CS 162: Operating Systems and Systems Programming

Lecture 20: Distributed File Systems [and Key-Value Store]*

November 7, 2019
Instructor: David E. Culler
https://cs162.eecs.berkeley.edu

Read: 3 Easy ch 49 (NFS), 50(AFS)

* Time permitting

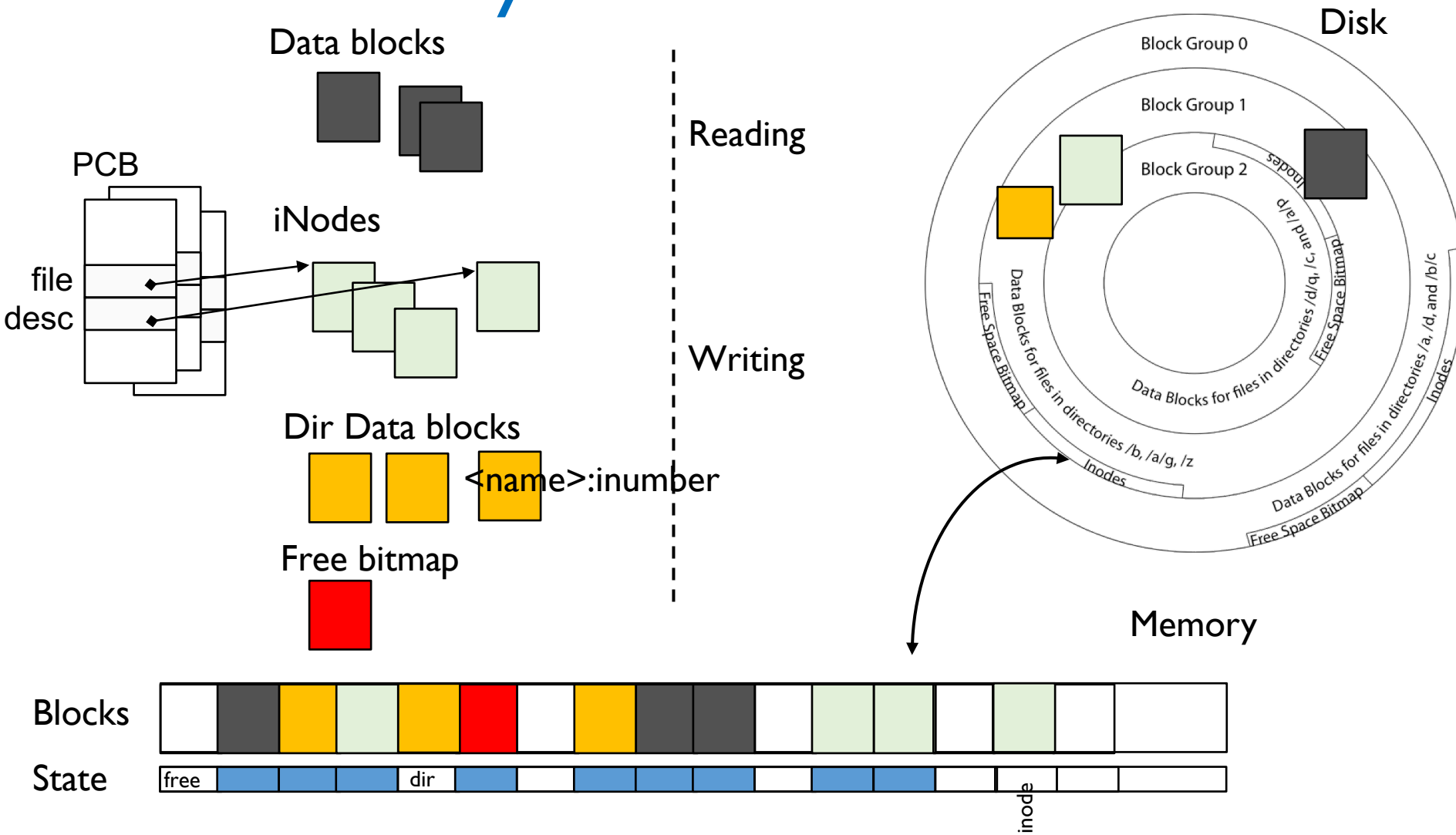
File Systems so far (1/3)

- File System:
  - Transforms blocks into Files and Directories
  - Optimize for size, access and usage patterns
  - Maximize sequential access, allow efficient random access
  - Projects the OS protection and security regime (UGO vs ACL)
- File defined by header, called "inode"
- Naming: translating from user-visible names to actual sys resources
  - Directories used for naming for local file systems
  - Linked or tree structure stored in files
- Multilevel Indexed Scheme
  - inode contains file info, direct pointers to blocks, indirect blocks, doubly indirect, etc..
  - NTFS: variable extents not fixed blocks, tiny files data is in header

File Systems so far (2/3)

- File layout driven by freespace management
  - Optimizations for sequential access: start new files in open ranges of free blocks, rotational optimization
  - Integrate freespace, inode table, file blocks and dirs into block group
- Deep interactions between mem management, file system, sharing
  - `mmap()`: map file or anonymous segment to memory
- Buffer Cache: Memory used to cache kernel resources, including disk blocks and name translations
  - Can contain "dirty" blocks (blocks yet on disk)

Recall: File System Buffer Cache



- Blocks being written back to disc go through a transient state

Buffer Cache – Issues and Opportunities

- Amount of memory devoted to it (adaptive)
- Replacement Policy
  - LRU vs Streaming
- Read-Ahead Prefetching
- Delayed Writes
- Disk / SSD optimizations
  - Sequential transfers, Multiple concurrent requests
- Recovery

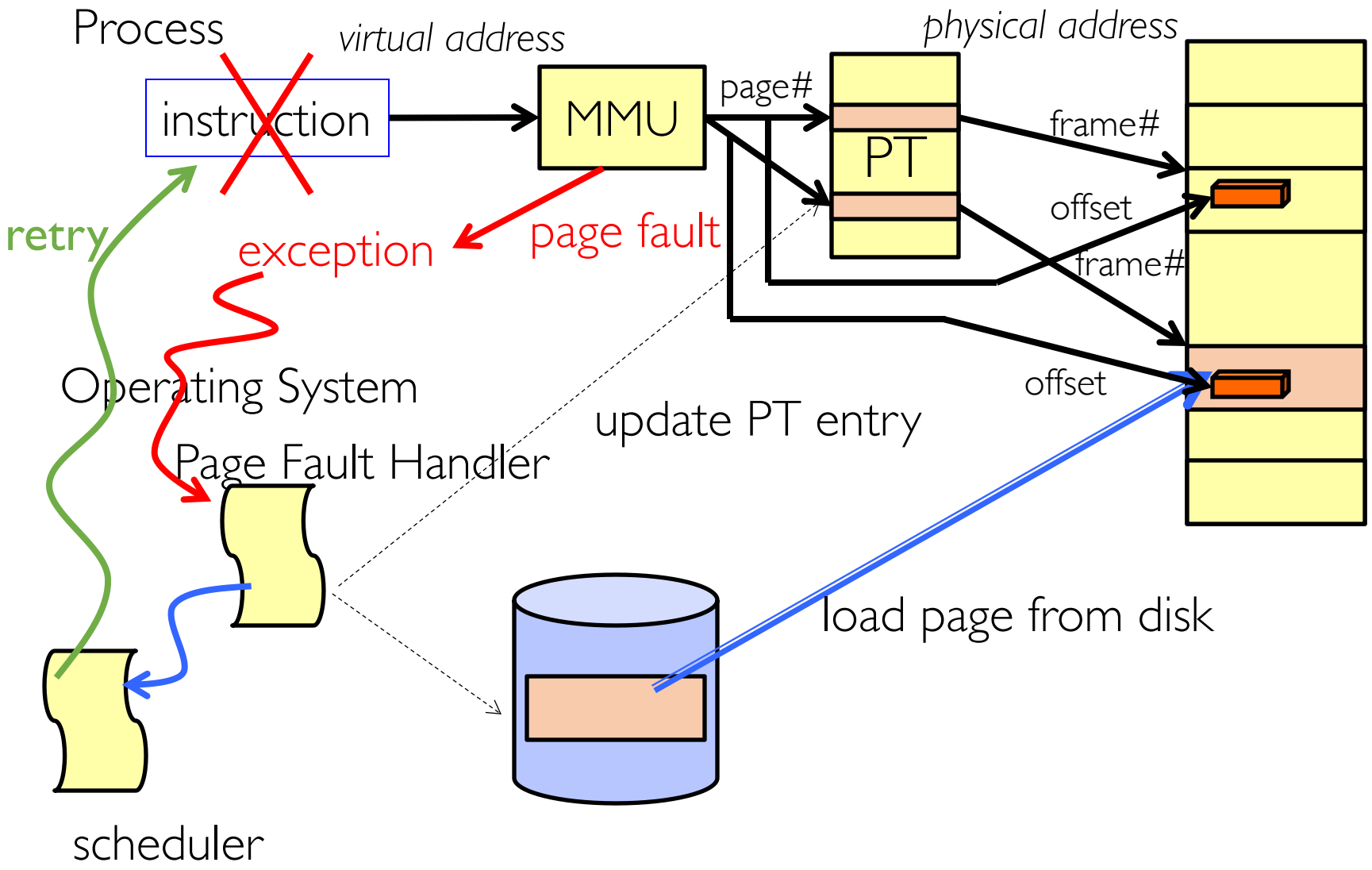
Memory Mapped Files

- Traditional I/O involves explicit transfers between buffers in process address space to/from regions of a file
  - This involves multiple copies into caches in memory, plus system calls

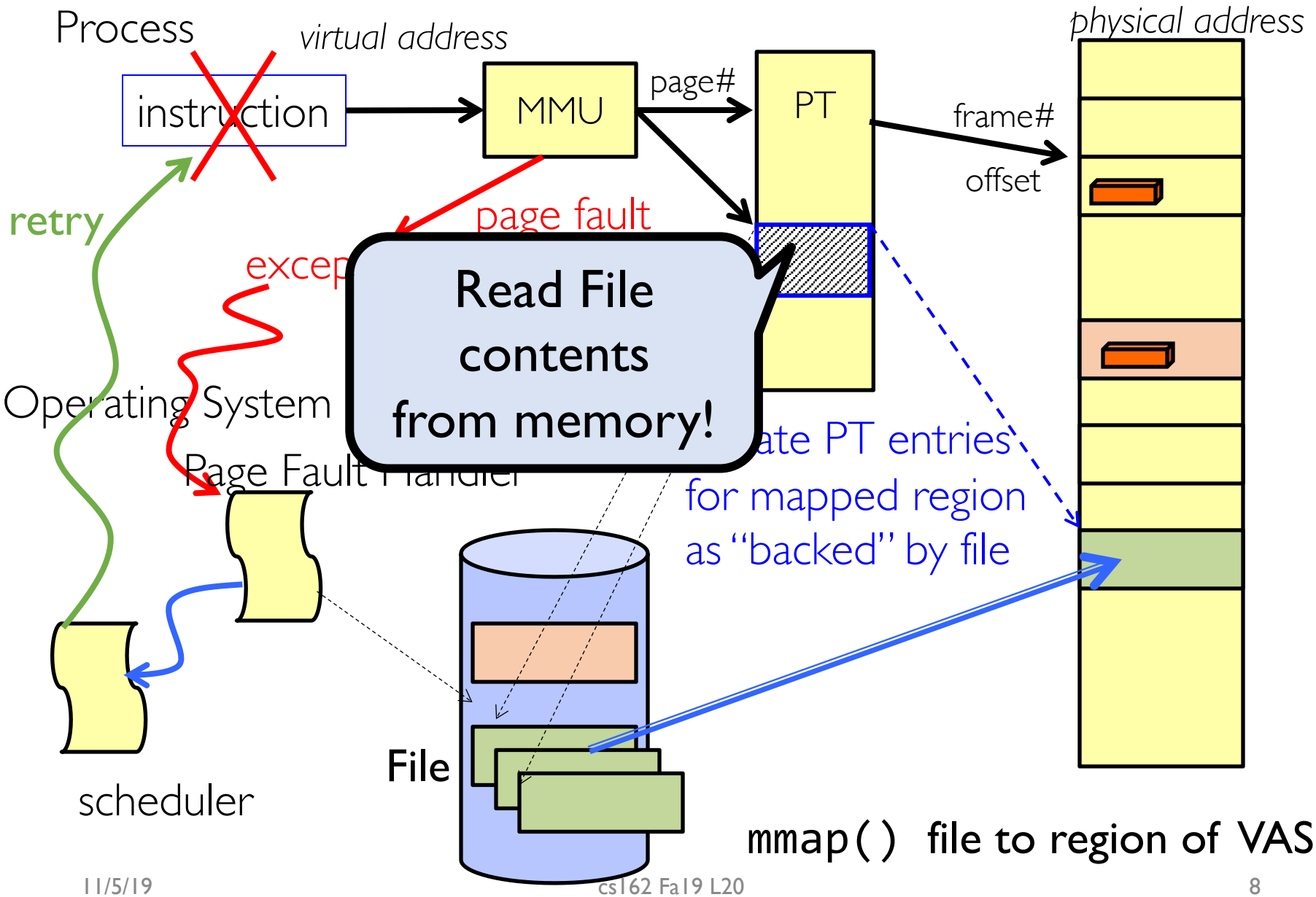
- What if we could "map" the file directly into an empty region of our address space
  - Implicitly "page it in" when we read it
  - Write it and "eventually" page it out

- Executable files are treated this way when we `exec` the process!!

Recall: Who Does What, When?



Using Paging to `mmap()` Files



```
MMAP(2)                    BSD System Calls Manual                    MMAP(2)

NAME
     mmap -- allocate memory, or map files or devices into memory

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <sys/mman.h>

     void *
     mmap(void *addr, size_t len, int prot, int flags, int fd,
         off_t offset);

DESCRIPTION
     The mmap() system call causes the pages starting at addr and continuing
     for at most len bytes to be mapped from the object described by fd,
     starting at byte offset offset.  If offset or len is not a multiple of
```

- May map a specific region or let the system find one for you
  - Tricky to know where the holes are
- Used both for manipulating files and for sharing between processes

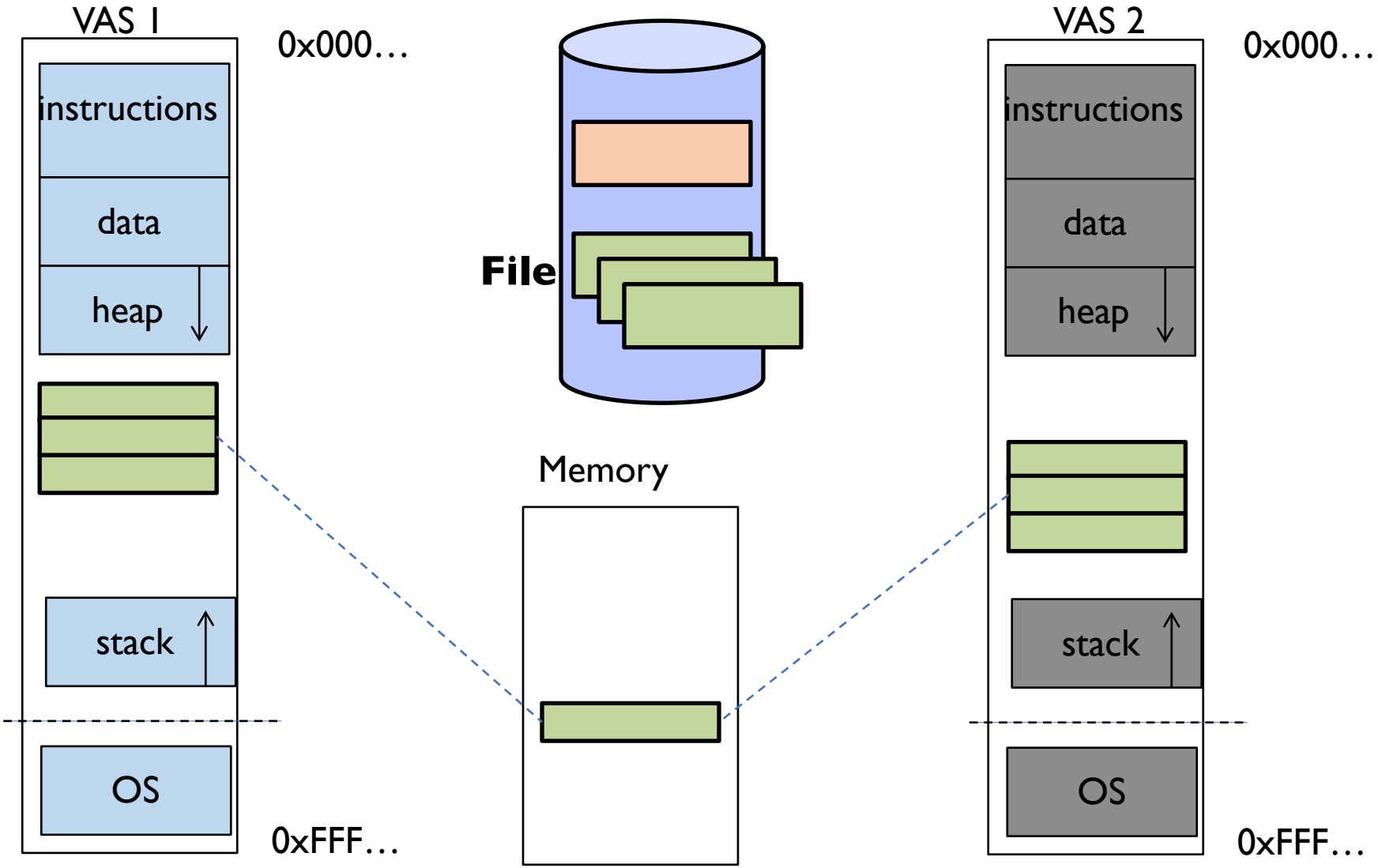
An mmap( ) Example

```
#include <sys/mman.h> /* also stdio.h, stdlib.h, string.h, fcntl.h, unistd.h */

int something = 162;

int main (int argc, char *argv[]) {
  int myfd;
  char *mfile;

  printf("Data  at: %16lx\n", (long
  printf("Heap at : %16lx\n", (long
  printf("Stack at: %16lx\n", (long

  /* Open the file */
  myfd = open(argv[1], O_RDWR | O_CR
  if (myfd < 0) { perror("open failed!

  /* map the file */
  mfile = mmap(0, 10000, PROT_READ|P
  if (mfile == MAP_FAILED) {perror(

  printf("mmap at : %16lx\n", (long

  puts(mfile);
  strcpy(mfile+20,"Let's write over
  close(myfd);
  return 0;
}
```

```
$ ./mmap test
Data  at:         105d63058
Heap at :      7f8a33c04b70
Stack at:      7fff59e9db10
mmap at :         105d97000
This is line one
This is line two
This is line three
This is line four
```

```
$ cat test
This is line one
ThiLet's write over its line three
This is line four
```

