CS162
Operating Systems and Systems Programming
Lecture 21

Reliability & Distributed Systems

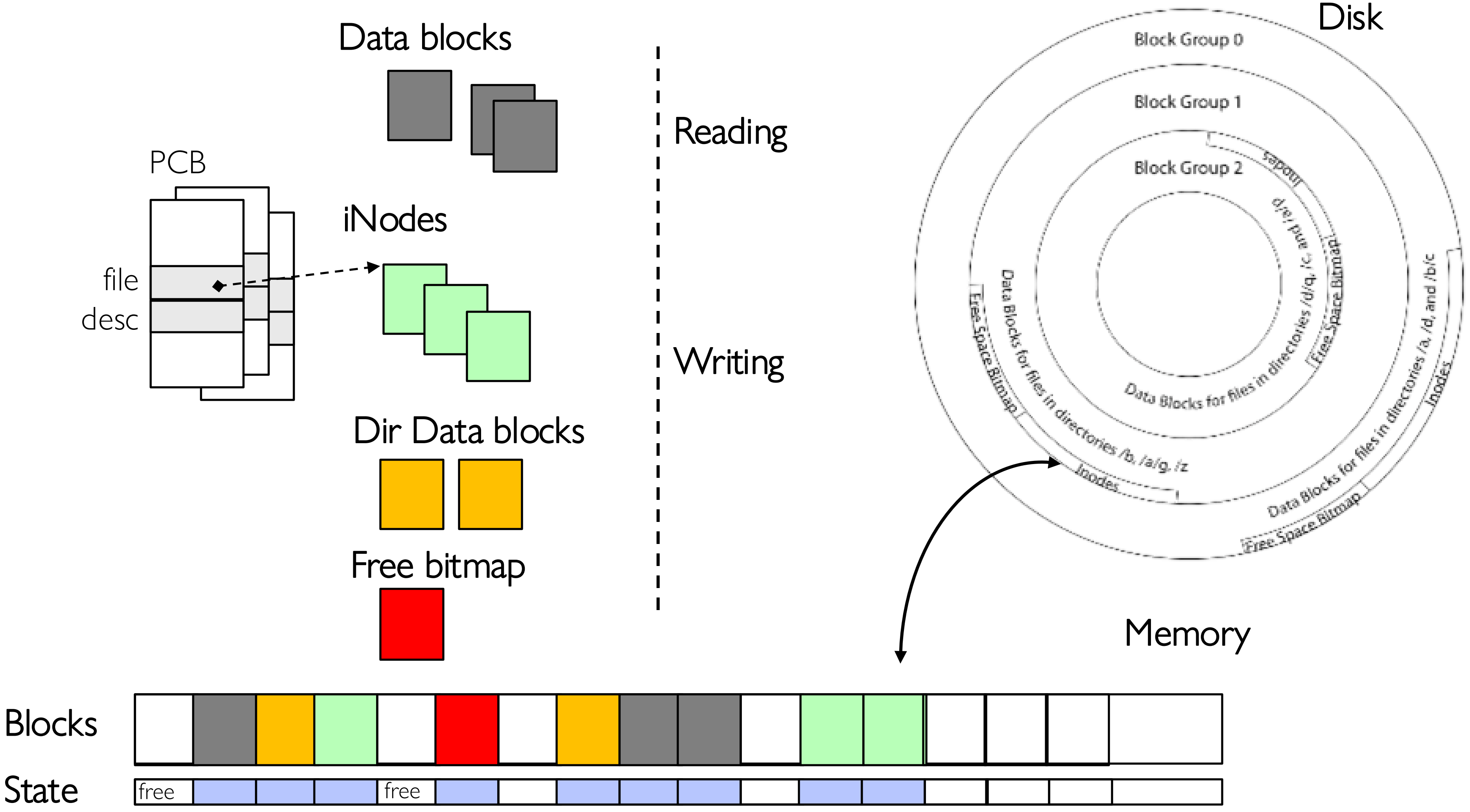
Professor Natacha Crooks

https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, Alison Norman and Lorenzo Alvisi

Recall: File System Buffer Cache

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



Recall: Storage Reliability Problem

Blocks written to buffer cache are flushed to disk periodically, but may be lost if your machine crashes before that

Even without a buffer cache, file system data structures may be inconsistent after a failure

- Single logical file operation can involve updates to multiple disk blocks (inode, indirect block, data block, bitmap, …)
- Within the storage device, single block write might update multiple sectors

How do we guarantee consistency regardless of when a crash occurs?

Recall: Boom!


HANDLED*** Address 8016a950 has base at 80100000
.6.2 irql:1f SYSVER 0xf0000565
Name
ntoskrnl.exe
atapi.sys
Disk.sys
Ntfs.sys
NTice.sys
Cdrom.SYS
KSecDD.SYS
win32k.sys
Cdfs.SYS
nbf.sys
netbt.sys
afd.sys
Parport.SYS
Dll Base DateStmp - Name
80010000
80007000
801db000
80237000
f1f48000
f228c000
f2290000
fe0c2000
fdca2000
fdc35000
f1f68000
f2008000

Recall: Two Reliability Approaches

Careful Ordering and Recovery

FAT & FFS + (fsck)

Each step builds structure,

Data block ⇐ inode ⇐ free ⇐ directory

Last step links it in to rest of FS

Recover scans structure looking for incomplete actions

Versioning and Copy-on-Write

ZFS, …

Version files at some granularity

Create new structure linking back to unchanged parts of old

Last step is to declare that the new version is ready

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)

