: Creating a File

- 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



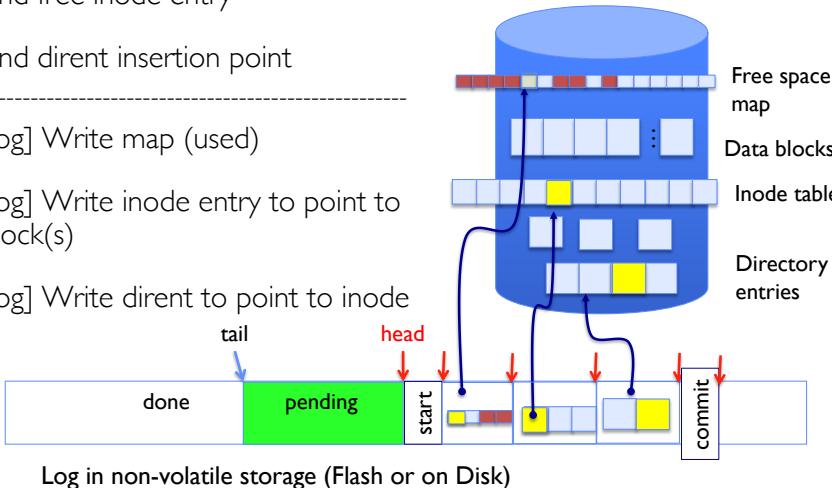
4/9/18

CS162 © UCB Spring 2018

Lec 20.30

Ex: Creating a file (as a transaction)

- Find free data block(s)
- Find free inode entry
- Find dirent insertion point
- [log] Write map (used)
- [log] Write inode entry to point to block(s)
- [log] Write dirent to point to inode



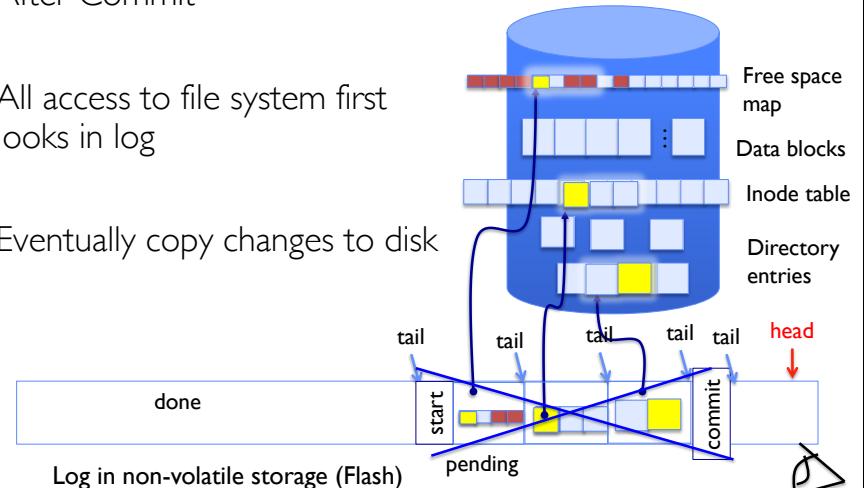
4/9/18

CS162 © UCB Spring 2018

Lec 20.31

ReDo Log

- After Commit
- All access to file system first looks in log
- Eventually copy changes to disk