Sharing through Mapped Files



- Also: anonymous memory between parents and children
  - no file backing – just swap space

File Systems So Far (3/3)

- File system operations involve multiple distinct updates to blocks on disk
  - Need to have all or nothing semantics
  - Crash may occur in the midst of the sequence
- Traditional file system perform check and recovery on boot
  - Along with careful ordering so partial operations result in loose fragments, rather than loss
- Copy-on-write provides richer function (versions) with much simpler recovery
  - Little performance impact since sequential write to storage device is nearly free
- 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

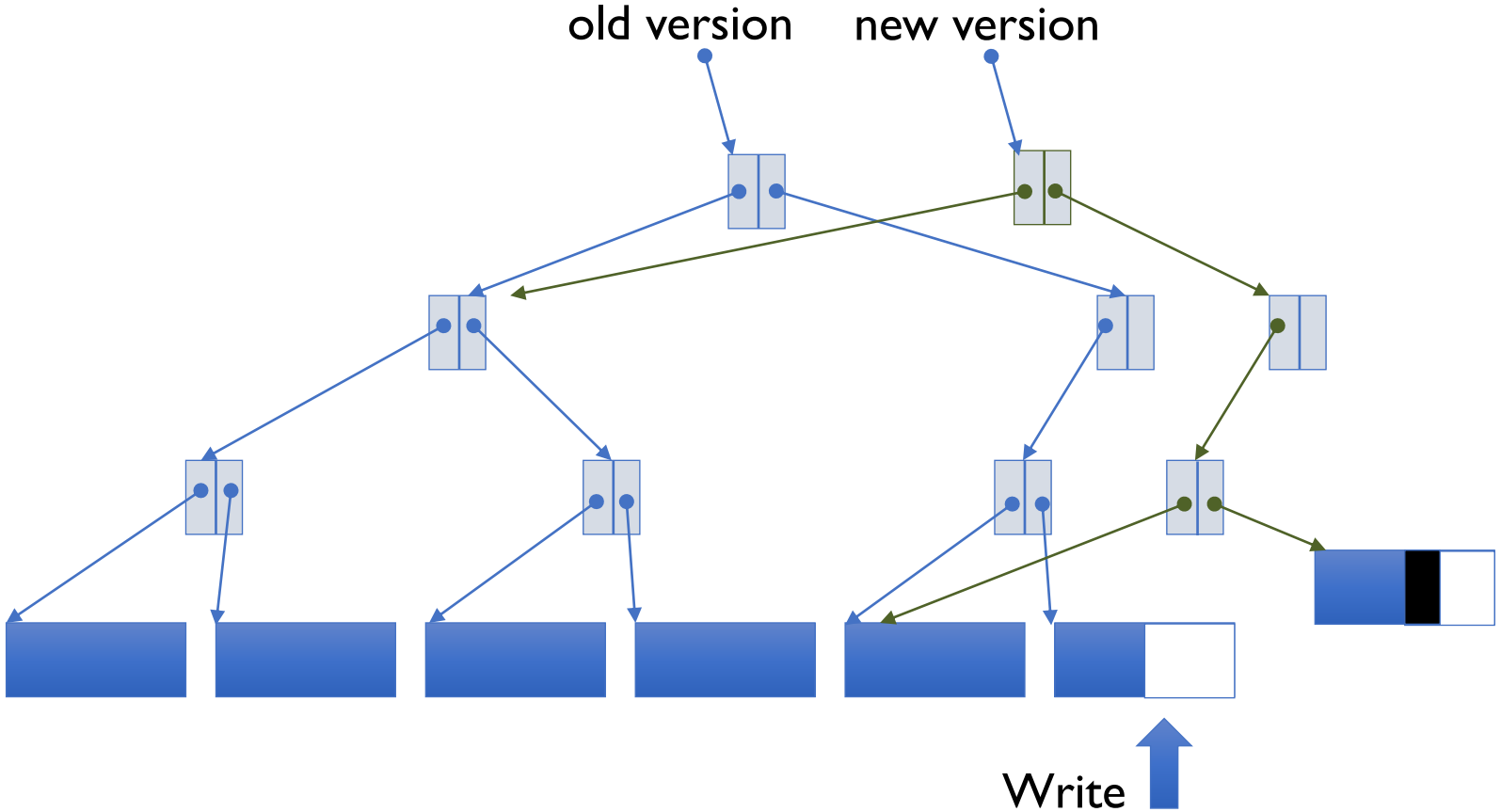
Recall: Threats to Durability

- 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
- Interrupted Operation
  - Crash or power failure in the middle of a series of related updates may leave stored data in an inconsistent state
- Loss of stored data
  - Failure of non-volatile storage media may cause previously stored data to disappear or be corrupted

Recall: 2 Approaches

- Careful Ordering & Recovery
  - FAT & FFS + (fsck)
  - Each step builds structure,
    - Data block << inode << free << directory
  - last links it in
  - Recover scans structure looking for incomplete actions
    - Scan inodes for unlinked files
    - Block bitmap for inodes
    - Directories for missing updates
- Versioning
  - ZFS, OpenZFS, WAFL
  - At some granularity…
  - Create new structure linking back to unchanged parts of old
  - Last step is to declare new version exists

COW with Smaller-Radix Blocks



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

Recall: 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

"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

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

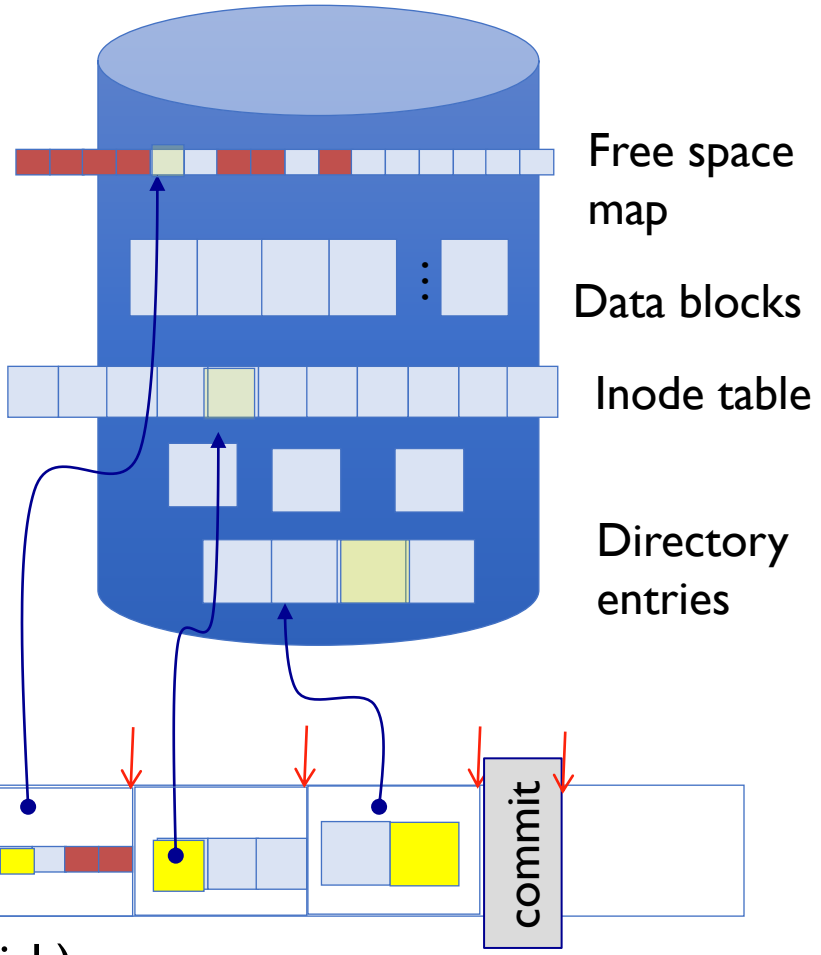
Transactional File Systems

- 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

Recall: Creating a file (as 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



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

Idempotent operations – repeat the write to the same place, no reads

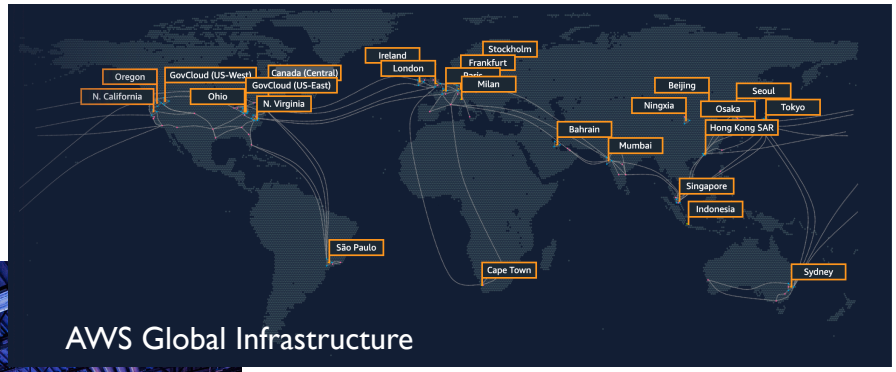
Journaled File Systems

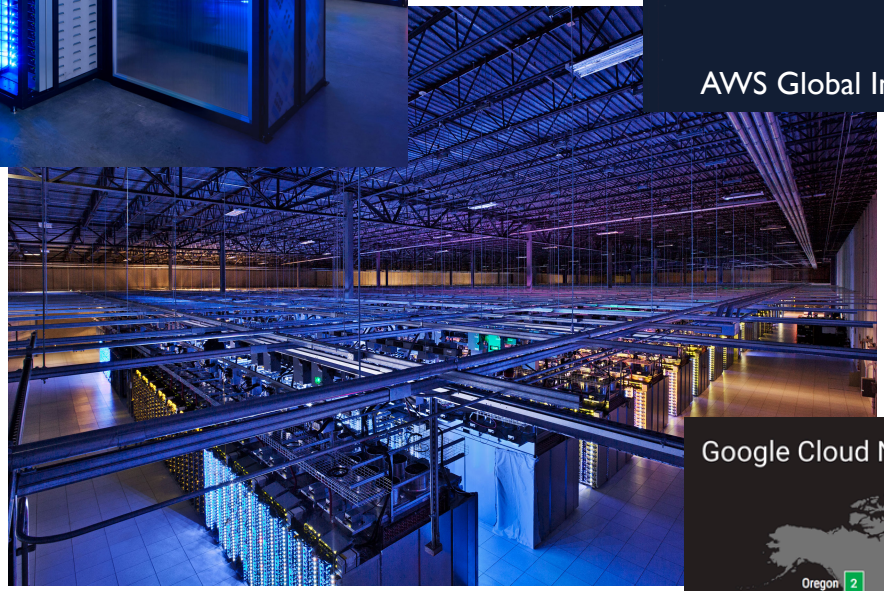
- Instead of modifying data structures on disk directly, write changes to a journal/log
  - Intention list: set of changes we intend to make
  - Log/Journal is append-only
  - Single commit record commits transaction
- Once changes are in the log, it is safe to apply changes to data structures on disk
  - Recovery can read log to see what changes were intended
  - Can take our time making the changes
    - As long as new requests consult the log first
- 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
  - Requires very careful engineering – action made as simple as possible

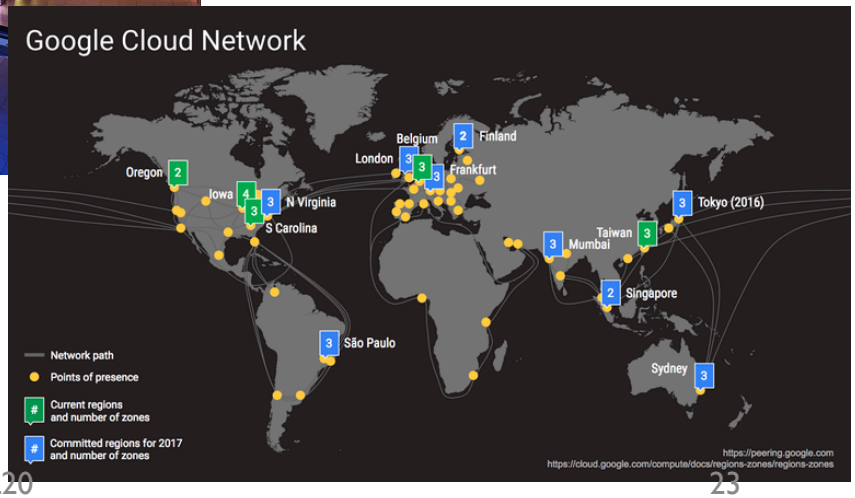
Access to storage (files) today

- File system over SSD or HDD on your local machine
- File Server in your organization (inst.eecs.Berkeley.edu)
  - Remote login (ssh), file transfer (scp) or mount
- Cloud storage
  - Accessed through web or app (drive, box, …)
  - Mounted on your local machine
  - Replicated and/or Distributed

Cloud Data Centers





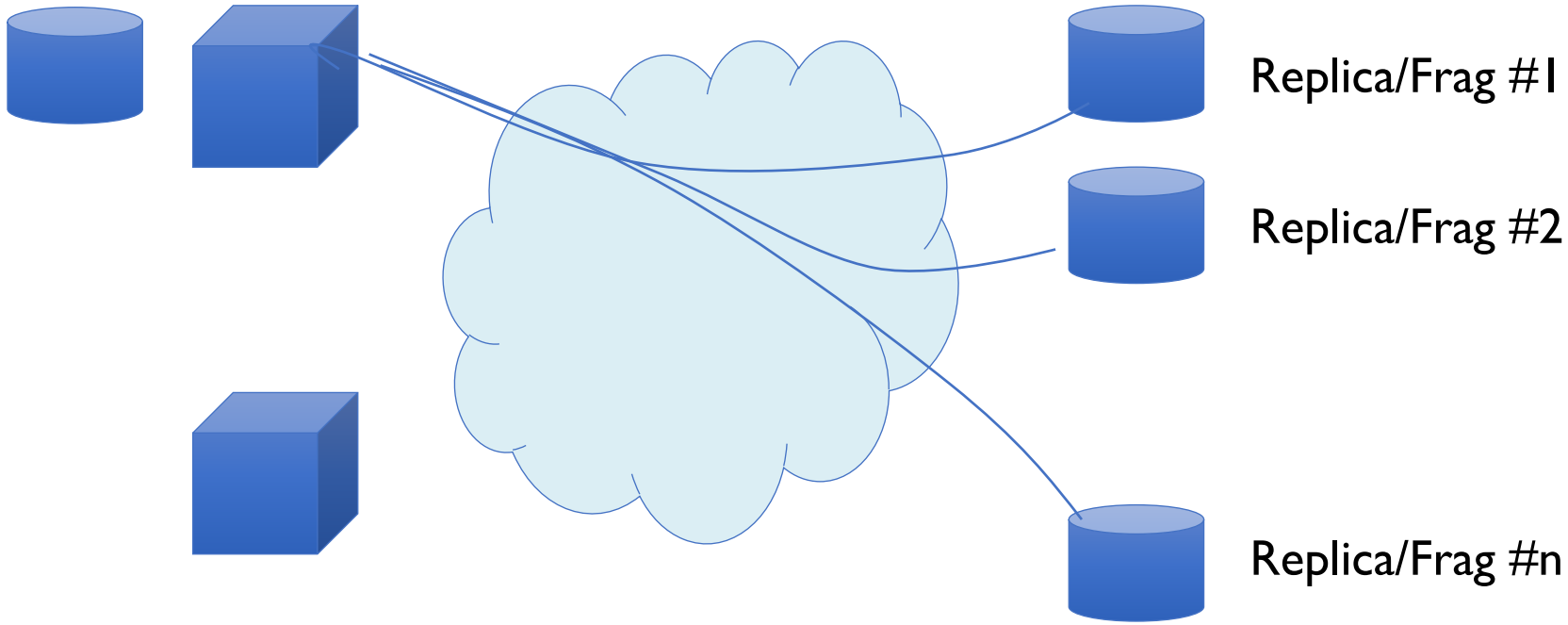


Cloud Storage Options

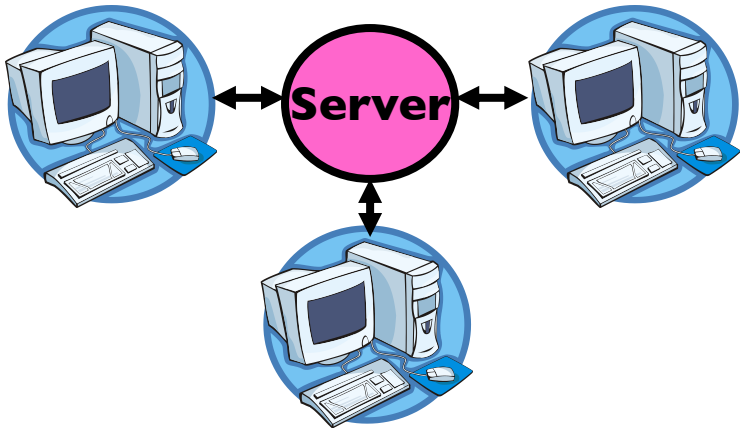
- Storage Account / Share is like disk "partition"
  - hold file system: directory, index, free map, data blocks
- Access methods: mount, REST, file xfer, synch
- Security: credentials, encryption (xfer, storage)
- Performance: HDDs, SSDs, provisioning, bursting
- Redundancy
  - Local RAID
  - Storage cluster in a Data Center
  - Zone redundant (across data centers)
  - Geographic regions

Geographic Replication – cluster, zone, geo

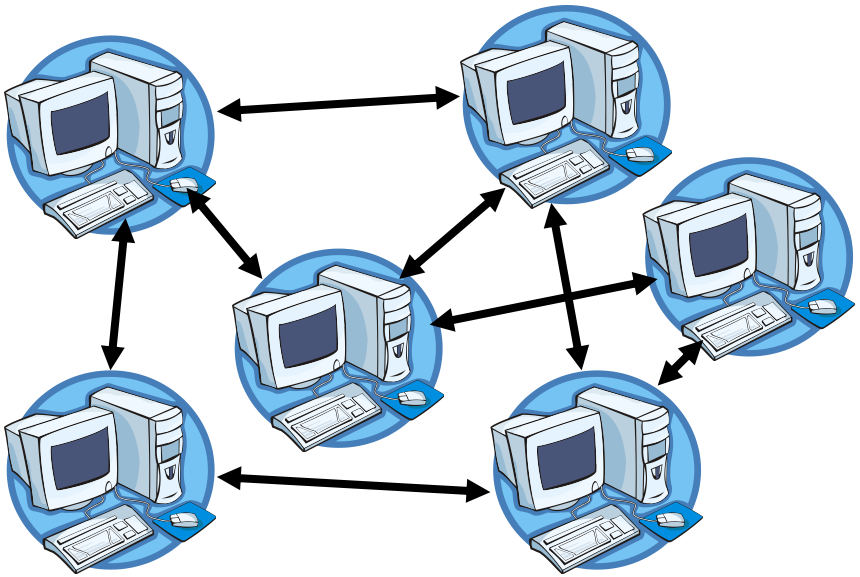
- Highly durable: Hard to destroy all copies
- Highly available for reads: Just talk to any copy
- What about for writes? Need every copy online to update all together?