– Many applications use temporary files and atomic rename

Key Concept: Transaction

A *transaction* is an atomic sequence of reads and writes that takes the system from one 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: Transactions in SQL

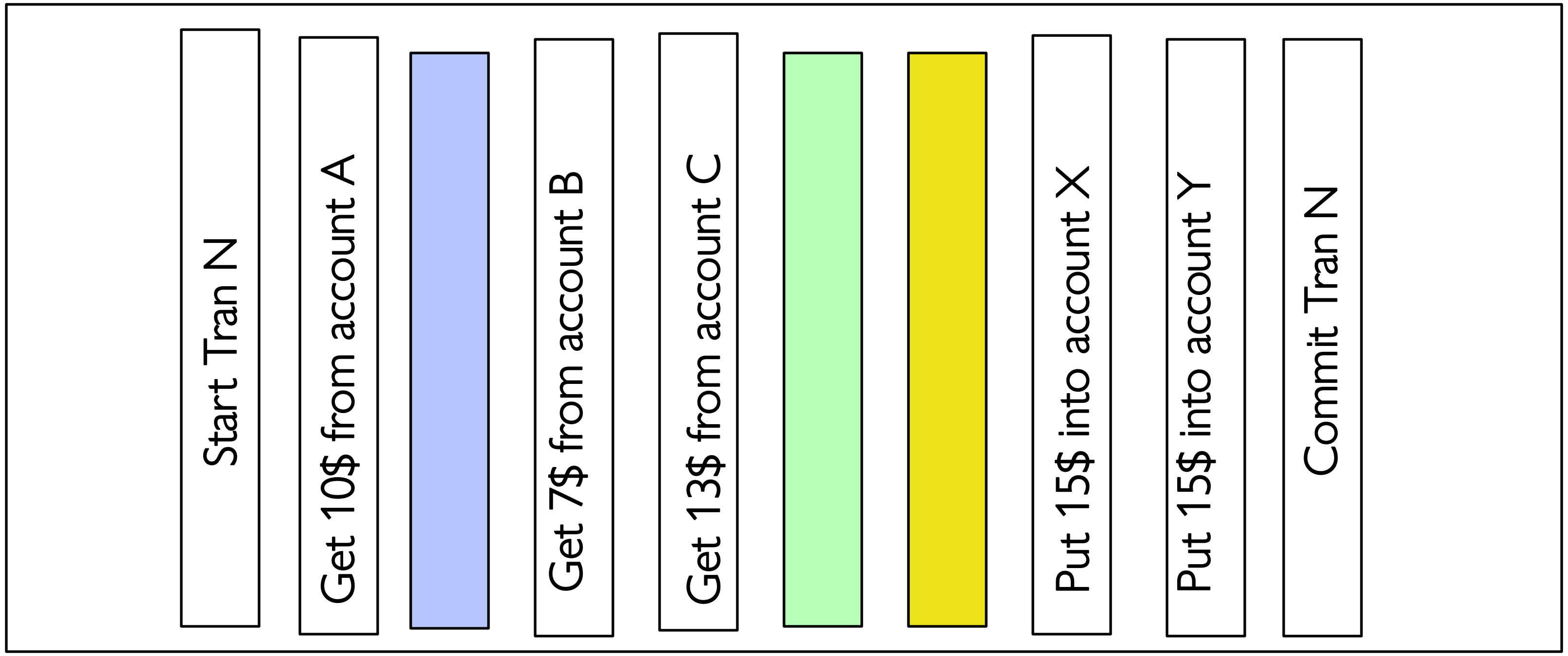
```
BEGIN;    --BEGIN TRANSACTION
 UPDATE accounts SET balance = balance - 100.00
   WHERE name = 'Alice';

 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

Useful Tool: 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 to all FS data structures are treated as transactions
  - A transaction is committed once it is fully written to the log
    - » Data forced to disk for reliability
    - » Process can be accelerated with NVRAM
  - Although the actual file system data structures may not be updated immediately, data is preserved in the log and replayed to recover
- Difference between "Log Structured" and "Journaling" file systems
  - In a Log Structured filesystem, data stays in log form
  - In a Journaling filesystem, log is only 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 specific blocks for it)