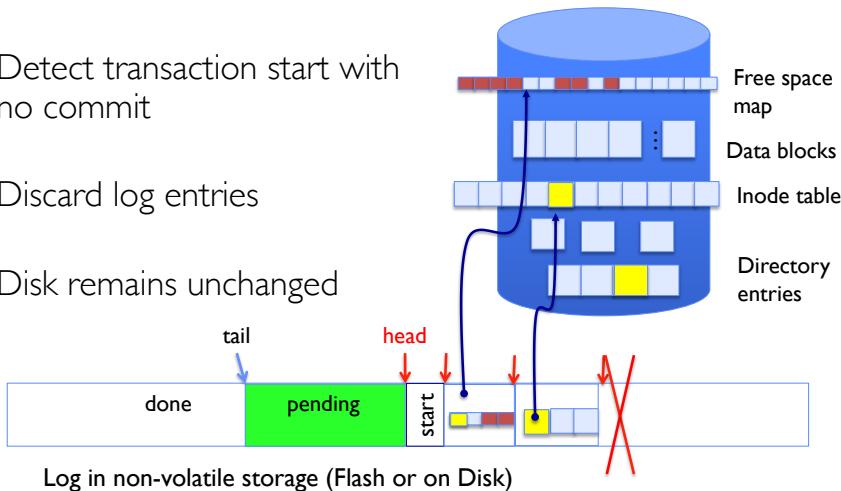
4/9/18

CS162 © UCB Spring 2018

Lec 20.32

Crash During Logging – Recover

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



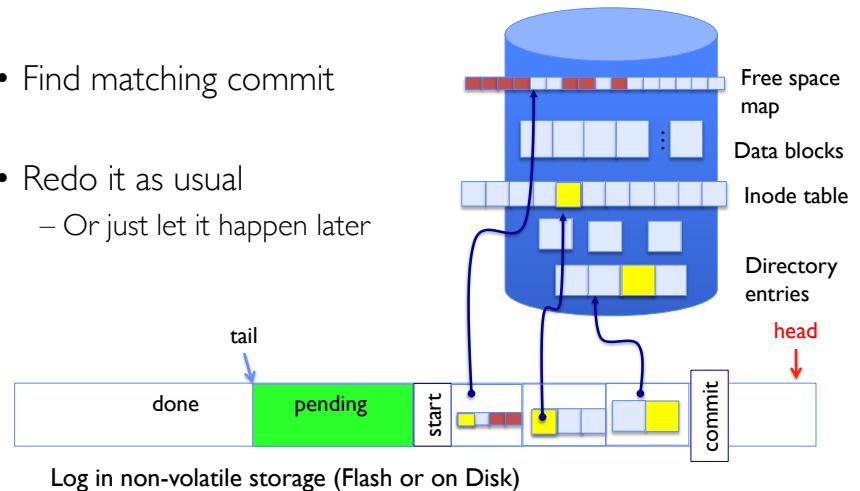
4/9/18

CS162 © UCB Spring 2018

Lec 20.33

Recovery After Commit

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

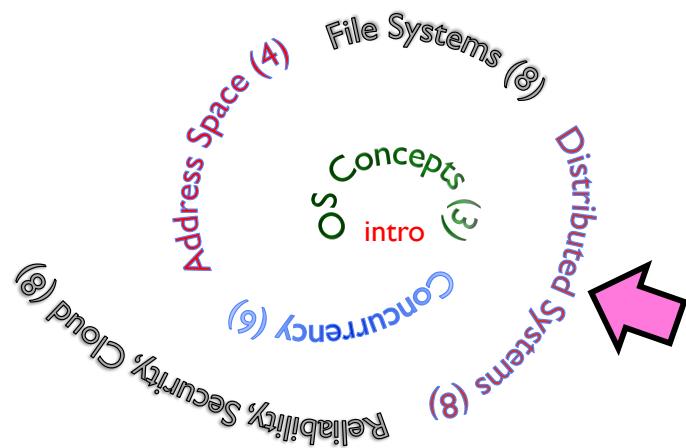


4/9/18

CS162 © UCB Spring 2018

Lec 20.34

Course Structure: Spiral



4/9/18

CS162 © UCB Spring 2018

Lec 20.35

Societal Scale Information Systems

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

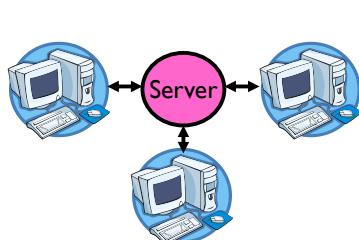


4/9/18

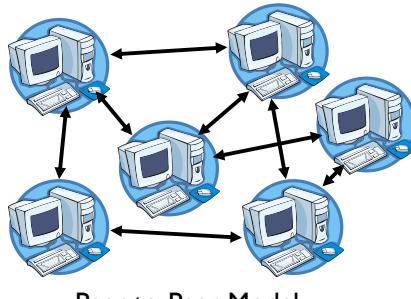
CS162 © UCB Spring 2018

Lec 20.36

Centralized vs Distributed Systems



Client/Server Model



Peer-to-Peer Model

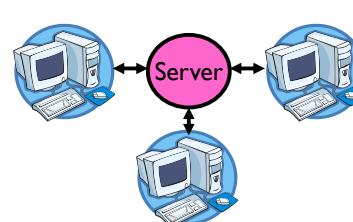
- **Centralized System:** System in which major functions are performed by a single physical computer
 - Originally, everything on single computer
 - Later: client/server model

4/9/18

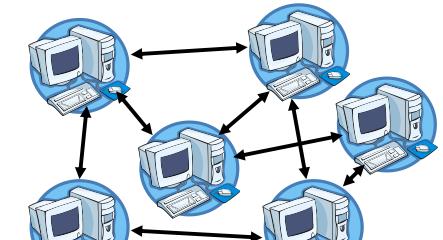
CS162 © UCB Spring 2018

Lec 20.37

Centralized vs Distributed Systems



Client/Server Model



Peer-to-Peer 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

4/9/18

CS162 © UCB Spring 2018

Lec 20.38

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

4/9/18

CS162 © UCB Spring 2018

Lec 20.39

Distributed Systems: Reality

- Reality has been disappointing
 - Worse availability: depend on every machine being up
 - » Lamport: “*a distributed system is one where I can't do work because some machine I've never heard of isn't working!*”
 - 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

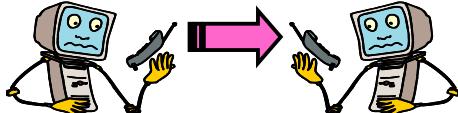
4/9/18

CS162 © UCB Spring 2018

Lec 20.40

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



4/9/18

CS162 © UCB Spring 2018

Lec 20.41

Summary

- **RAID:** Redundant Arrays of Inexpensive Disks
 - RAID1: mirroring, RAID5: Parity block
- Use of Log to improve Reliability
 - Journaling file systems such as ext3, NTFS
- **Transactions:** ACID semantics
 - Atomicity
 - Consistency
 - Isolation
 - Durability

4/9/18

CS162 © UCB Spring 2018

Lec 20.42