Centralized vs Distributed



Client/Server Model



Peer-to-Peer Model

- **Centralized System:** Major functions performed on one physical computer
  - Many cloud services logically centralized, but not physically so
- **Distributed System:** Physically separate computers working together to perform a single task

Distributed: Why?

- Simple, **cheaper** components
- Easy to add capability **incrementally**
- Let multiple users cooperate
  - Physical components owned by different users
  - Enable **collaboration** between diverse users
- Availability - continue operation despite failures of some parts

Distributed Systems Goal

- **Transparency:** Hide "distributed-ness" from external observer, make system simpler
- **Types**
  - Location: Location of resources is invisible
  - Migration: Resources can move without user knowing
  - Replication: Invisible extra copies of resources (for reliability, performance)
  - Parallelism: Job split into multiple pieces, but looks like a single task
  - Fault Tolerance: Components fail without users knowing

Challenge of Coordination

- Components communicate over the network
  - Send messages between machines

- Need to use messages to **agree on system state**
  - in a centralized system the center "rules"

Recall: What is a Protocol?

- **An agreement on how to communicate**
  - **Syntax:** Format, order messages are sent and received
  - **Semantics:** Meaning of each message
- Described formally by a state machine
- A distributed system is embodied by a protocol

Recall: Examples of Protocols in Human Interactions

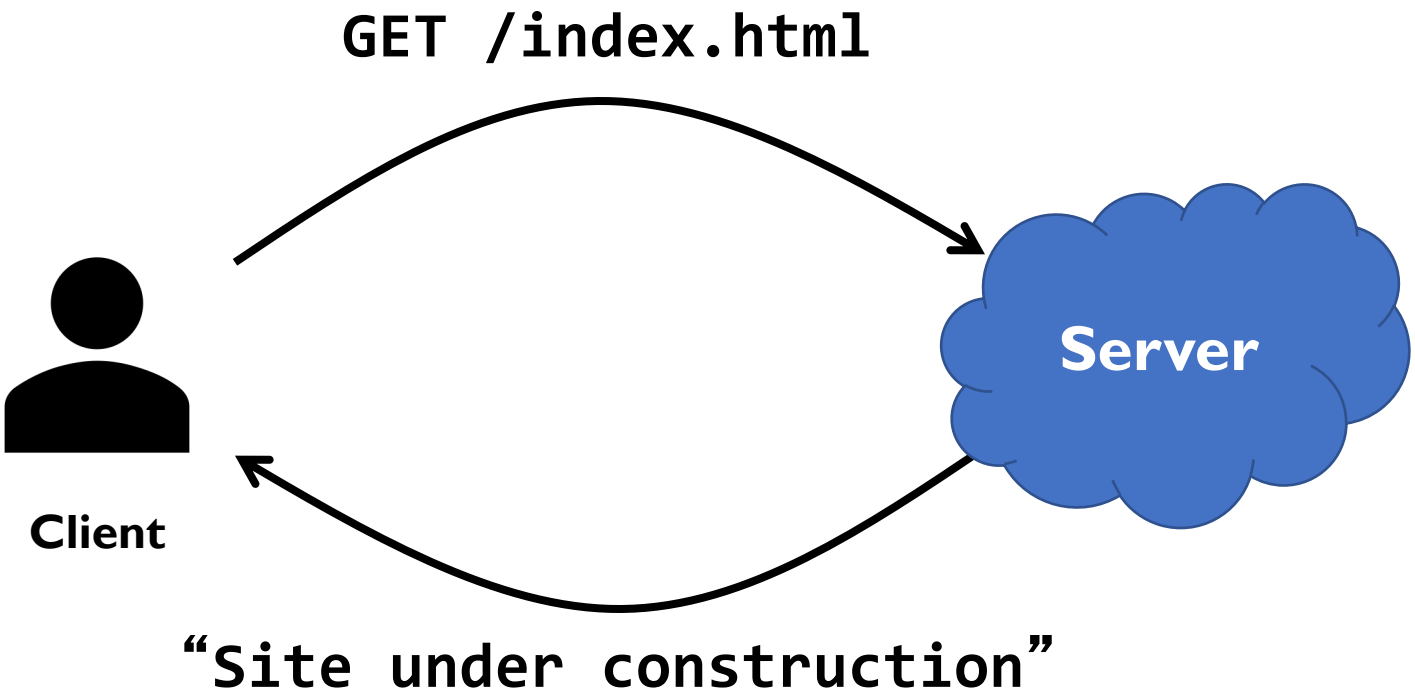
Telephone

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

Recall: Clients and Servers

- Client program
  - Running on end host
  - Requests service
  - E.g., Web browser
- Server program
  - Running on end host
  - Provides service
  - E.g., Web server



Recall: Client-Server Communication

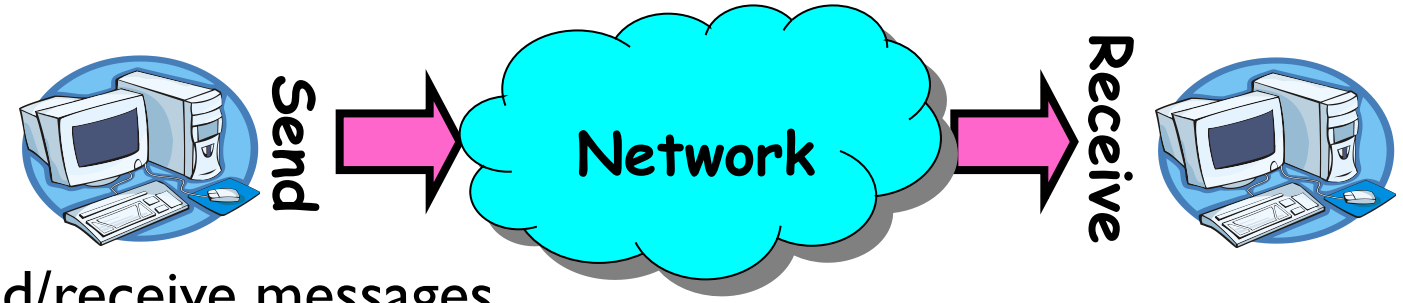
- Client "sometimes on"
  - Initiates a request to the server when interested
  - E.g., Web browser on your laptop or cell phone
  - Doesn't communicate directly with other clients
  - Needs to know the server's address
- Server is "always on"
  - Services requests from many client hosts
  - E.g., Web server for the *www.berkeley.edu*
  - Doesn't initiate contact with the clients
  - Needs a fixed, well-known address

Peer-to-Peer Communication

- No always-on server at the center of it all
  - Hosts may come and go, change addresses
  - Hosts may have a different address for each interaction with the system
- Example: Peer-to-peer file sharing (BitTorrent)
  - Any host can request files, send files, search for files
  - Scalability by harnessing millions of peers
  - Each peer acting as both client and server
- Many examples today / but also hybrids

Distributed System Protocols are Built by Message Passing

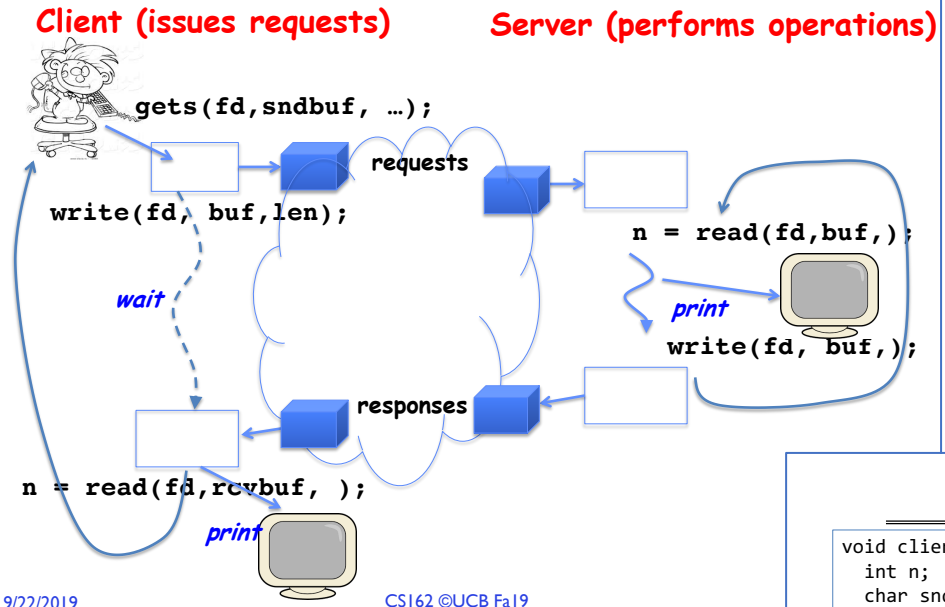
- How do you actually program a distributed application?
  - Multiple threads, running on different machines
    - How do they coordinate and communicate



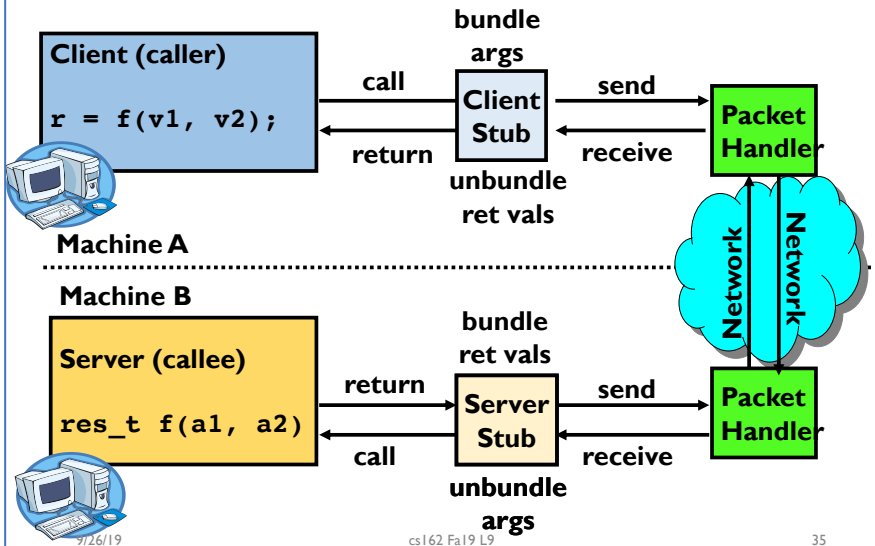
  - send/receive messages
    - Already atomic: no receiver gets portion of a message and two receivers cannot get same message
- Interface:
  - Mailbox: temporary holding area for messages
    - Includes both destination location and queue
  - `Send(message,mbox)`
    - Send message to remote mailbox 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

Recall

Silly Echo Server – running example



RPC Information Flow

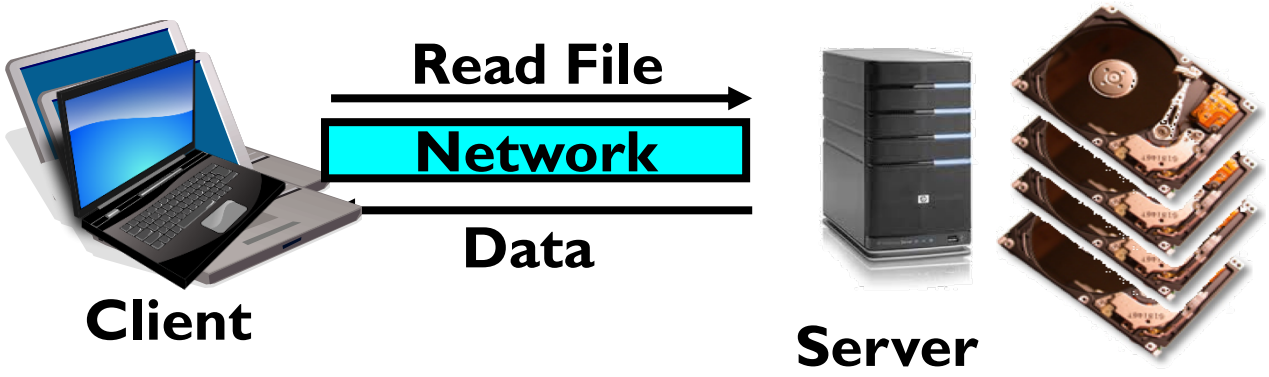


Echo client-server example

```
void client(int sockfd) {
  int n;
  char sndbuf[MAXIN]; char rcvbuf[MAXOUT];
  getreq(sndbuf, MAXIN);        /* prompt */
  while (strlen(sndbuf) > 0) {
    write(sockfd, sndbuf, strlen(sndbuf)); /* send */
    memset(rcvbuf,0,MAXOUT);             /* clear */
    n=read(sockfd, rcvbuf, MAXOUT-1);    /* receive */
    write(STDOUT_FILENO, rcvbuf, n);        /* echo */
    getreq(sndbuf, MAXIN);               /* prompt */
  }
}
```

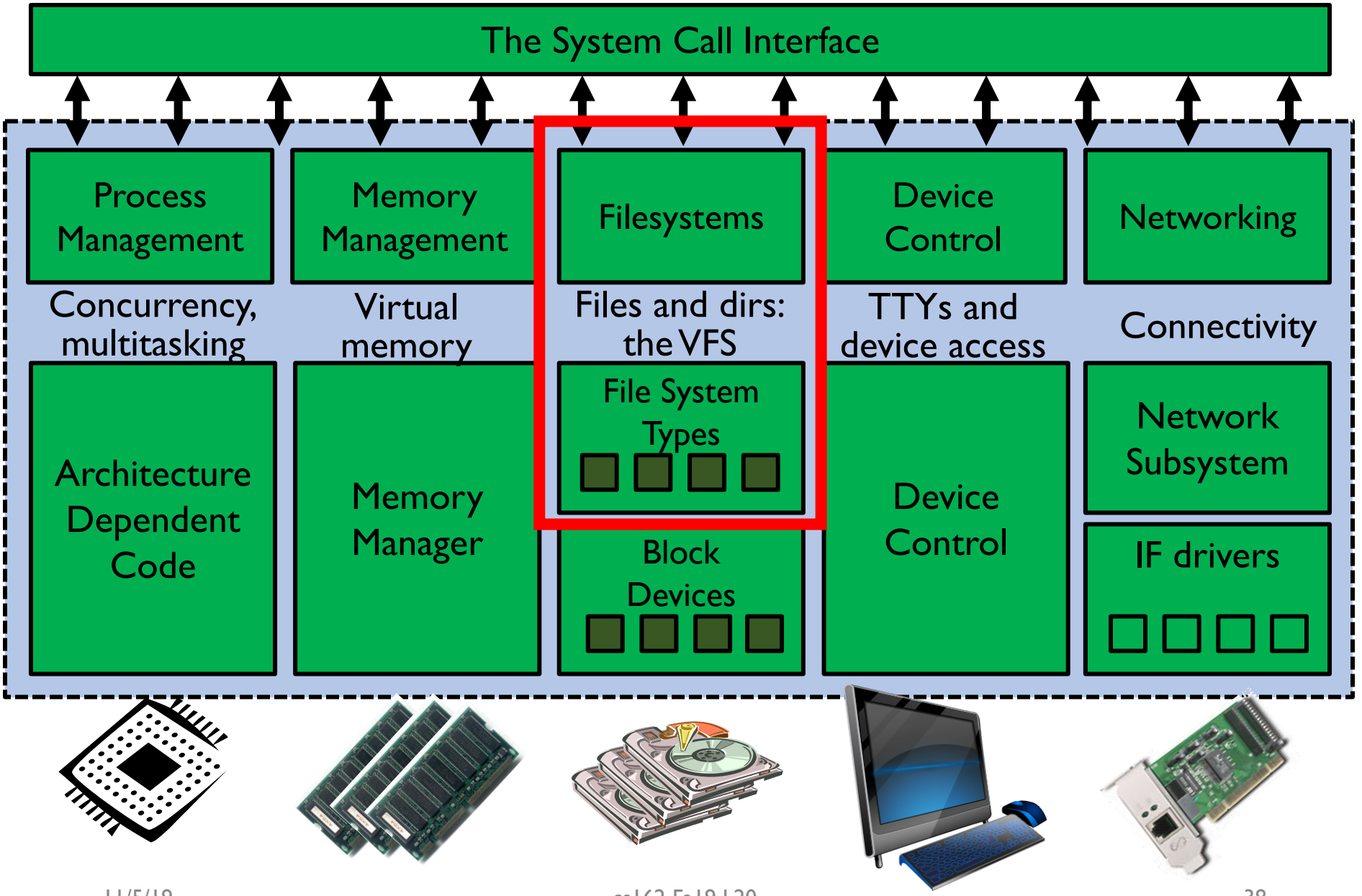
```
void server(int consockfd) {
  char reqbuf[MAXREQ];
  int n;
  while (1) {
    memset(reqbuf,0, MAXREQ);
    n = read(consockfd,reqbuf,MAXREQ-1); /* Recv */
    if (n <= 0) return;
    n = write(STDOUT_FILENO, reqbuf, strlen(reqbuf));
    n = write(consockfd, reqbuf, strlen(reqbuf)); /* echo*/
  }
}
```

Distributed File Systems



- Transparent access to files stored on a remote disk
- *Mount* remote files into your local file system
  - Directory in local file system refers to remote files
  - e.g., `/home/oksi/162/` on laptop actually refers to `/users/oski` on campus file server

Enabling Design: VFS



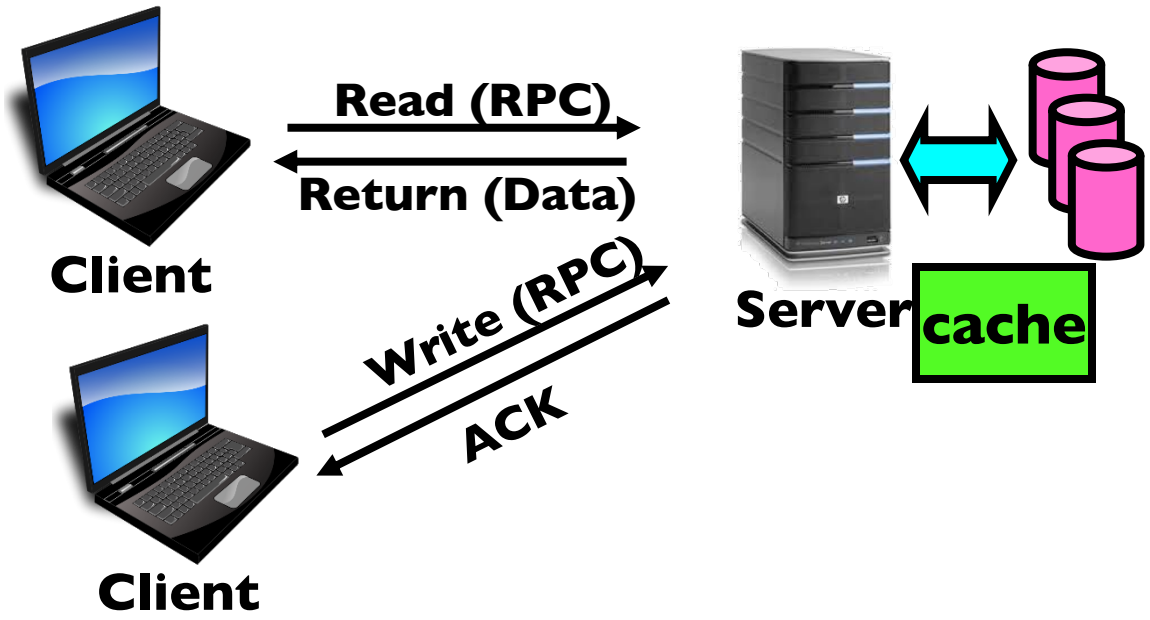
VFS (Virtual Filesystem Switch)

- Similar to device drivers: possible to plug in different implementations of the same interface
  - Just need to provide inodes, files, directories, etc.
  - Doesn't matter if these are on local disk or remote!