Once changes are in the log, they can be safely applied to other data structures

- E.g. modify inode pointers and directory entries

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

Creating a File (No Journaling Yet)

Find free data block(s)

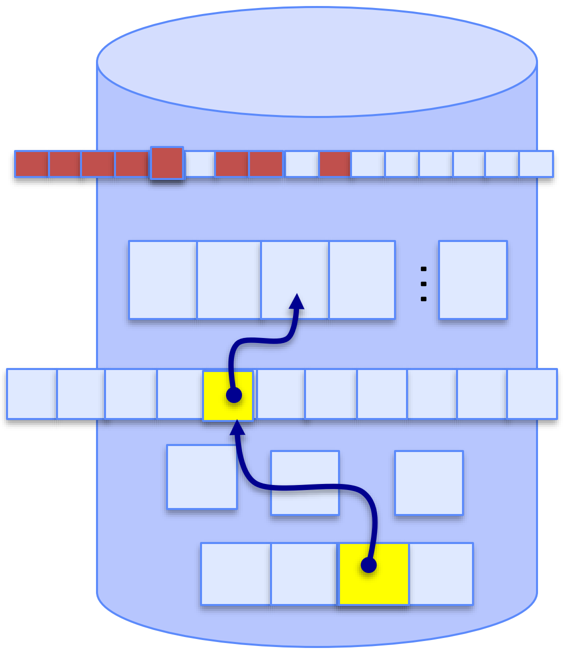
Find free inode entry

Find dirent insertion point

Write bitmap (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)

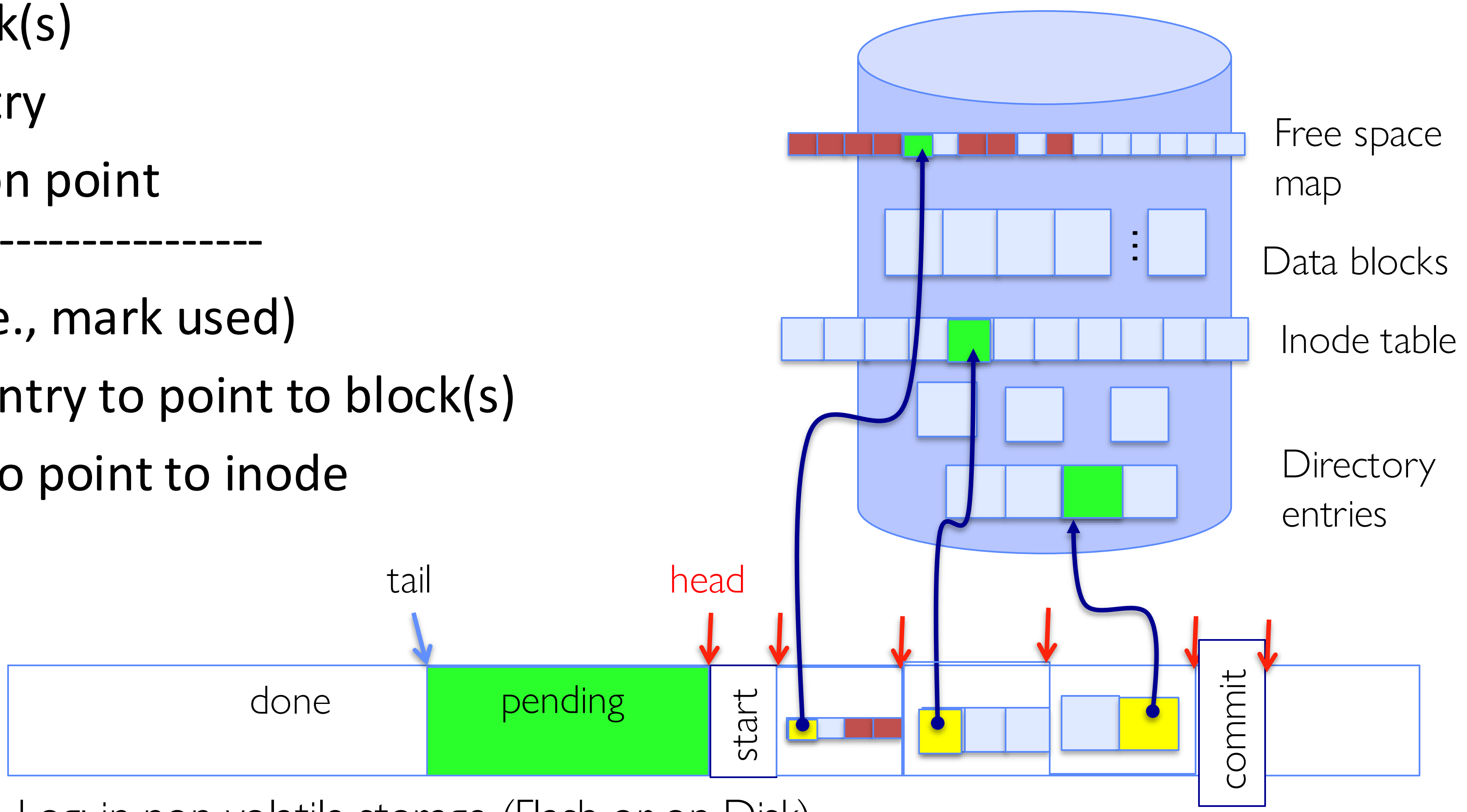
Find free inode entry

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



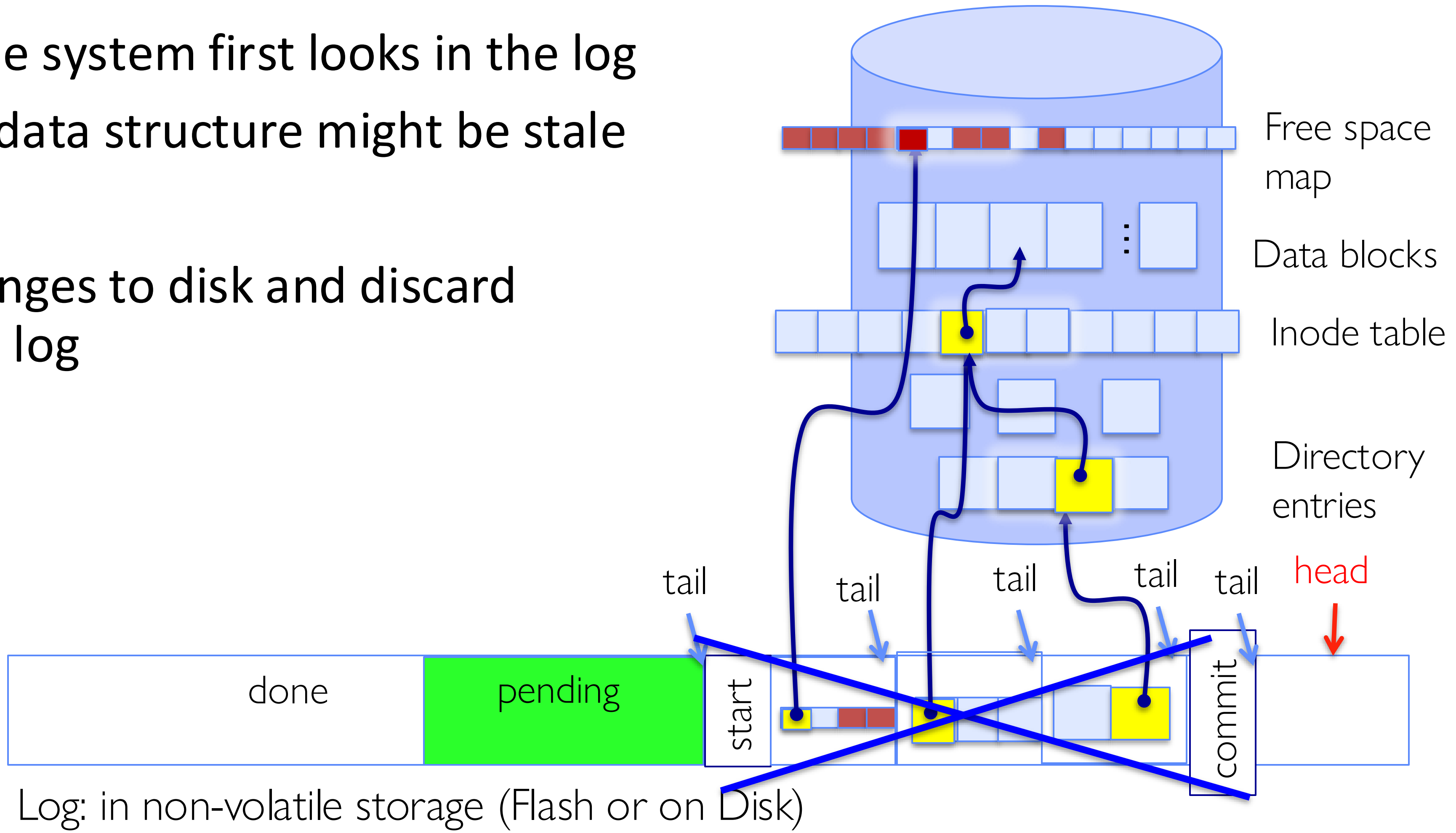
Log: in non-volatile storage (Flash or on Disk)

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



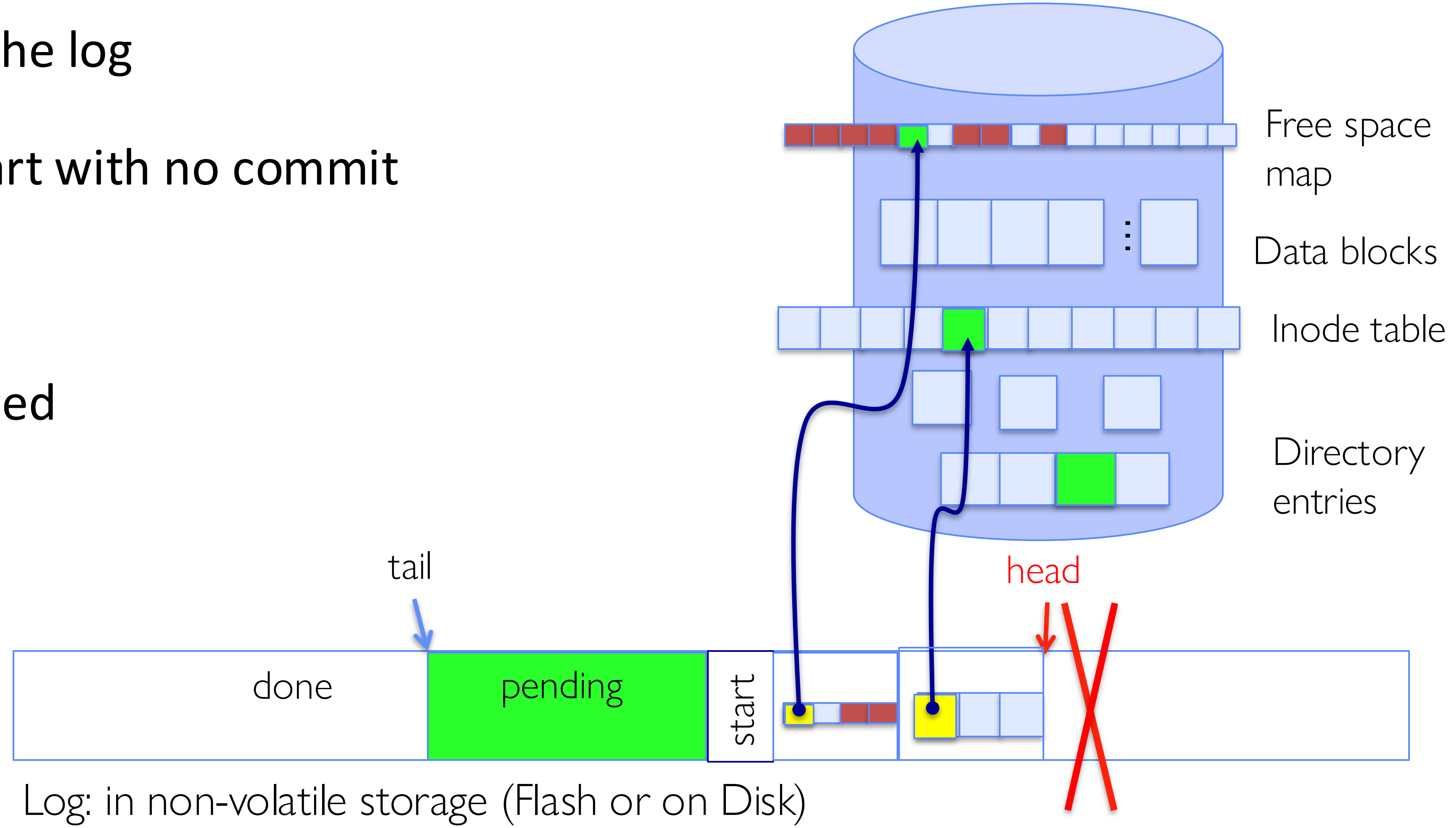
Crash Recovery: Discard Partial Transactions

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)

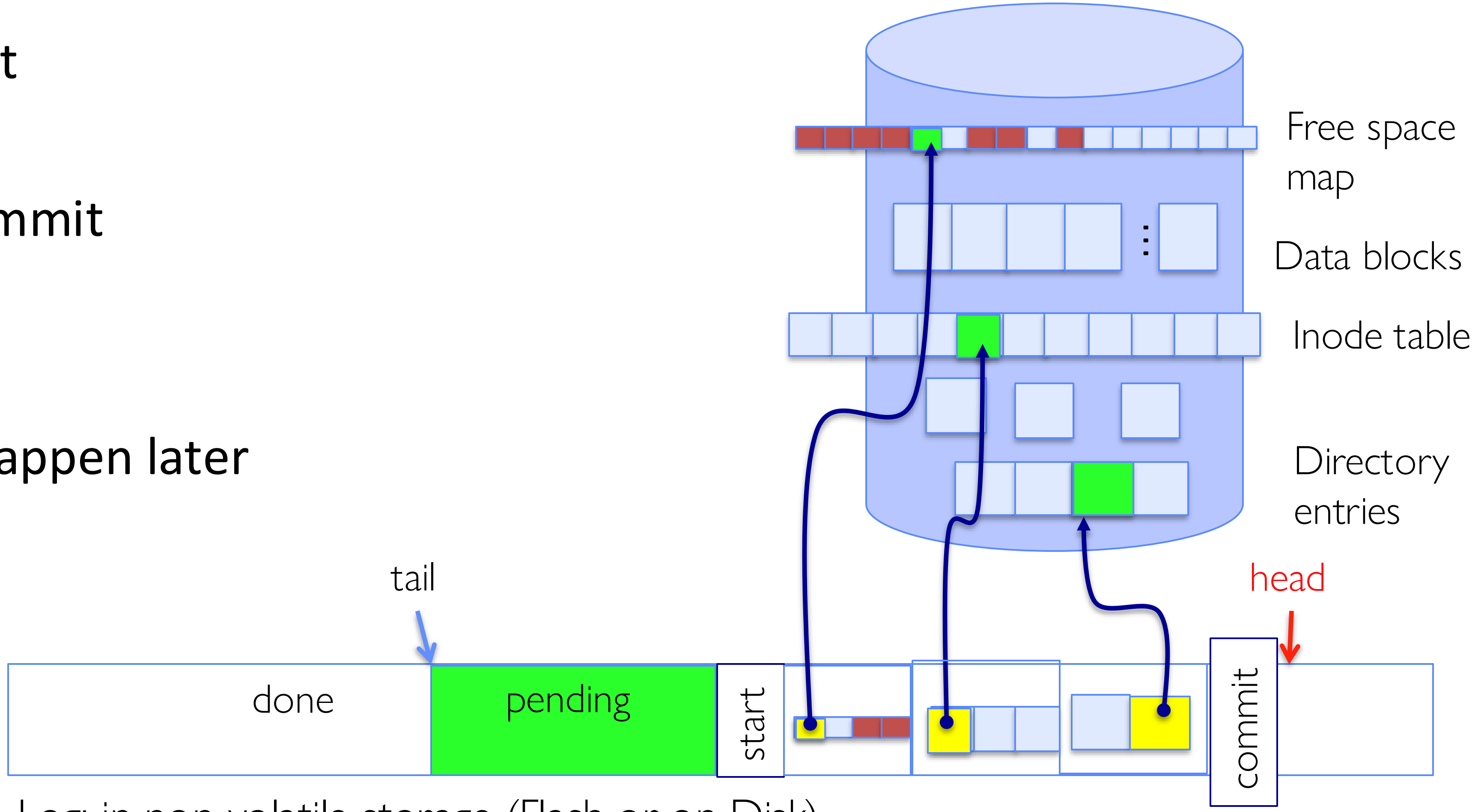
Crash Recovery: Keep Complete Transactions

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

Topic Breakdown

- Virtualizing the CPU
  - Process Abstraction and API
  - Threads and Concurrency
  - Scheduling
- Virtualizing Memory
  - Virtual Memory
  - Paging
- Persistence