- **Key Idea:** Same system call interface is used to interact with many different types of filesystems

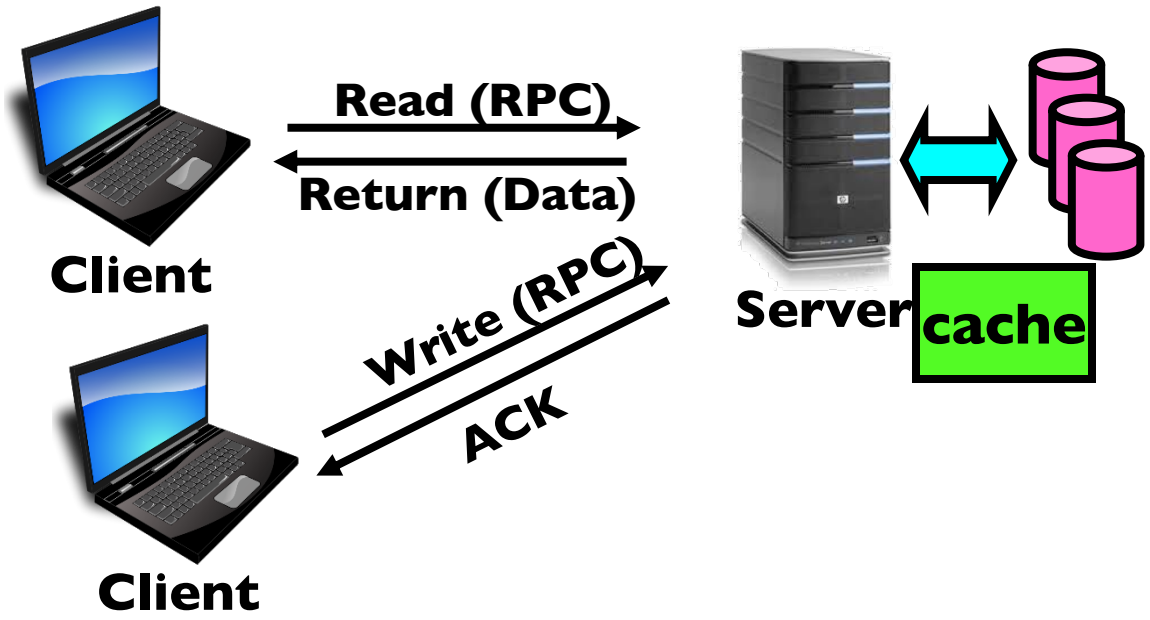
- Yet another reason why virtual machines can work well

Simple Distributed File System



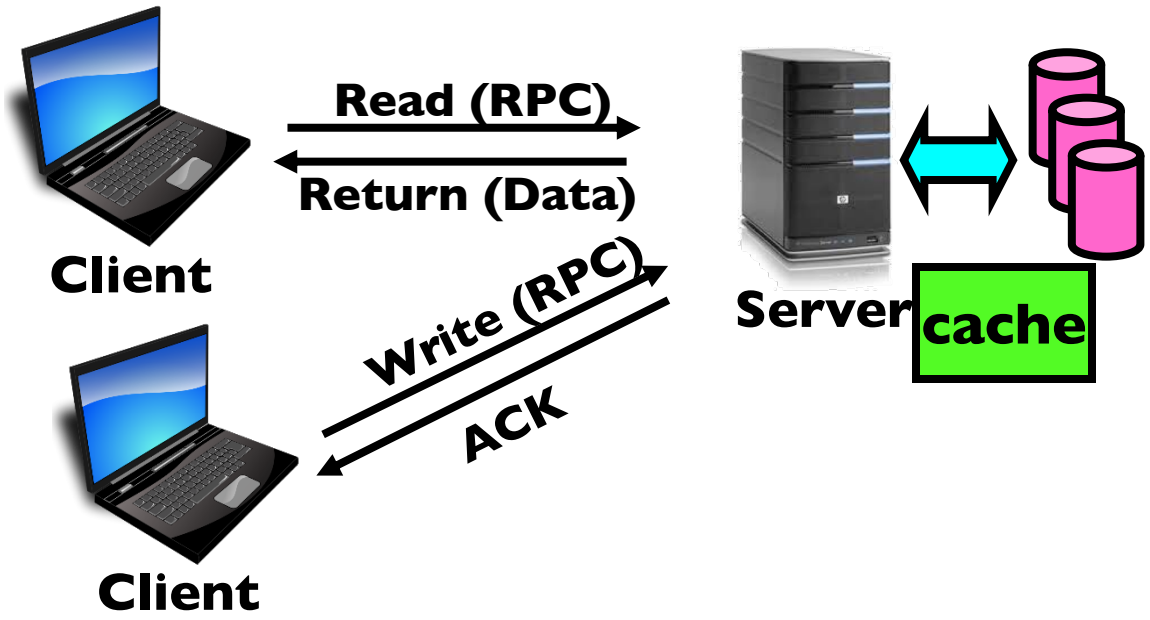
- Remote Disk: Opens, Reads, Writes, Closes forwarded to server
  - Use Remote Procedure Calls (RPC) to translate file system calls into remote requests
  - Server may cache files in memory to response more quickly

Simple Distributed File System



- Advantage: Server acts as final authority on file contents
  - Can be engineered for reliability / availability
- Sharing files is natural

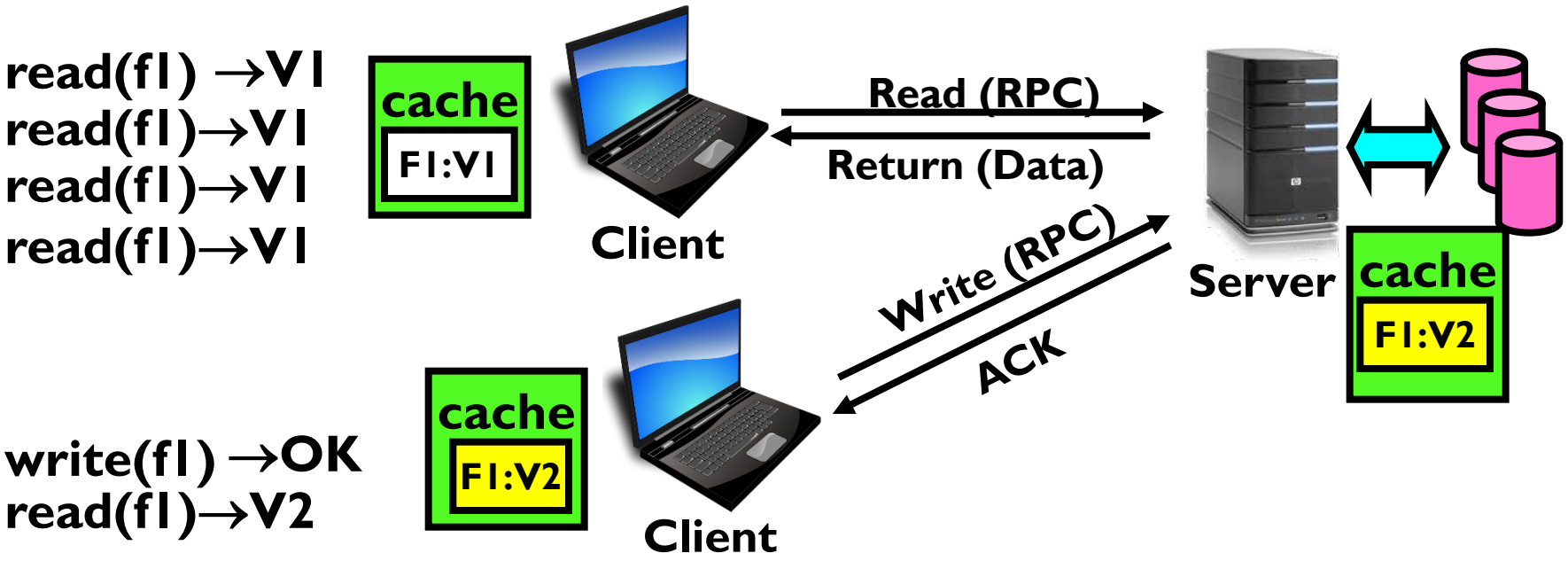
Simple Distributed File System



- Performance issues
  - Server may be a bottleneck
  - Going across network is much slower than local memory
    - What about local disk?
- How might we overcome these?

Break

Local Caching



- Idea: Use caching to reduce network load
  - In practice: use buffer cache at source and destination
- Advantage: if open/read/write/close can be done locally, don't need to do any network traffic
- New problem: Consistency across caches

Dealing with Failures

- What if server crashes? Can client wait until it comes back and just continue making requests?
  - Changes in server's cache but not in disk are lost

- What if there is shared state across RPC's?
  - Client opens file, then does a seek
  - Server crashes
  - What if client wants to do another read?
- Similar problem: What if client removes a file but server crashes before acknowledgement?

Stateless Protocol

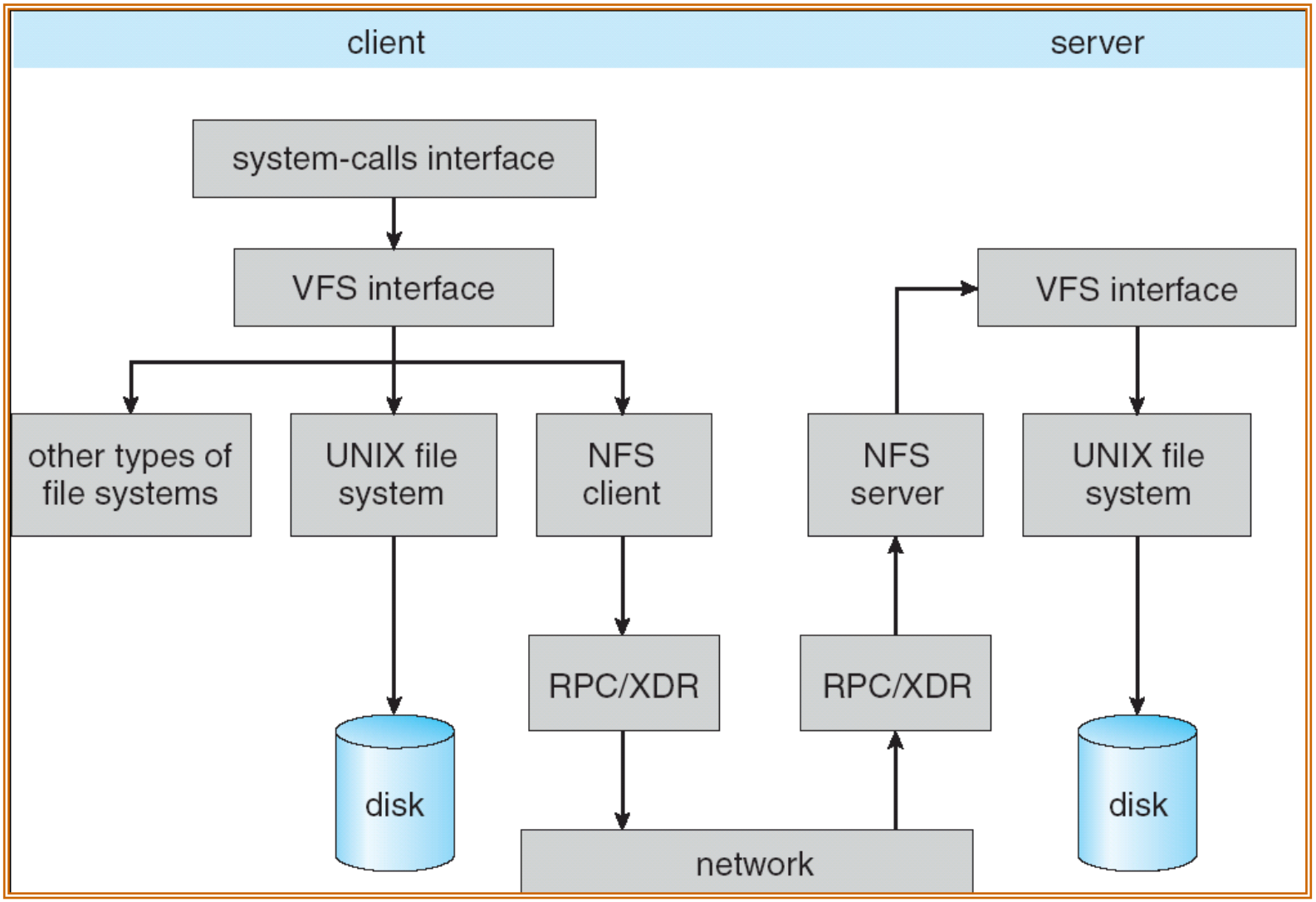
- A protocol in which **all information** required to service a request is included with the request
- Even better: **Idempotent Operations** – repeating an operation multiple times is same as executing it just once (e.g., storing to a mem addr.)
- Client: timeout expires without reply, just run the operation again (safe regardless of first attempt)

- Recall HTTP: Also a stateless protocol
  - Include cookies with request to simulate a session

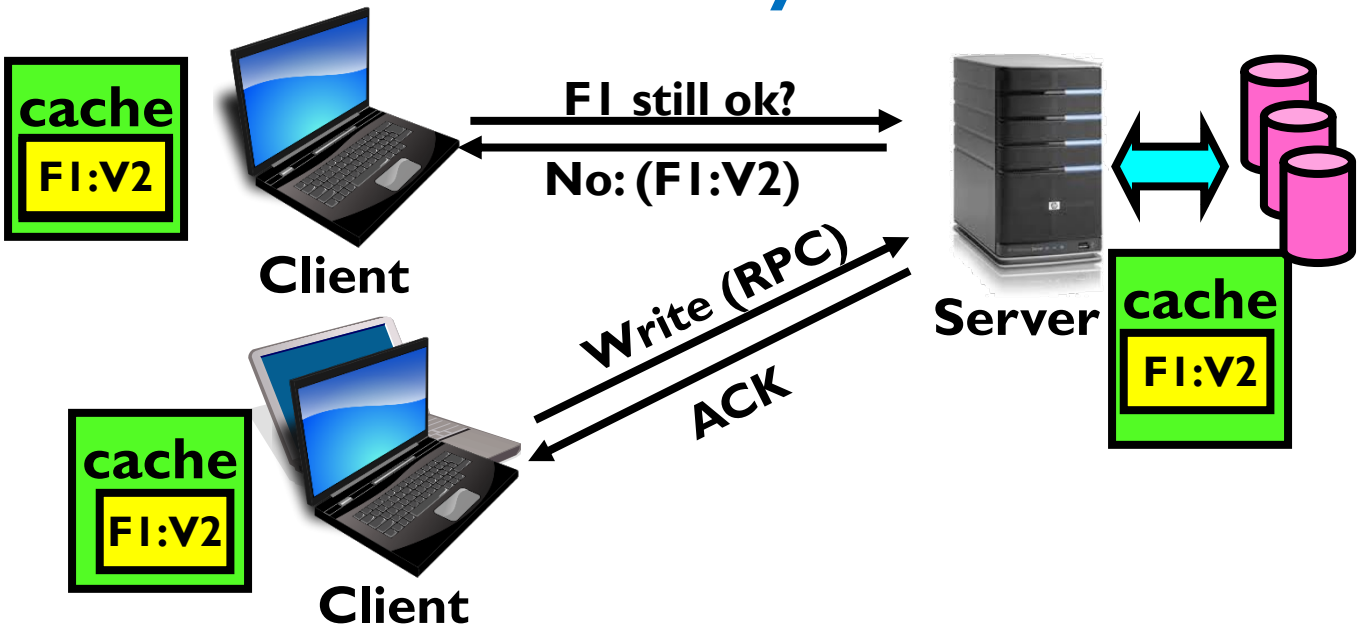
Network File System (Sun)

- Defines an RPC protocol for clients to interact with a file server
  - E.g., read/write files, traverse directories, …
  - **Stateless** to simplify failure cases
- Keeps most operations idempotent
  - Even removing a file: Return advisory error second time
- **Don't buffer writes** on server side cache
  - Reply with acknowledgement **only** when modifications reflected on disk

NFS Architecture

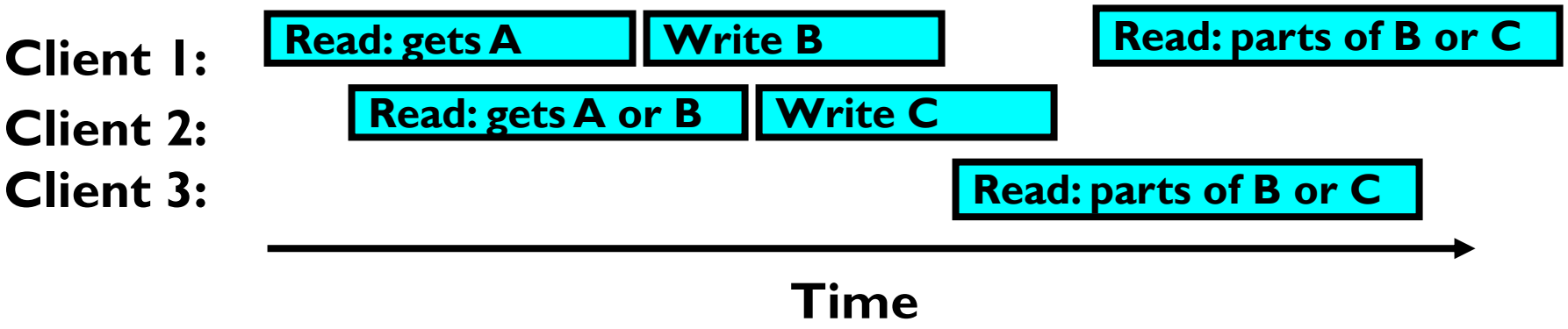


NFS Cache consistency



- Clients flush local changes to server on `close()`
- Clients periodically contact server to check if local file version is out of date
  - 3-30 sec. intervals (configuration parameter)
- What if multiple clients write to same file?
  - No guarantees: could see either version, or parts of both

Sequential Ordering Constraints



- What if we wanted to match single-machine case?
  - If read finishes before write starts, get old copy
  - If read starts after write finishes, get new copy
  - Otherwise, get either new or old copy
- For NFS:
  - If read starts more than 30 seconds after write, get new copy; otherwise, could get partial update

In Everyday Life

Where do we meet?		
Where do we meet?	Where do we meet? At Nefeli's	Where do we meet? At Top Dog
Where do we meet? At Nefeli's At Top Dog	Where do we meet? At Nefeli's At Top Dog	Where do we meet? At Nefeli's At Top Dog
Where do we meet? At Nefeli's	Where do we meet? At Top Dog At Nefeli's	Where do we meet? At Nefeli's At Top Dog

The Shared Storage Abstraction

- Information (and therefore control) is communicated from one point of computation to another by
  - The former storing/writing/sending to a location in a shared address space
  - And the second later loading/reading/receiving the contents of that location

- Memory (address) space of a process
- File systems
- Dropbox, …
- Google Docs, …
- Facebook, …

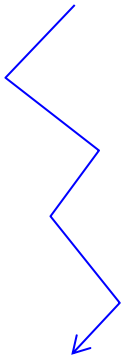
Interlude

What are you assuming?

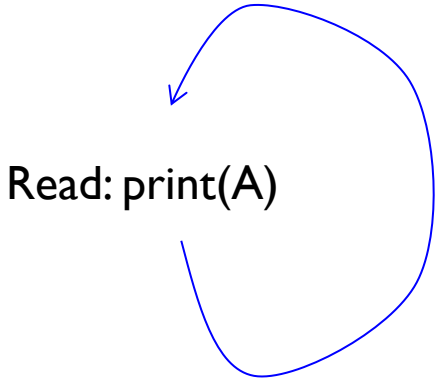
- Writes happen
  - Eventually a write will become visible to readers
  - Until another write happens to that location
- Within a sequential thread, a read following a write returns the value written by that write
  - Dependences are respected
  - Here a control dependence
  - Each read returns the most recent value written to the location

For example

Write: A := 162



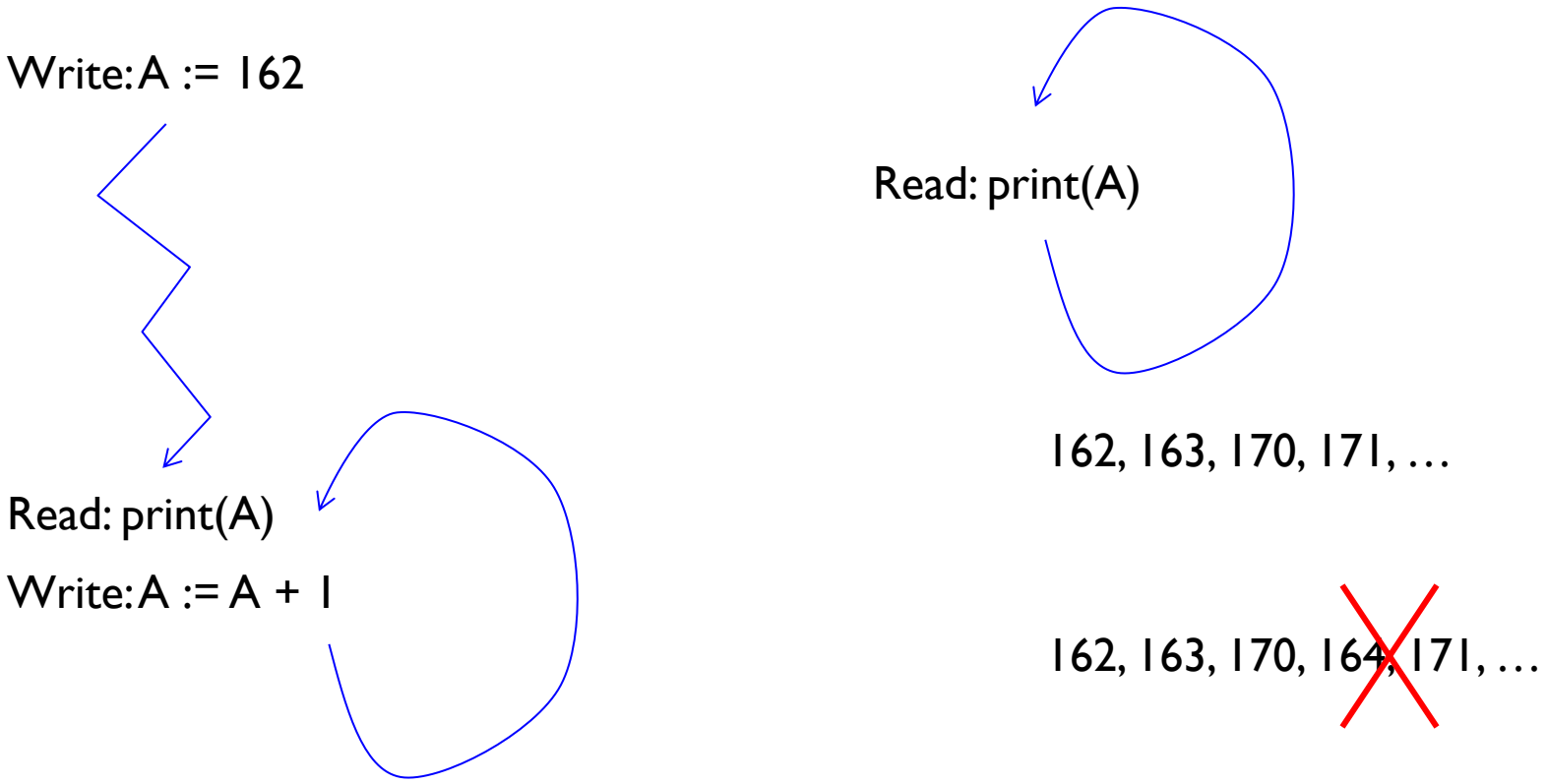
Read: print(A)



What are you assuming?

- Writes happen
  - Eventually a write will become visible to readers
  - Until another write happens to that location
- Within a sequential thread, a read following a write returns the value written by that write
  - Dependences are respected
  - Here a control dependence
  - Each read returns the most recent value written to the location
- A sequence of writes will be visible in order
  - Control dependences
  - Data dependences

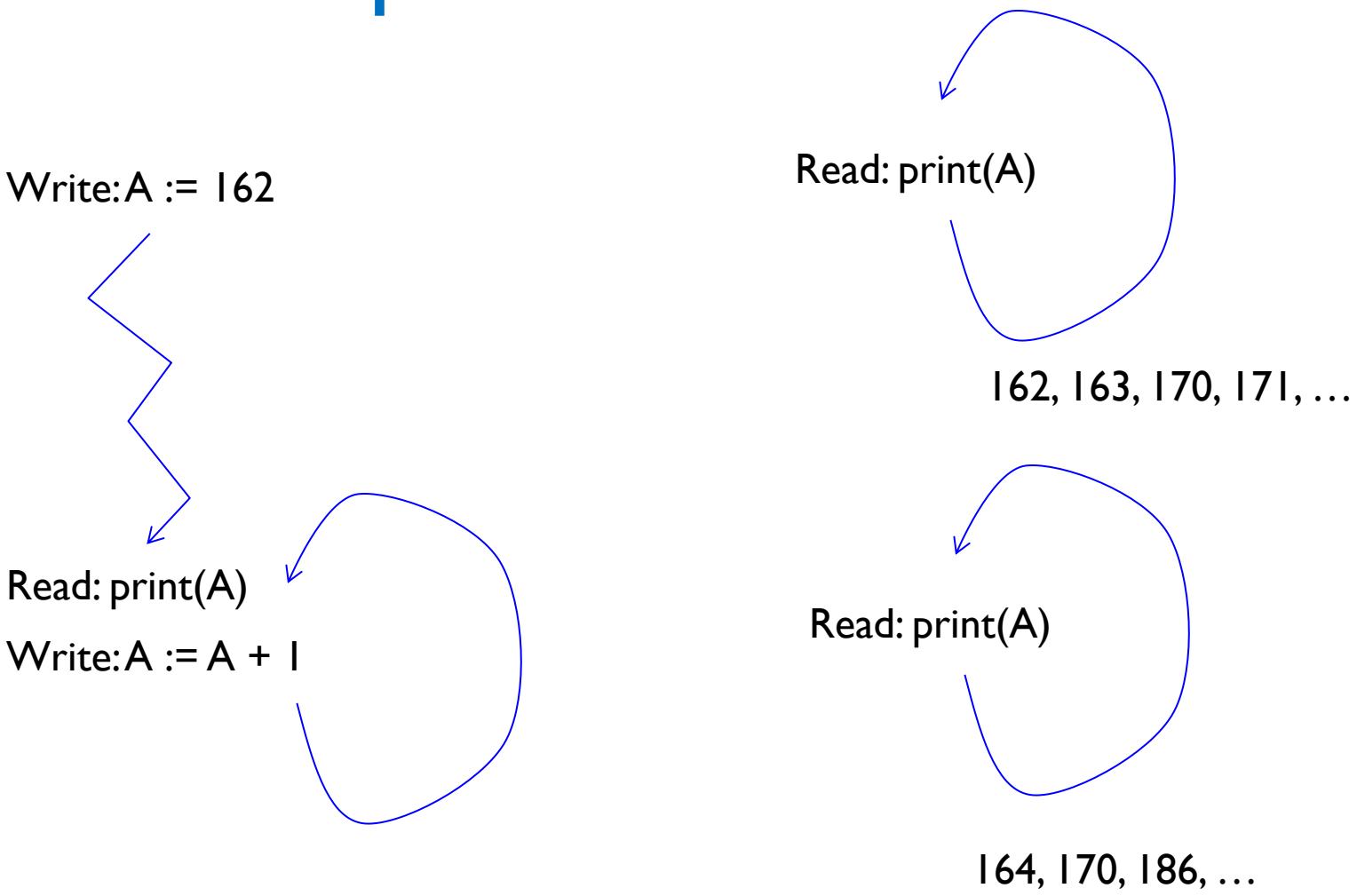
For example


Write: A := 162
Read: print(A)
Write: A := A + 1
Read: print(A)
162, 163, 170, 171, …
162, 163, 170, 164, 171, …

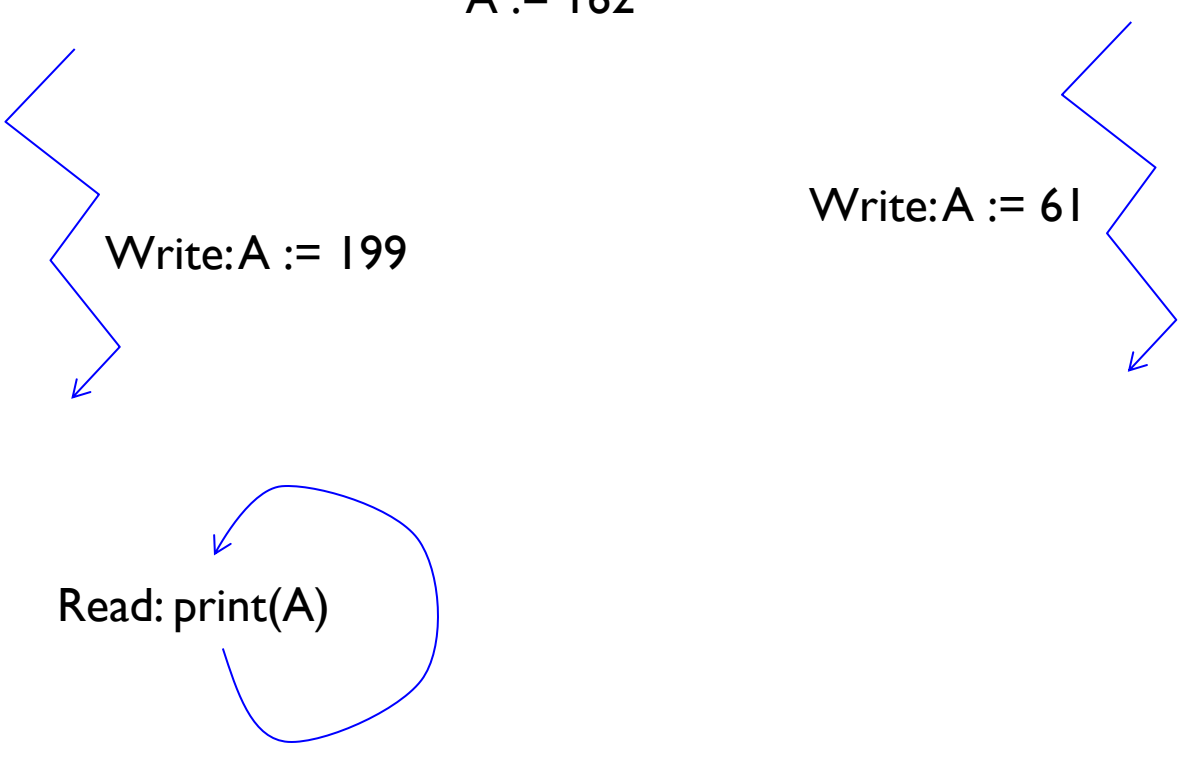
What are you assuming?

- Writes happen
  - Eventually a write will become visible to readers
  - Until another write happens to that location
- Within a sequential thread, a read following a write returns the value written by that write
  - Dependences are respected
  - Here a control dependence
  - Each read returns the most recent value written to the location
- A sequence of writes will be visible in order
  - Control dependences
  - Data dependences
  - May not see every write, but the ones seen are consistent with order written
- All readers see a consistent order
  - It is as if the total order was visible to all and they took samples

For example


Write: A := 162
Read: print(A)
Write: A := A + 1
Read: print(A)
162, 163, 170, 171, …
Read: print(A)
164, 170, 186, …

For example



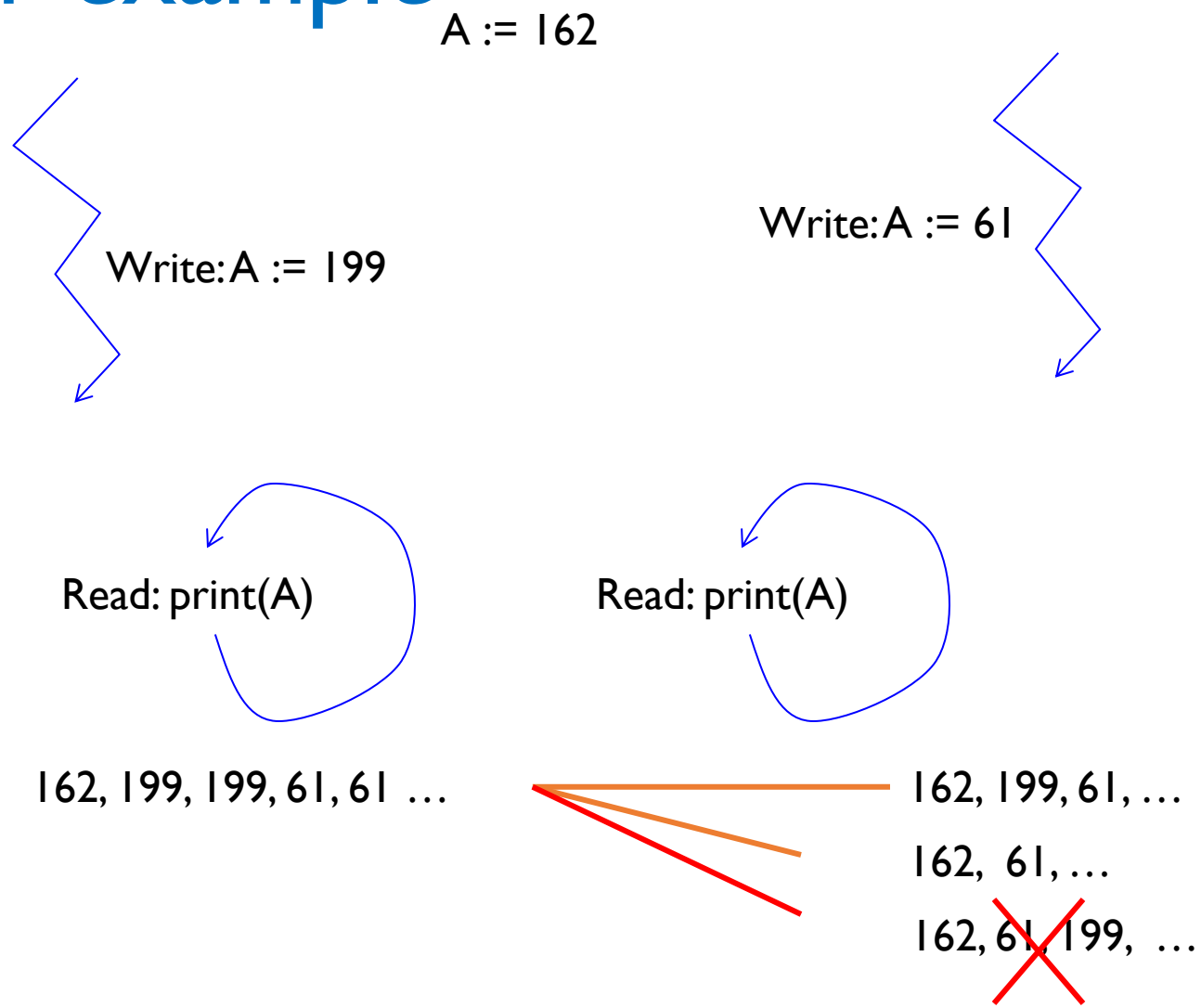
162, 199, 199, 61, 61 …

162, 61, 199, …

61, 199, …

162, 199, 61, 199 …

For example


A := 162
Write: A := 199
Write: A := 61
Read: print(A)
Read: print(A)
162, 199, 199, 61, 61 …
162, 199, 61, …
162, 61, …
162, 61, 199, …