  - IO devices
  - File Systems
- Distributed Systems
  - Challenges with distribution
  - Data Processing & Storage

Topic Breakdown

Virtualizing the CPU	Process Abstraction and API Threads and Concurrency Scheduling
Virtualizing Memory	Virtual Memory Paging
Persistence	IO devices File Systems
Distributed Systems	Challenges with distribution Data Processing & Storage

What is a Distributed System?

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

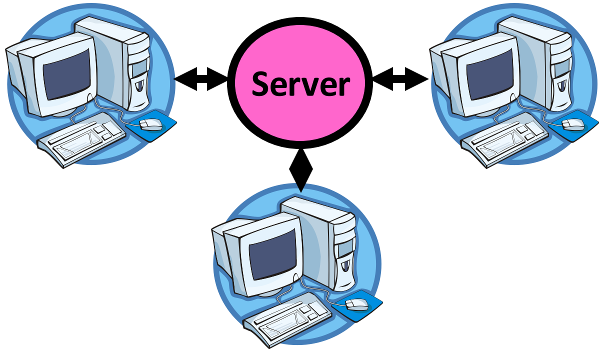
Leslie Lamport

Centralized vs Distributed Systems

So far, considered a single computer!
All computation was done on the local computer in isolation

The world is a large distributed system

Two types of distributed systems

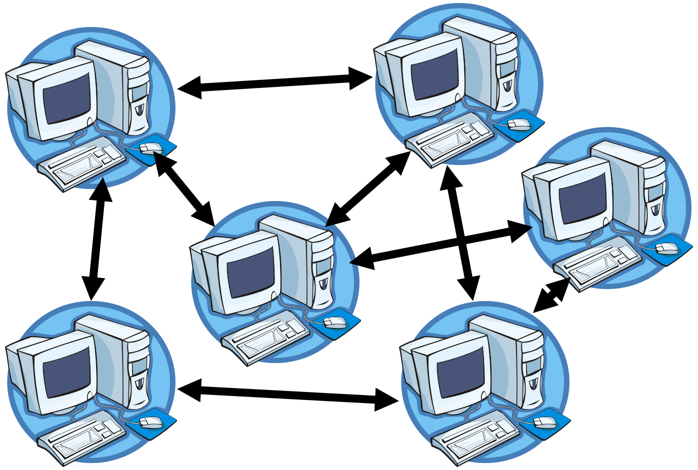


Client/Server Model

One or more server provides *services* to clients

Clients make *remote procedure calls* to server

Server serves *requests* from clients



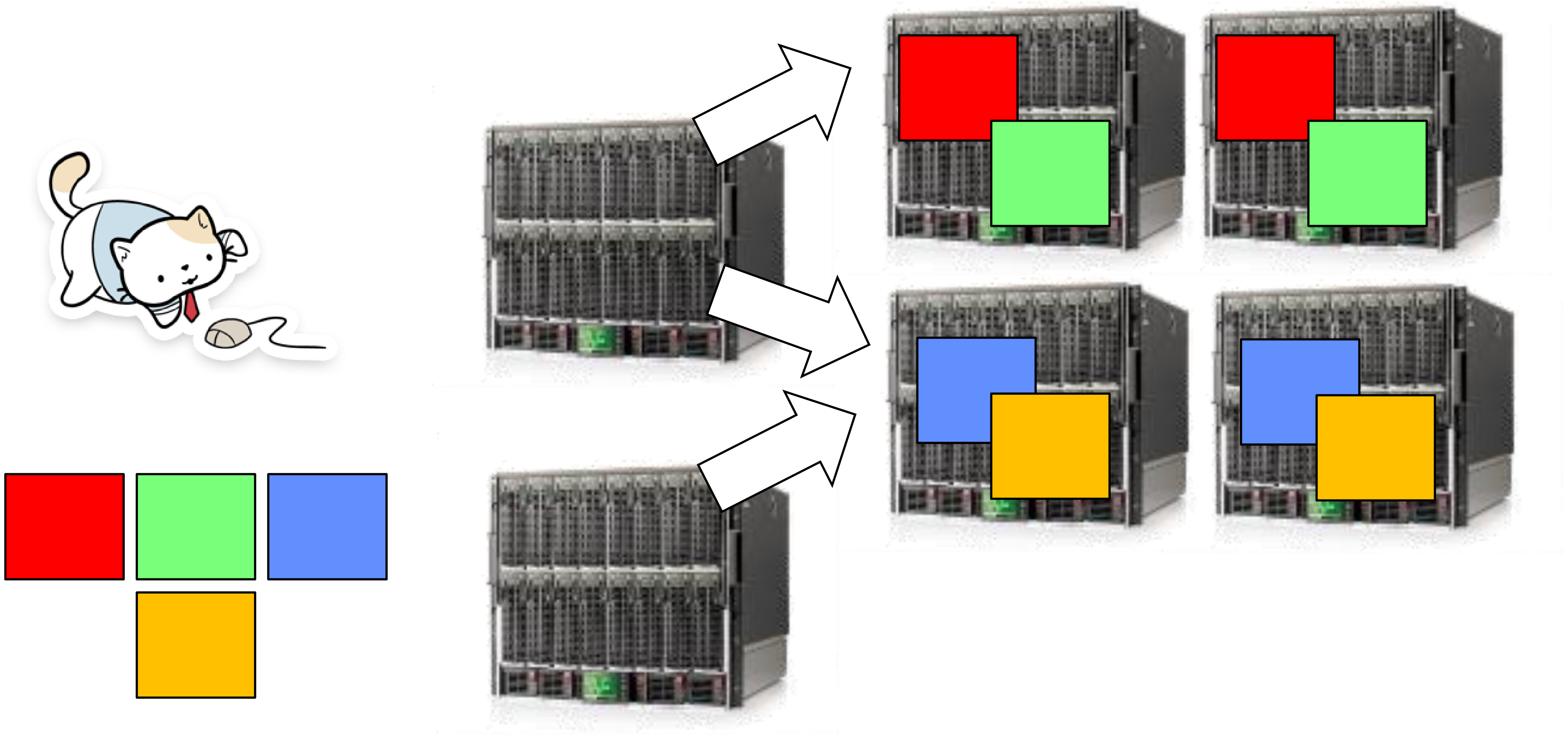
Peer-to-Peer Model

Each computer acts as a peer

No hierarchy or central point of coordination

All-way communication between peers through *gossiping*