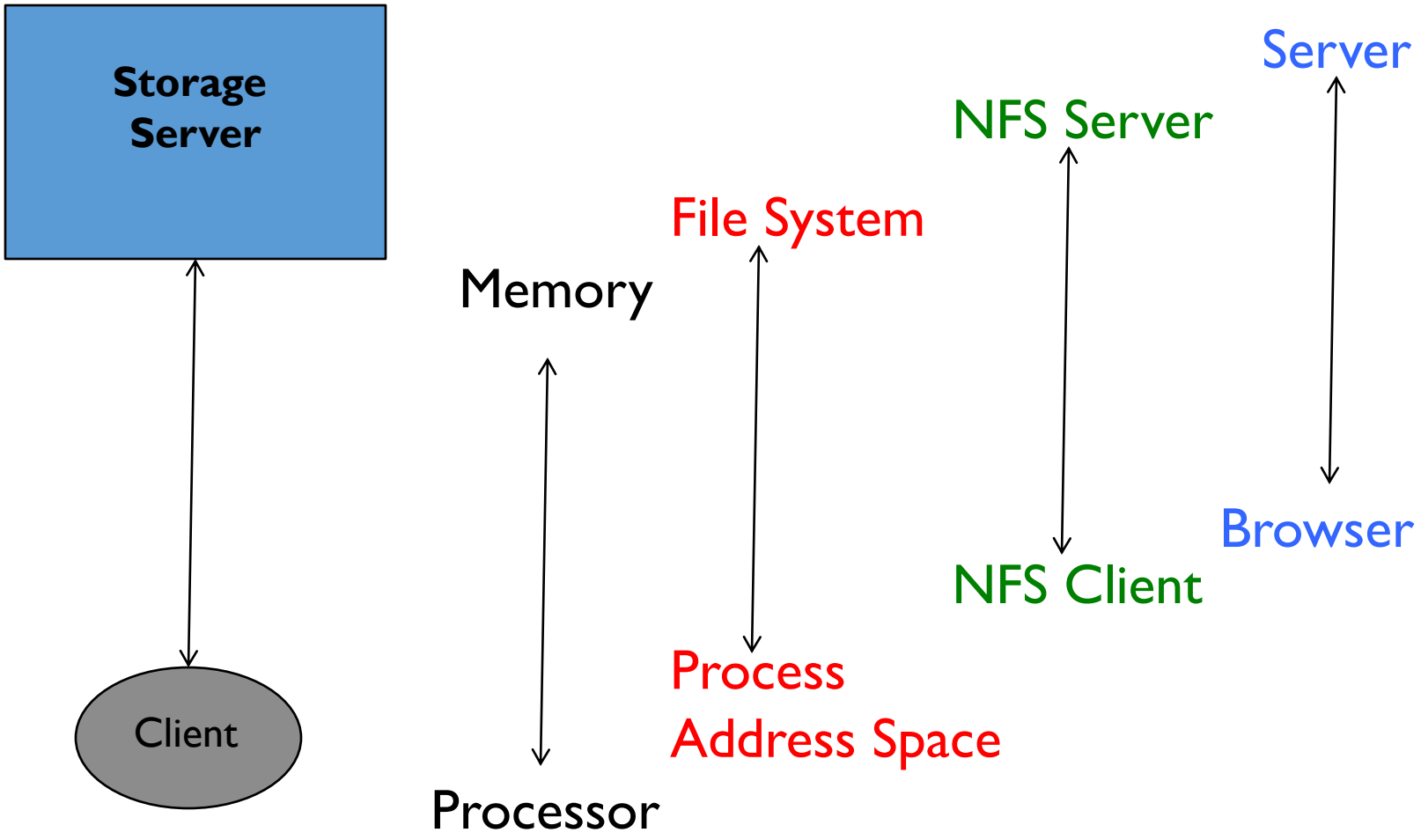
What is the key to performance AND reliability

- Replication

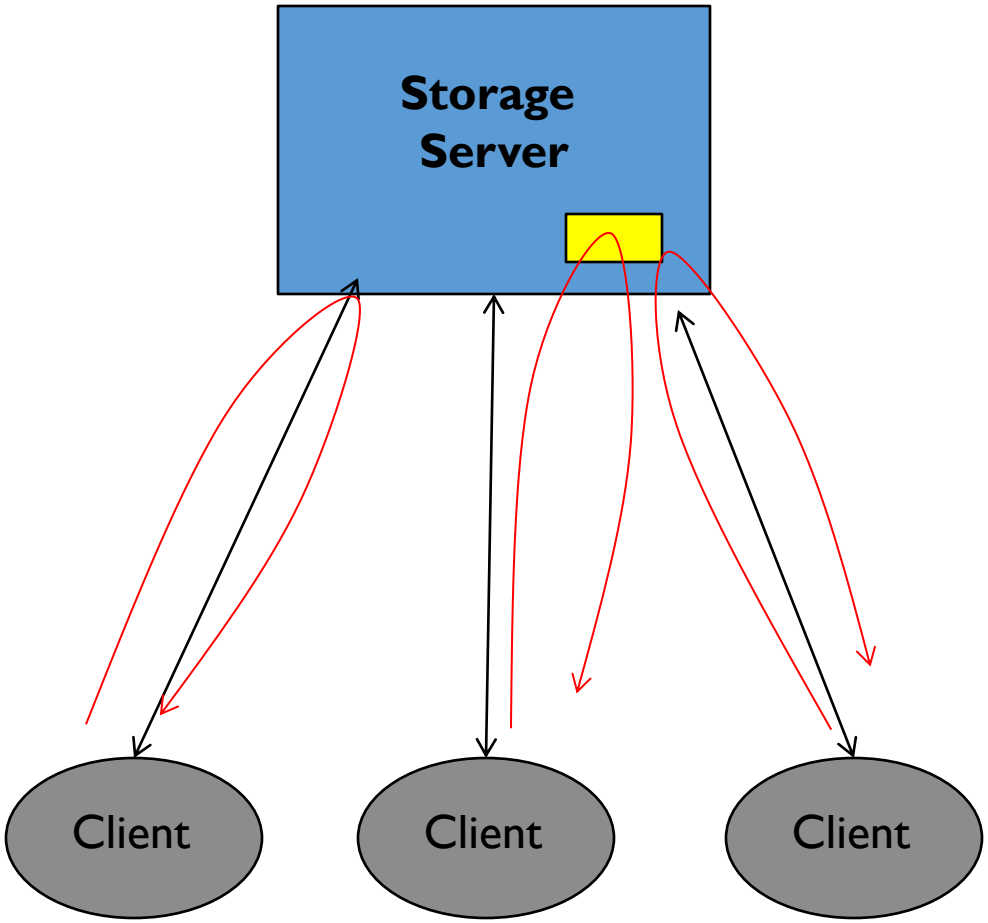
What is the source of inconsistency?

- Replication

Any Storage Abstraction

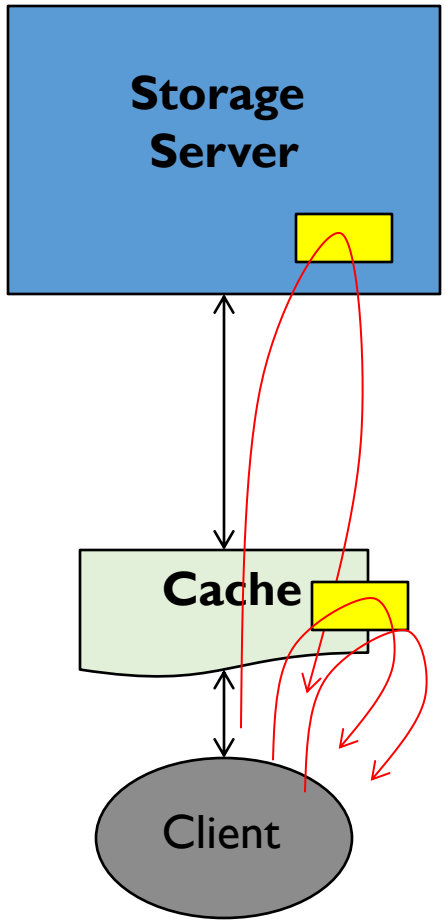


Multiple Clients access server: OK



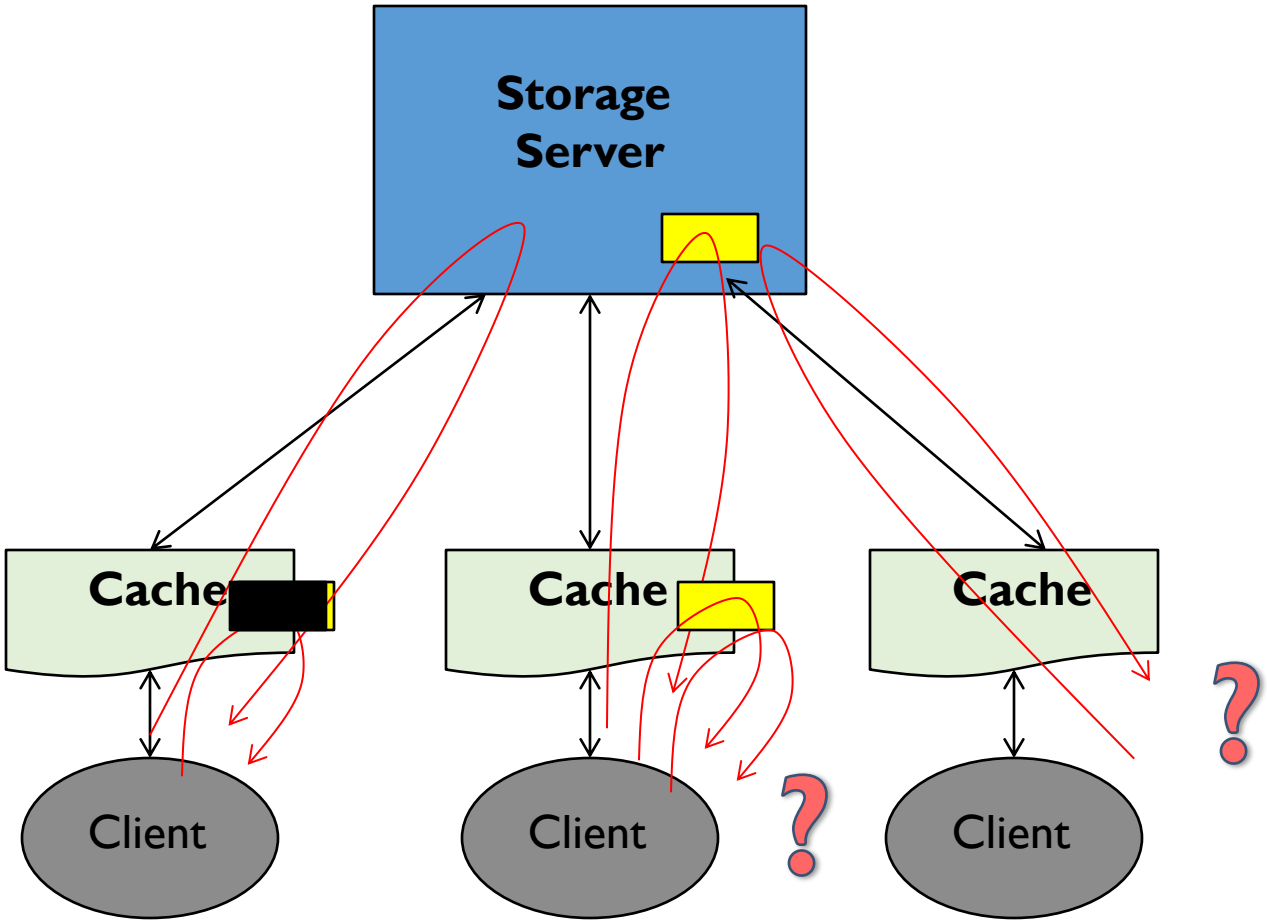
- But slow

Multi-level Storage Hierarchy: OK



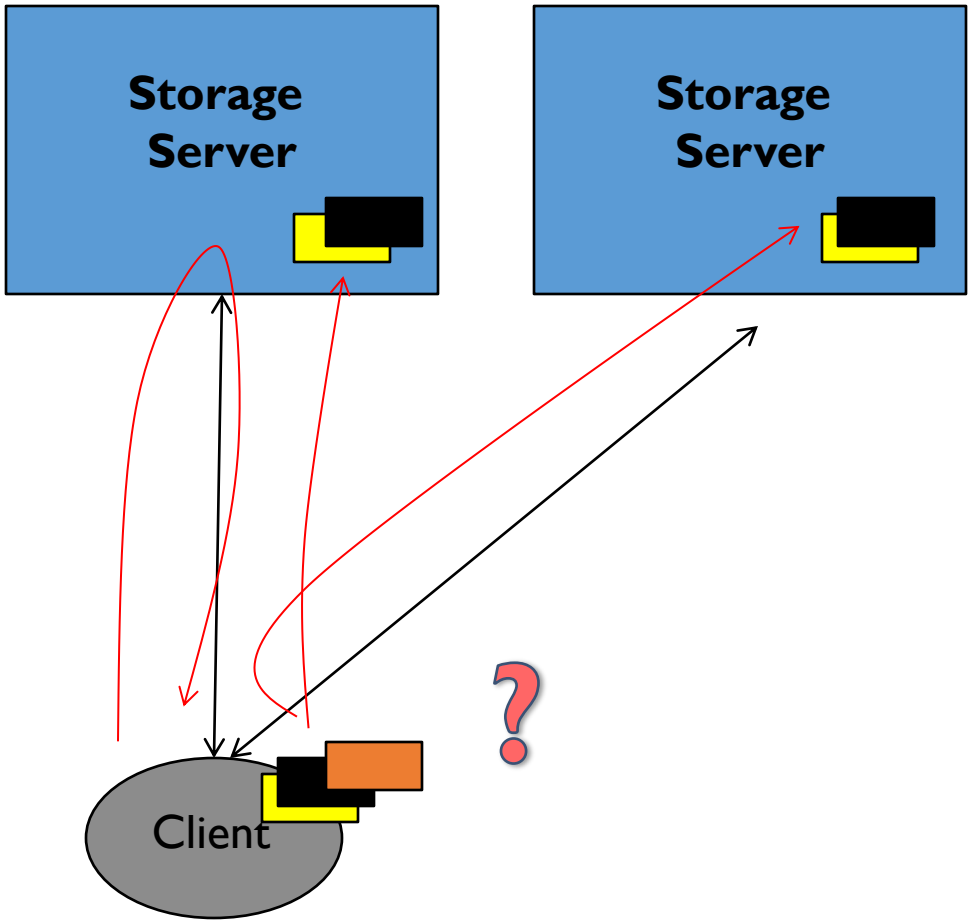
- Replication within storage hierarchy to make it fast

Multiple Clients and Multi-Level



- Fast, but not OK

Multiple Servers

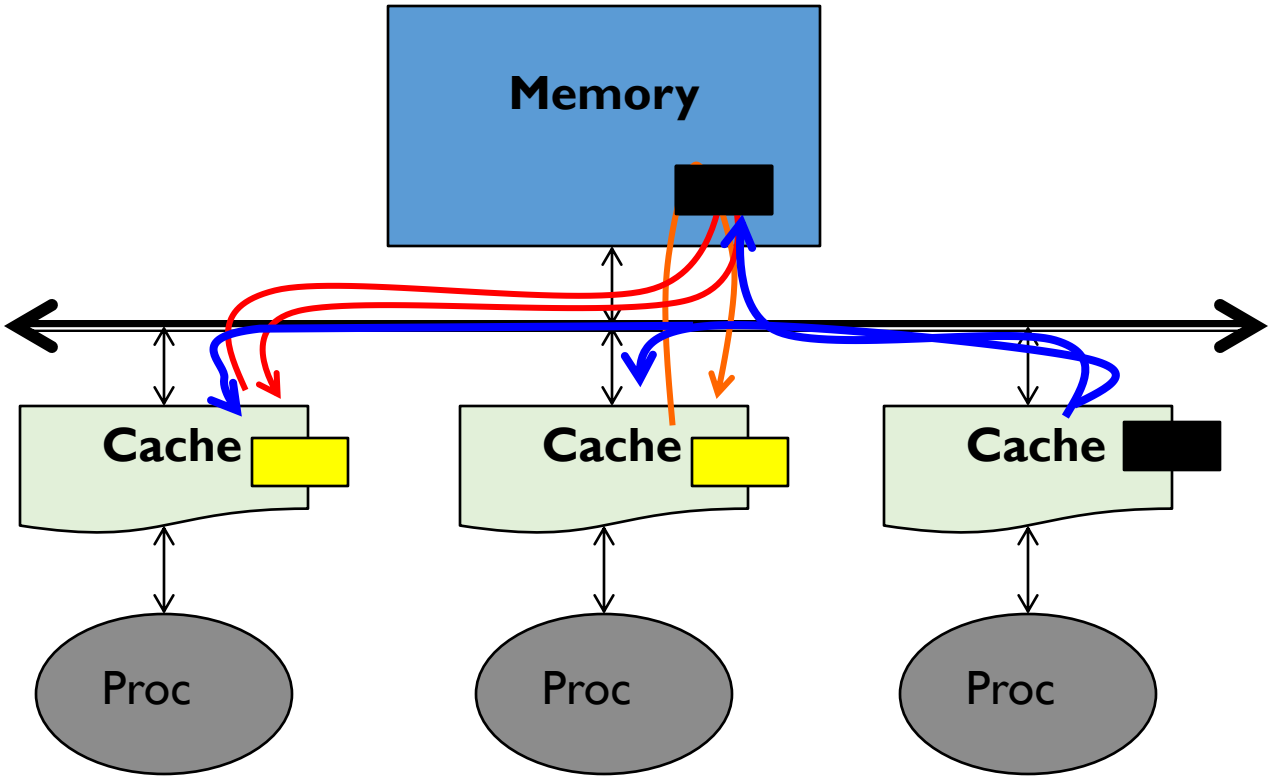


- What happens if cannot update all the replicas?
- Availability => Inconsistency

Basic solution to multiple client replicas

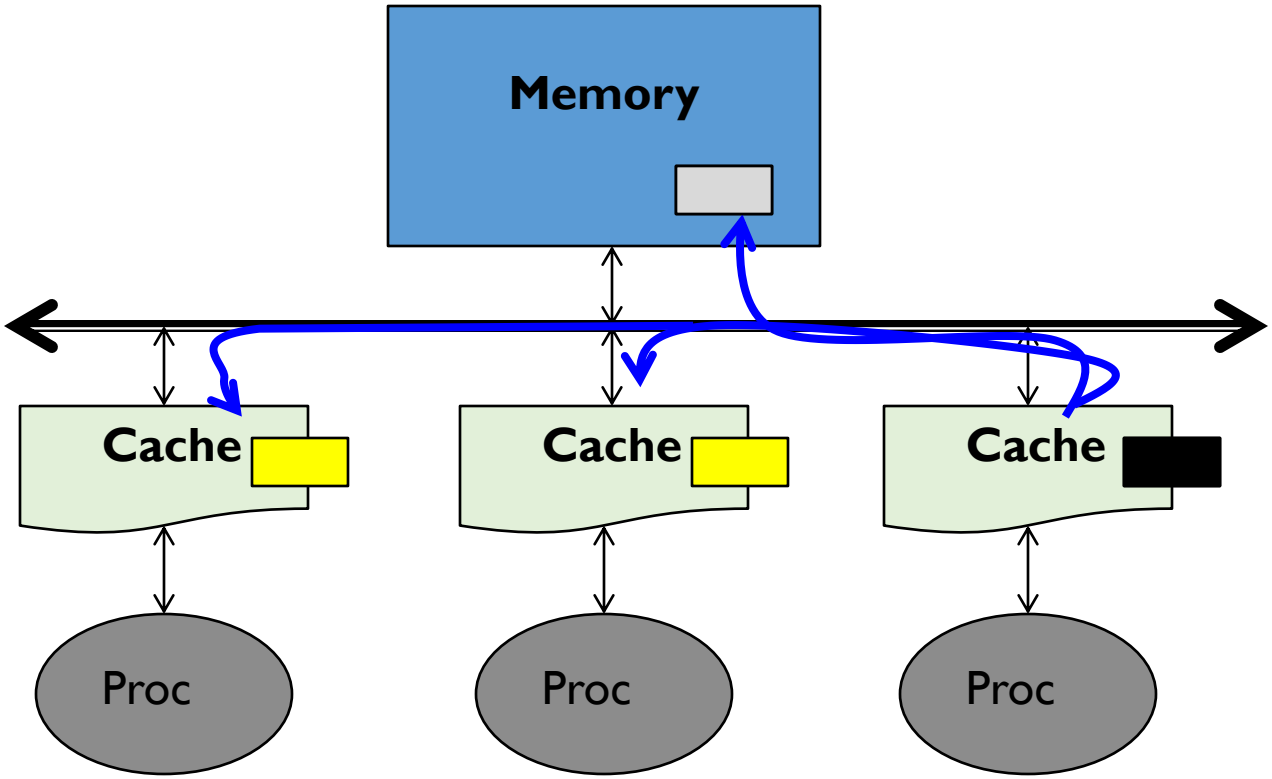
- Enforce single-writer multiple reader discipline
- Allow readers to cache copies
- Before an update is performed, writer must gain exclusive access
- Simple Approach: invalidate all the copies then update
- Who keeps track of what?