Example: How do I store all my data?



The promise of distributed systems

Availability

Proportion of time system is in functioning condition

=> One machine goes down, use another

Fault-tolerance

System has well-defined behavior when fault occurs

=> Store data in multiple locations

Scalability

Ability to add resources to system to support more work

⇒Just add machines when need more storage/processing power

Transparency

The ability of the system to mask its complexity behind a simple interface

Transparency

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

The challenges of distributed systems

How do you get machines to communicate?

How do you get machines to coordinate?

How do you deal with failures?

How do you deal with security (corrupted machines)?

Topic Roadmap

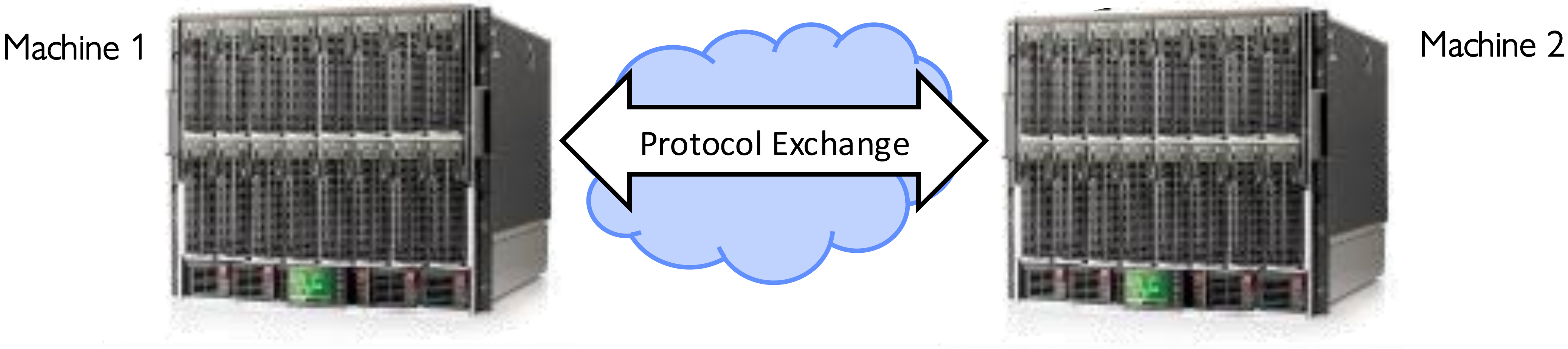
Distributed File System: NFS

Peer-To-Peer System: The Internet

Distributed Data Processing
(MapReduce and Spark)

Coordination
(Atomic Commit and Consensus)

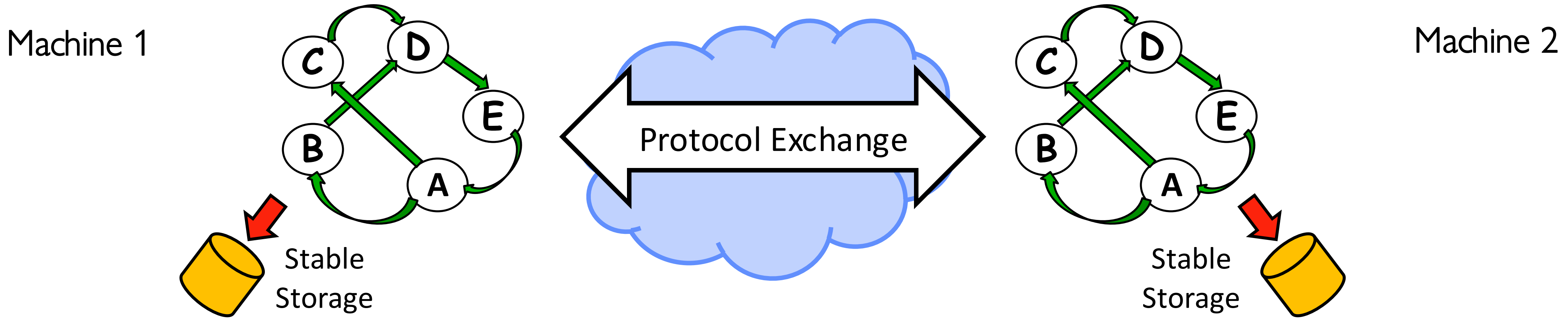
How do machines communicate?



A protocol is an agreement on how to communicate, covering