The Multi-processor/Core case



- Interconnect is a broadcast medium
- All clients can observe all writes and invalidate local replicas (write-thru invalidate protocol)

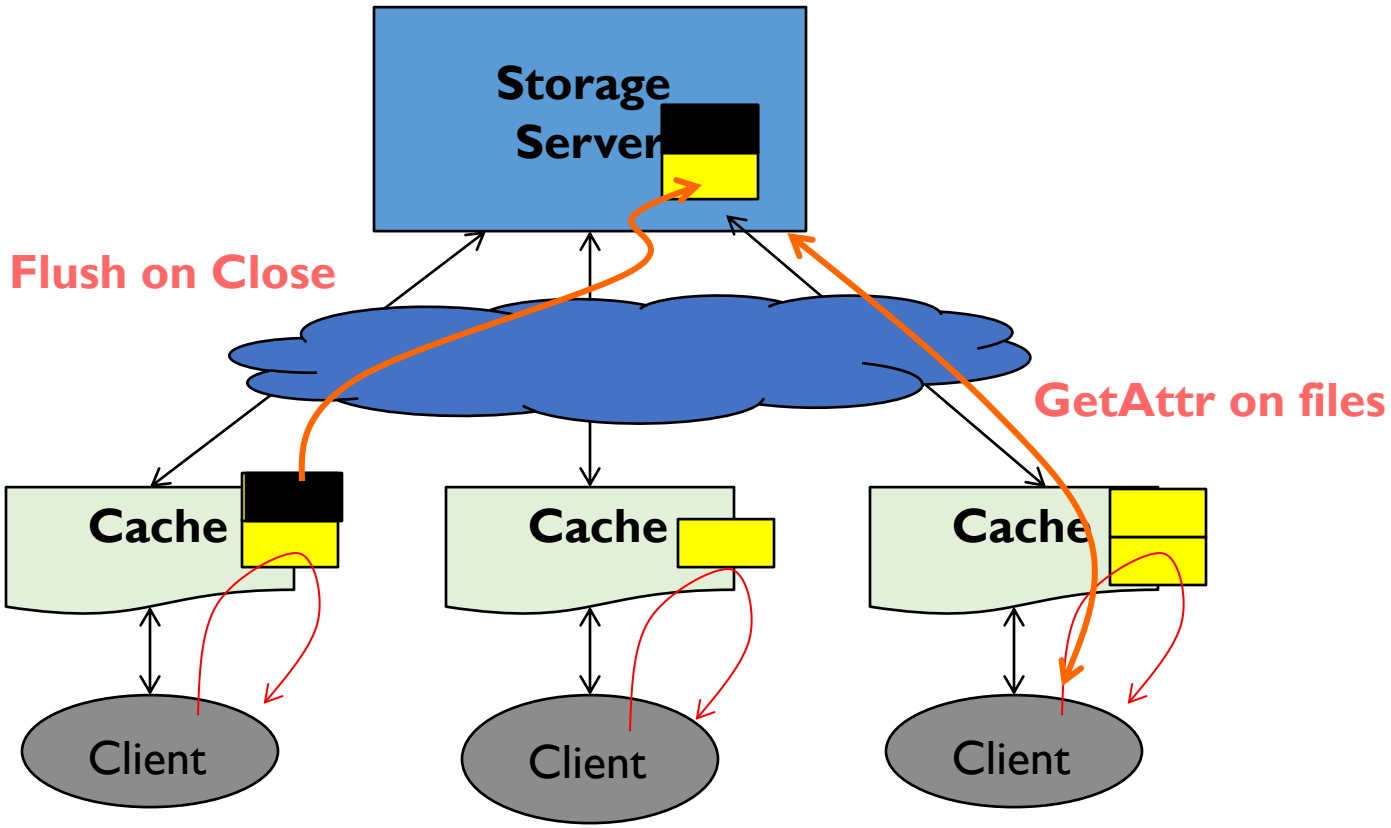
The Multi-processor/Core case



- Write-Back via read-exclusive
- Atomic Read-modify-write

~ Interlude

NFS "Eventual" Consistency



- Stateless server allows multiple cached copies
  - Files written locally (at own risk)
- Update Visibility by "flush on close"
- GetAttributes on file ops to check modify since cache

Other Options

- Server can keep a "directory" of cached copies
- On update, sends invalidate to clients holding copies
- Or can send updates to clients
- Pros and Cons ???

- OS "Consistency" ≈ Architecture "Coherence"
  - invalidate copies prior to write (or atomic write thru)
  - OS Sequential Ordering ≈ Arch. (Sequential) Consistency
- Write buffer treated as primary copy
  - like transaction log

Network File System Pros/Cons

+ Simple, highly portable
  - Just need to speak RPC protocol to participate

- Sometimes inconsistent
- Doesn't scale well to lots of clients
  - Clients keep checking to see if their caches stale
  - Server becomes bottleneck due to polling messages

Andrew File System (AFS)

- Clients cache **entire files** (on local disk) rather than individual data blocks upon an open
- All reads/writes occur against local copy
  - Reduces network traffic
- Changes flushed to server on `close`
  - Clients don't see partial updates – all or nothing!
- *Callbacks* – **server** tracks who has copies of each file, **informs them** if their copy is now stale
  - Client will fetch new version on next open

Andrew File System (AFS)

- Clients no longer need to poll server for cache invalidation, less network traffic

- Client disk as cache: More files can be cached
  - Read only workload: No need to involve server

- Consistency still has issues but easier to describe
  - Two clients have file open at same time and both write: last to `close` wins (overwrites other client's update)

Failure in AFS

- Client fails?
  - Need to double check validity of all cached files
  - May have missed callback alerts from server while down
- Server fails?
  - Clients must be made aware of this
  - Clients must reestablish callbacks
- Callbacks mean server maintains more state than in NFS design

NFS/AFS Issues

- Performance: Central file server is a bottleneck
- Availability: Server is a single point of failure
- Higher cost for server hardware, maintenance compared to client machines

Sharing Data, rather than Files ?

- Key:Value stores are used everywhere
- Native in many programming languages
  - Associative Arrays in Perl
  - Dictionaries in Python
  - Maps in Go
  - …
- What about a collaborative key-value store rather than message passing or file sharing?
- Can we make it scalable and reliable?

*Time permitting

Key Value Storage

Simple interface

- `put(key, value); // Insert/write "value" associated with key`
- `get(key); // Retrieve/read value associated with key`

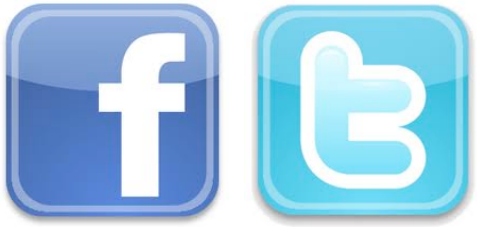
Why Key Value Storage?

- Easy to Scale
  - Handle huge volumes of data (e.g., petabytes)
  - Uniform items: distribute easily and roughly equally across many machines
- Simple consistency properties
- Used as a simpler but more scalable "database"
  - Or as a building block for a more capable DB

Key Values: Examples

- Amazon:
  - Key: customerID
  - Value: customer profile (e.g., buying history, credit card, ..)

- Facebook, Twitter:
  - Key: UserID
  - Value: user profile (e.g., posting history, photos, friends, …)



- iCloud/iTunes:
  - Key: Movie/song name
  - Value: Movie, Song

KV Storage Systems in the Wild

- **Amazon**
  - DynamoDB: internal key value store used to power Amazon.com (shopping cart)
  - Simple Storage System (S3)

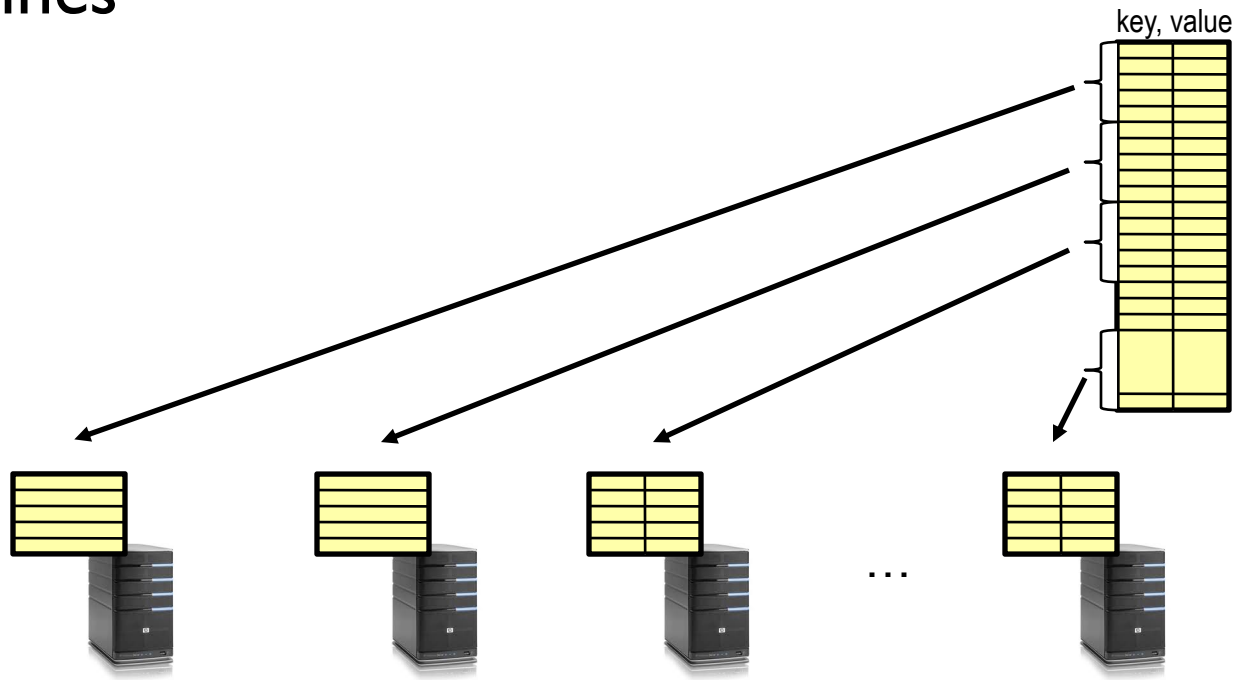
- **BigTable/HBase/Hypertable:** distributed, scalable data storage

- **Cassandra**: "distributed data management system" (developed by Facebook)

- **Memcached:** in-memory key-value store for small chunks of arbitrary data (strings, objects)

Key Value Store

- Also called Distributed Hash Tables (DHT)
- Main idea: **partition** set of key-value pairs across many machines



Important Questions

- **put(key, value)**:
  - where do you store a new (key, value) tuple?
- **get(key)**:
  - where is the value associated with a given "key" stored?

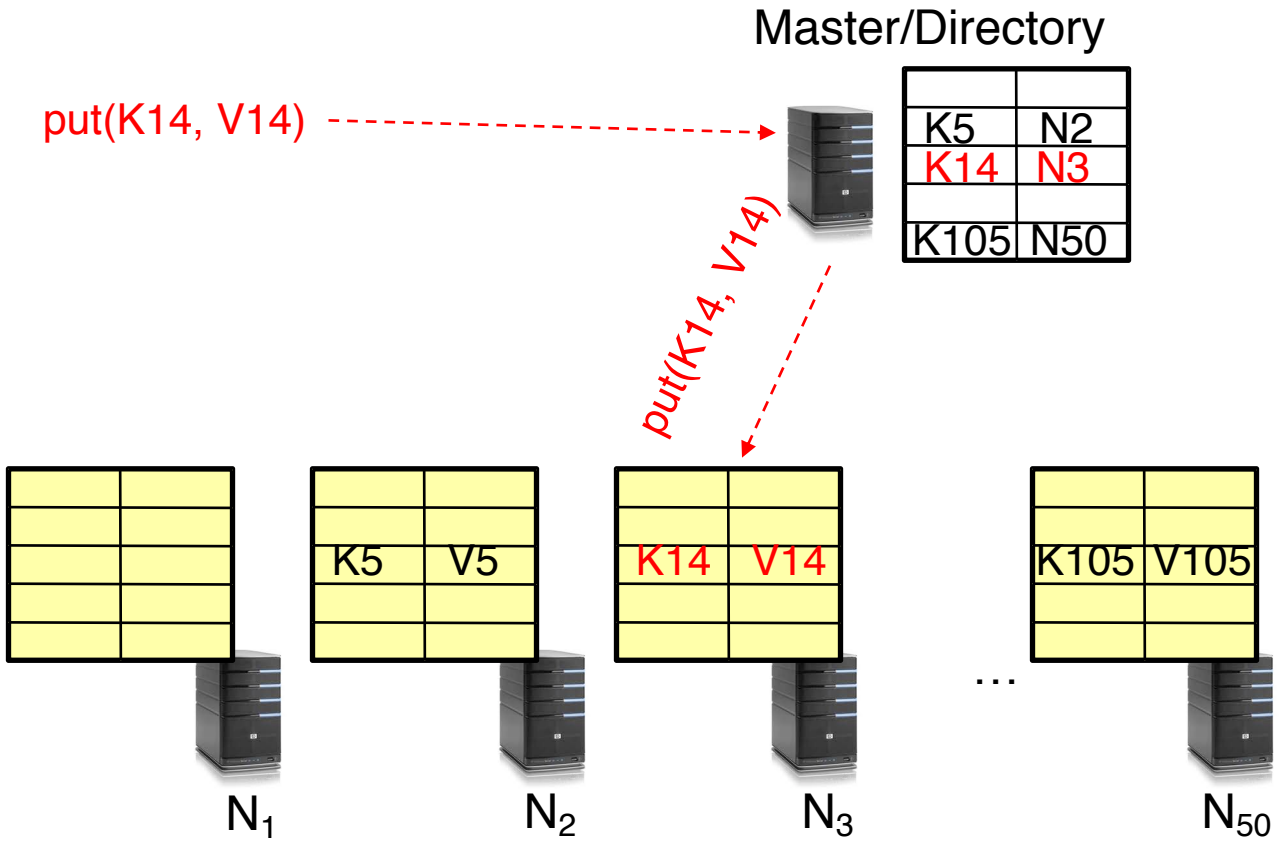
- And, do the above while providing
  - Fault Tolerance
  - Scalability
  - Consistency

How to solve the "where?"

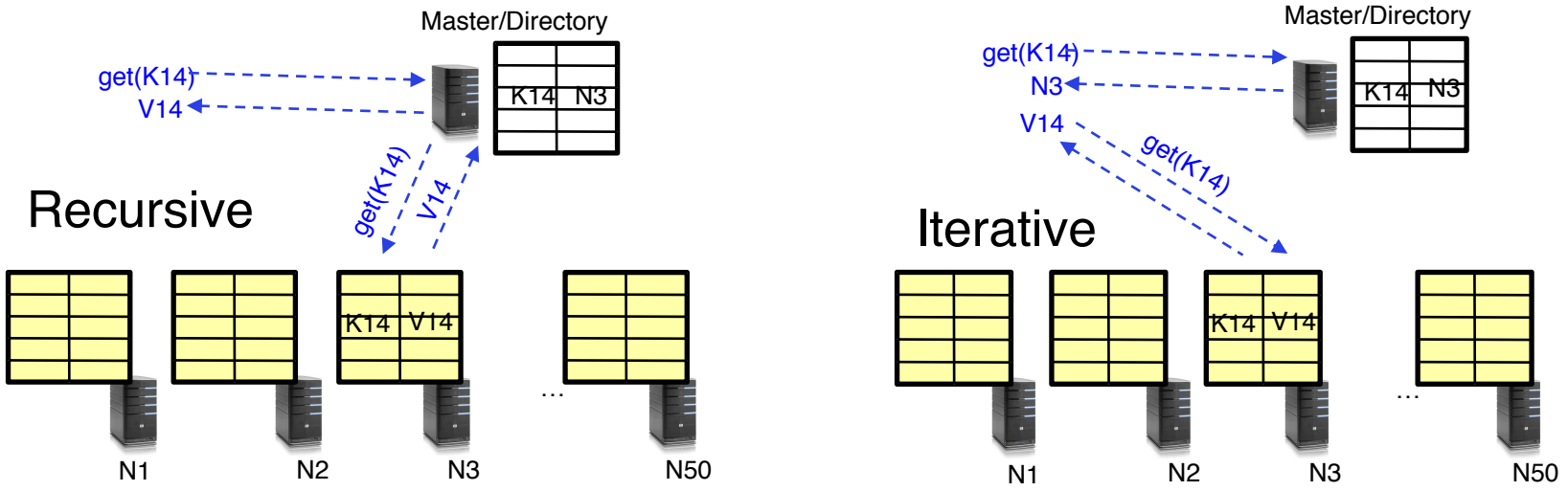
- Hashing
  - But what if you don't know who are all the nodes that are participating?
  - Perhaps they come and go …
  - What if some keys are really popular?
- Lookup
  - Hmm, won't this be a bottleneck and single point of failure?

Directory-Based Architecture

Have a node maintain the mapping between **keys** and the **machines (nodes)** that store the **values** associated with the **keys**



Iterative vs. Recursive Query



- Recursive Query: Directory Server Delegates
- Iterative Query: Client Delegates

Iterative vs Recursive Query

Recursive

+ Faster, as directory server is typically close to storage nodes
+ Easier for consistency: directory can enforce an order for all puts and gets
- Directory is a performance bottleneck

Iterative

+ More scalable, clients do more work
- Harder to enforce consistency

Scalability: How easy is it to make the system bigger?