- 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

How do machines communicate?



A protocol is an agreement on how to communicate, covering

- 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

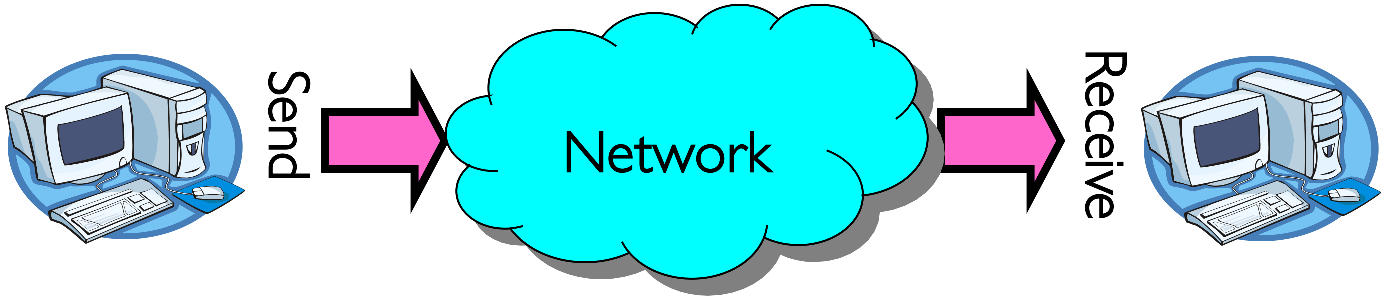
Examples of Protocols in Human Interactions

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 Matei…."
   Or: "Hi, it's me" (← what's *that* about?)
7. Caller: "Hey, do you think … blah blah blah …" **pause**
8. Callee: "Yeah, blah blah blah …" **pause**
9. Caller: Bye
10. Callee: Bye
11. Hang up

Message Passing

How do you actually program a distributed application?



Interface:

- Mailbox (mbox): temporary holding area for messages
- Send(message,mbox)
- Receive(buffer,mbox)

Question: Data Representation

An object in memory has a machine-specific binary representation

Without shared memory, externalizing an object requires us to turn it into a sequential sequence of bytes

– Serialization/Marshalling: Express an object as a sequence of bytes

– Deserialization/Unmarshalling: Reconstructing the original object from its marshalled form at destination

Simple Data Types