- Storage: Use more nodes
- Number of Requests
  - Can serve requests from all nodes on which a value is stored in parallel
  - Master can replicate a popular item on more nodes
- Master/Directory Scalability
  - Replicate It (multiple identical copies)
  - Partition it, so different keys are served by different directories

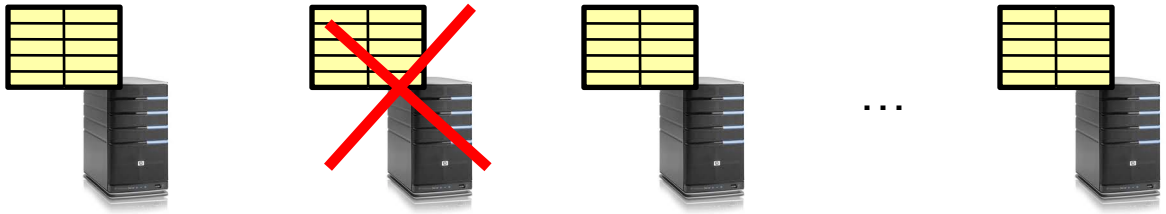
Scalability: Load Balancing

- Directory tracks available storage at each node
  - Prefer to insert at nodes with more storage available
- What happens when a new node is added?
  - Cannot insert only new values at new node
  - Move values from heavily loaded nodes to new node
- What happens when a node fails?
  - Replicate values from failed node to other nodes

Scaling Up Directory

- Directory contains number of entries equal to number of key/value pairs in entire system
  - Could be tens or hundreds of billions of pairs
- Solution: **Consistent Hashing**
  - Assign each node a unique ID in [0..2^m-1]
  - Assume we can hash keys to same range of IDs
  - Each (key,value) stored at node with smallest ID larger than hash(key)
- Important property: Adding a new bucket doesn't require moving lots of existing values to new buckets

Challenges



- **Fault Tolerance:** handle machine failures without losing data and without degradation in performance

- **Scalability:**
  - Need to scale to thousands of machines
  - Need to allow easy addition of new machines

Key to Node Mapping Example

Partitioning example with
m = 6 → ID space: 0..63

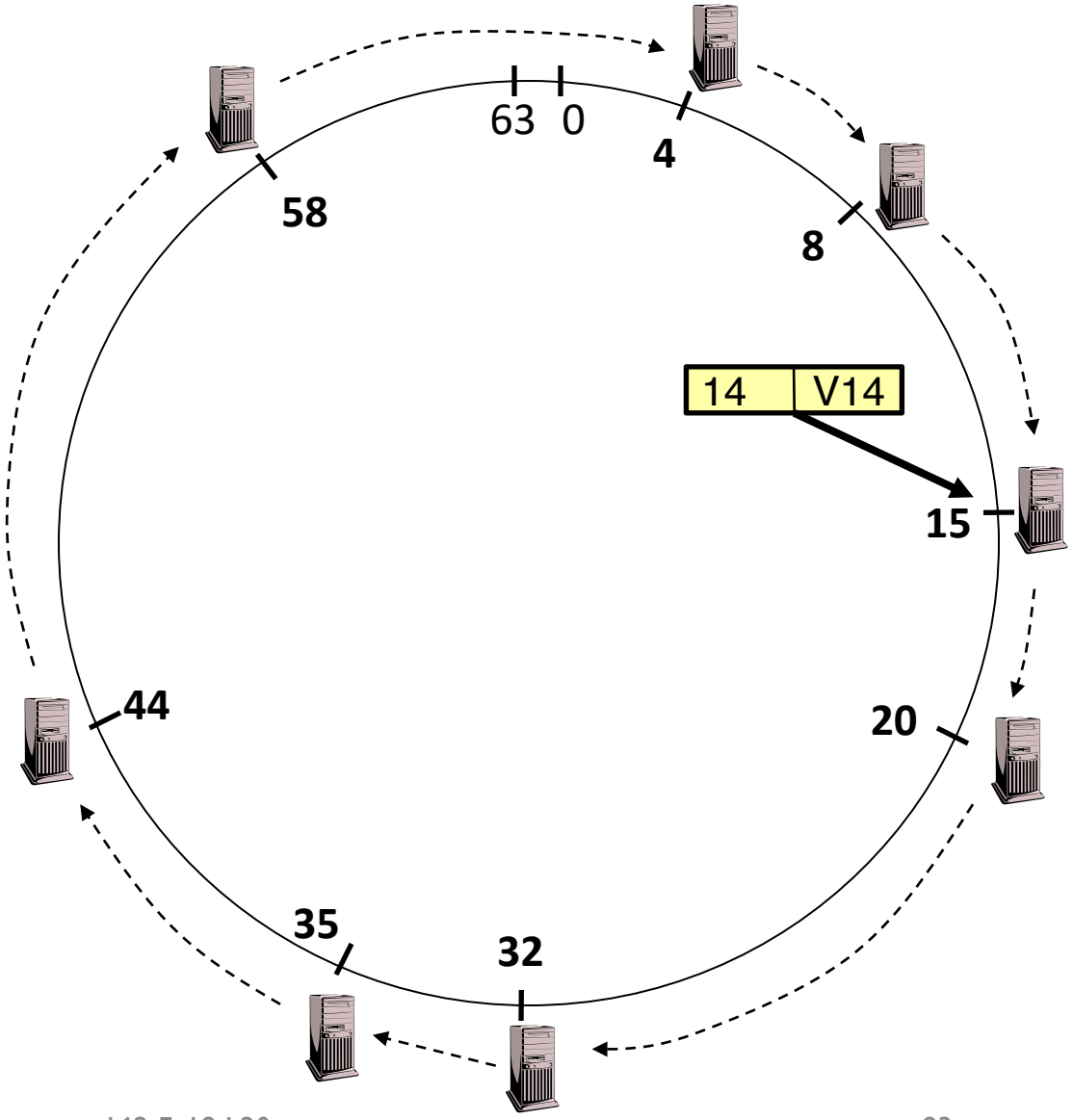
Node 8 maps keys [5,8]

Node 15 maps keys [9,15]

Node 20 maps keys [16, 20]

…

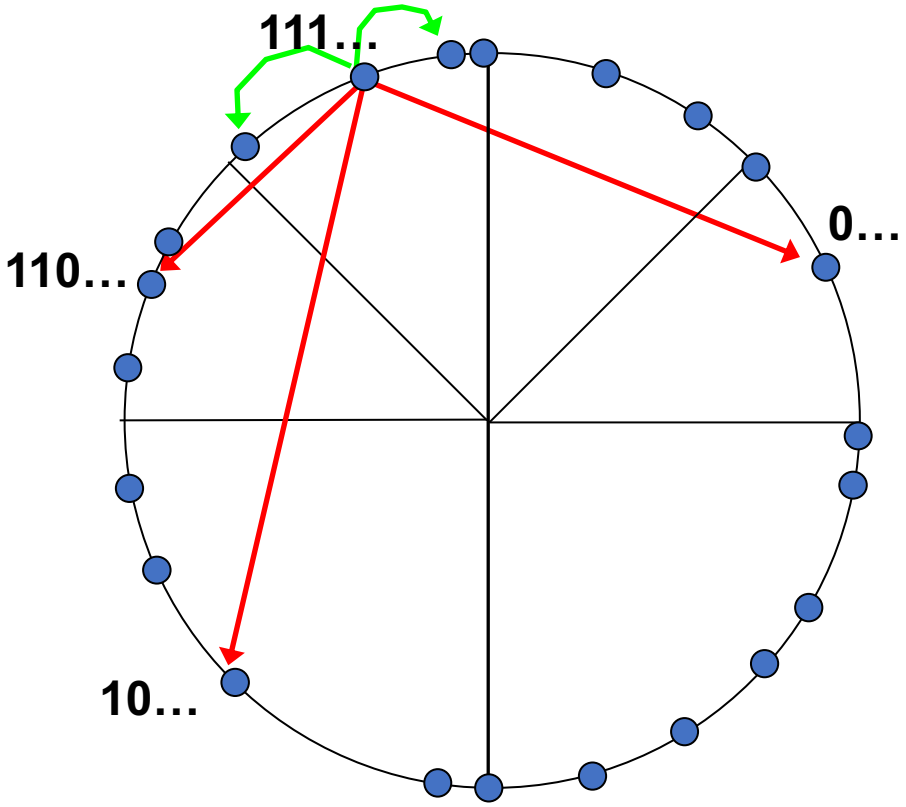
Node 4 maps keys [59, 4]



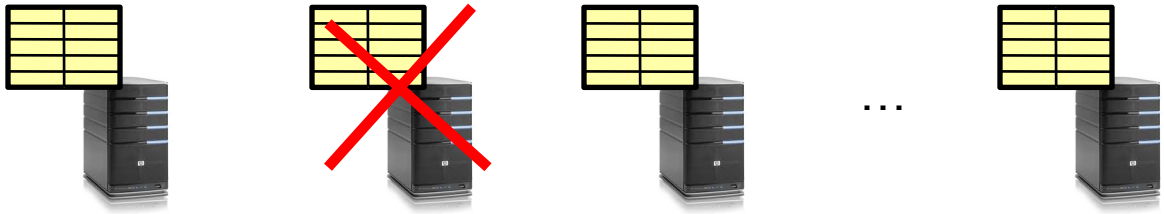
Performing a Lookup

- Fully decentralized
  - Any node can act as a directory for clients
  - Still works if a node leaves the network
- Each node knows about its successor and predecessor in the "circle"
  - All that is strictly needed for correctness
- Faster lookups: Each node maintains a routing table, allows client to get closer to destination in one hop

Example: Chord



Challenges

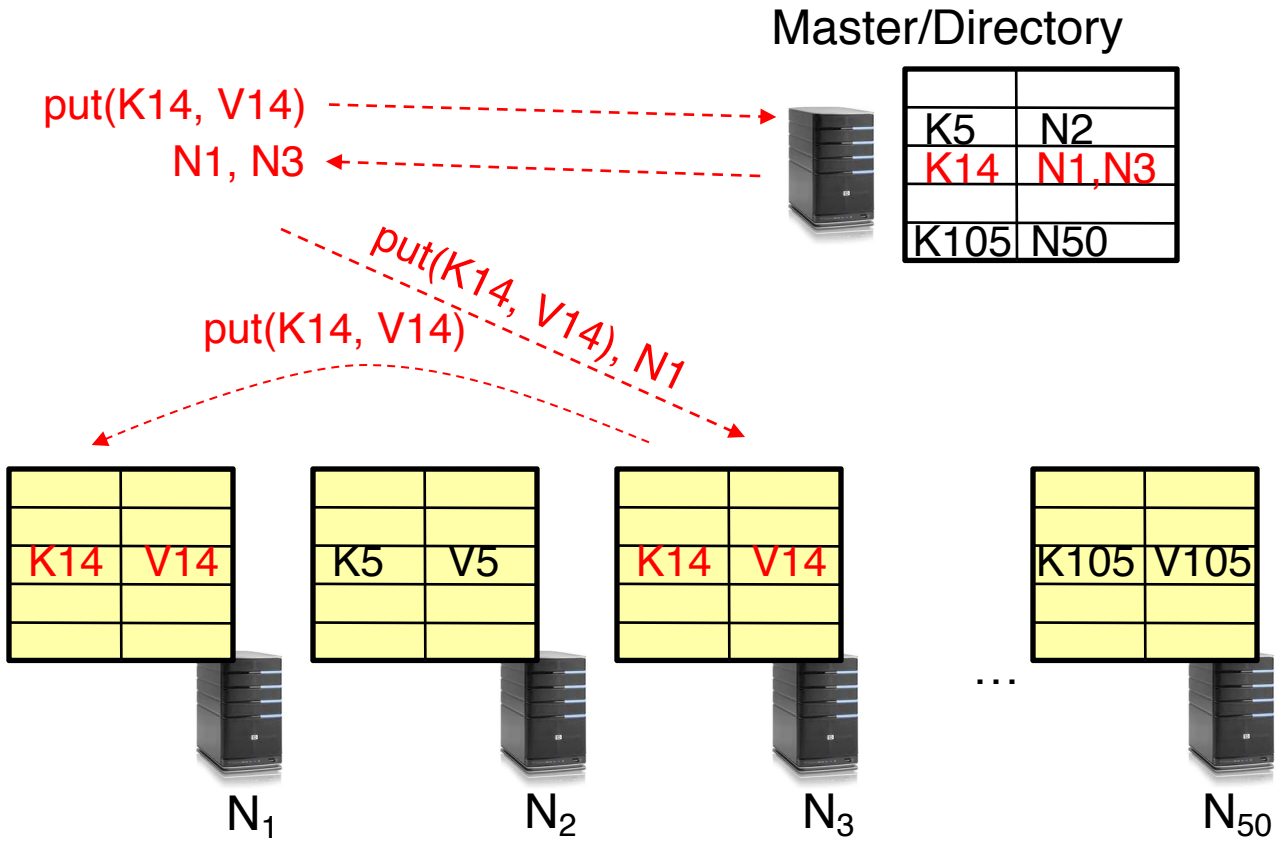


- **Consistency:** maintain data consistency in face of node failures and message losses

- **Heterogeneity** (if deployed as peer-to-peer systems):
  - Latency: 1ms to 1000ms
  - Bandwidth: 32 Kb/s to 1 Gb/s

Fault Tolerance

- Replicate value on several nodes
- Usually, place replicas on different racks in a datacenter to guard against rack failures



Consistency

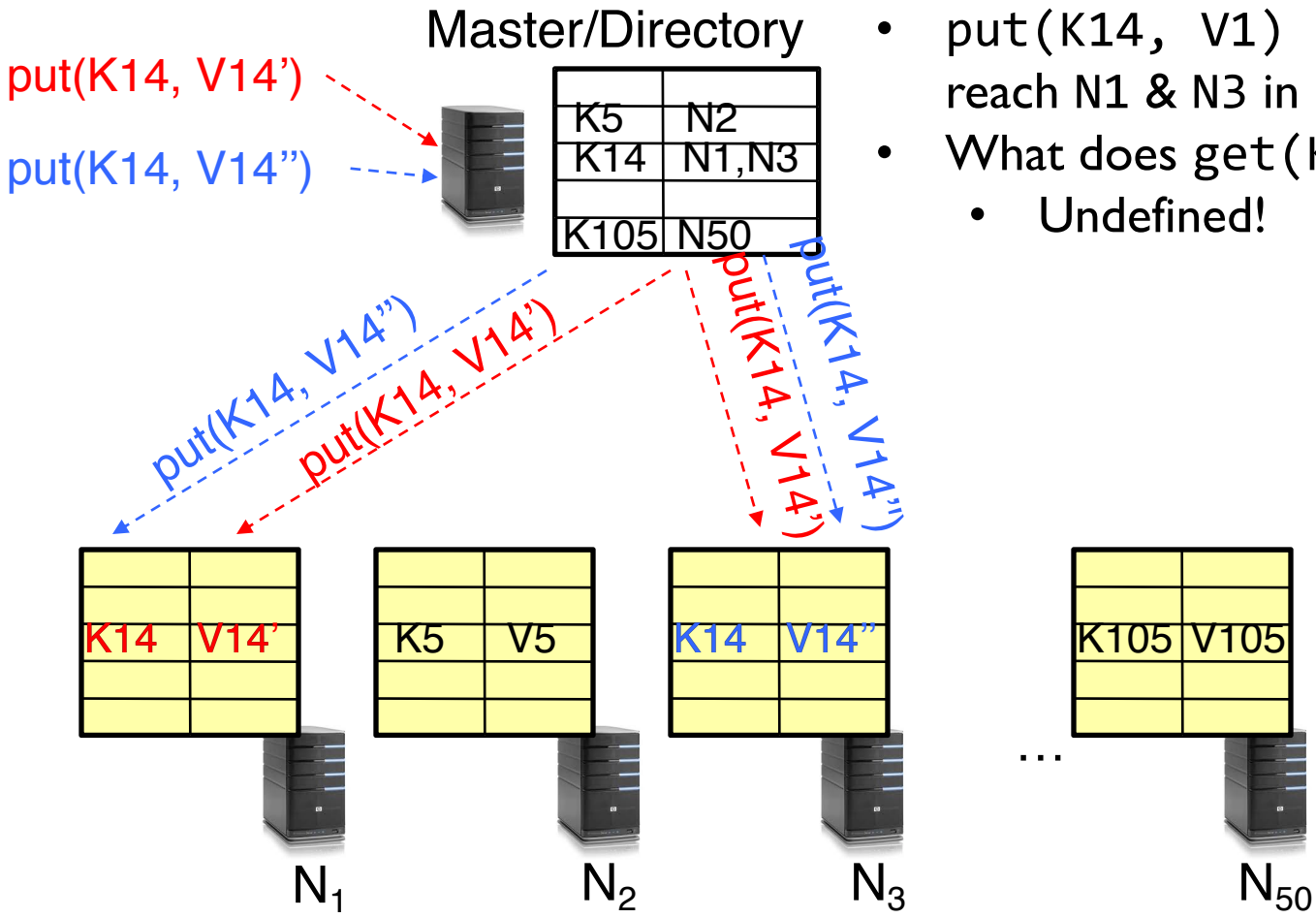
- Need to make sure a value is replicated correctly
- How do you know a value is replicated on every expected node?
- Wait for acknowledgements from all expected nodes

Consistency

- What happens if a node fails during replication?
  - Pick another node and try again
- What happens if a node is slow?
  - Slow down entire `put`? Pick another node?
- In general with multiple replicas: slow `put` and fast `get` operations

Consistency

- If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order



- `put(K14, V1)` and `put(K14, V2)` reach N1 & N3 in reverse order
- What does `get(K14)` return?
  - Undefined!

The Promise of Dist. Systems

- **Availability:** One machine goes down, overall system stays up
- **Durability:** One machine loses data, but *system* does not lose anything
- **Security:** Easier to secure each component of the system individually?

Dist. System – the darker side

- **Availability:** Failure in one machine causes others to hang waiting for it
  - **Two sides of Fate sharing**
- **Durability:** Lots of machine might lose your data
- **Security:** More components means more points of attack

- Engineering of distributed systems – both cloud and end hosts – are fundamentally more reliable than in the 80's and 90's when the approach emerged

Parallel vs Distributed

- Distributed: different machines responsible for different parts of task
  - Usually no centralized state
  - Usually about different responsibilities or redundancy
- Parallel: different parts of same task performed on different machines
  - Usually about performance

Summary

- Distributed File Systems: Transparent access to files located on remote disks
  - Caching for performance
  - Blocks or whole files
  - Introduces consistency issues
    - File save vs write
    - Remote vs actively shared
  - NFS: Check periodically for changes to server copy
  - AFS: Server notifies client of changes
- Key Value Store: Simple put and get operations
  - Fault tolerance: replication
  - Scalability: Add nodes, balance load, no central directory
  - Consistency: Quorum consensus for better performance

Summary

- Consensus Goal: Everyone agrees on the state of the distributed system
  - Doesn't depend who you ask
  - Doesn't matter if nodes go down

- Distributed Transactions ???:
  - Atomic, can't revert once agreement is reached