```
uint32_t x;
```

Suppose I want to write a x to a file

First, open the file: `FILE* f = fopen(“foo.txt”, “w”);`

Then, I have two choices:

1. `fprintf(f, “%lu”, x);`
2. `fwrite(&x, sizeof(uint32_t), 1, f);`

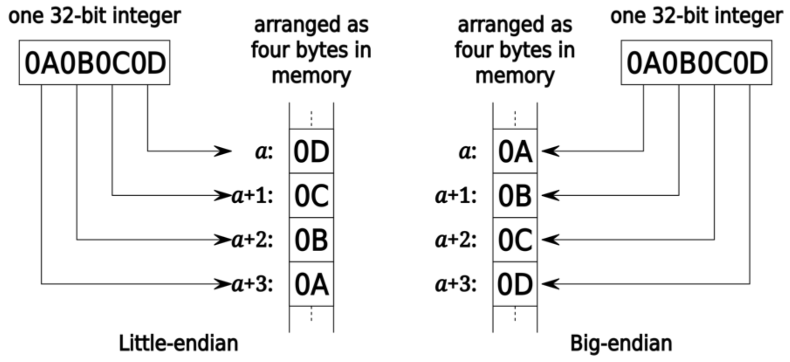
Neither one is “wrong” but sender and receiver should be consistent!

Machine Representation: Endianness

What order are the bytes of a multi-byte integer stored in physical memory?

Big Endian: most significant byte is stored first

Little Endian: least significant byte is stored first




```
int val = 0x12345678;
int i;
printf("val = %x\n", val);
for (i = 0; i < sizeof(val); i++) {
  printf("val[%d] = %x\n", i, ((uint8_t *) &val)[i]);
}
```

```
(base) CullerMac19:code09 culler$ ./endian
val = 12345678
val[0] = 78
val[1] = 56
val[2] = 34
val[3] = 12
```

Processor	Endianness
Motorola 68000	Big Endian
PowerPC (PPC)	Big Endian
Sun Sparc	Big Endian
IBM S/390	Big Endian
Intel x86 (32 bit)	Little Endian
Intel x86_64 (64 bit)	Little Endian
Dec VAX	Little Endian
Alpha	Bi (Big/Little) Endian
ARM	Bi (Big/Little) Endian
IA-64 (64 bit)	Bi (Big/Little) Endian
MIPS	Bi (Big/Little) Endian

What About Richer Objects?

```
typedef struct word_count
{
  char *word;
  int count;
  struct word_count *next;
}
word_count_t;
```

Consider `word_count_t` of Homework 0 and 1 …

Each element contains:

- An `int`
- A *pointer* to a string (of some length)
- A *pointer* to the next element

`fprintf_words` writes these as a sequence of lines (character strings with `\n`) to a file stream

What if you wanted to write the whole list as a binary object (and read it back as one)?

- How do you represent the string?
- Does it make any sense to write the pointer?

Data Serialization Formats

JSON, XML and Protocol Buffers are commonly used in web applications

Lots of ad-hoc formats


```
{ “faculty”:
      [
            { “id”: 1,
              “firstname”: “Natacha”,
              “lastname”: “Crooks”
            },
            { “id”: 2,
              “firstname”: “Matei”,
              “lastname”: “Zaharia”
            }
      ]
}
```

Data Serialization Formats

Name	Creator-maintainer	Based on	Standardized?	Specification	Binary?	Human-readable?	Supports references?[e]	Schema-IDL?	Standard APIs	Supports [hide] Zero-copy operations
Apache Avro	Apache Software Foundation	N/A	No	Apache Avro™ 1.8.1 Specification	Yes	No	N/A	Yes (built-in)	N/A	N/A
Apache Parquet	Apache Software Foundation	N/A	No	Apache Parquet[1]	Yes	No	No	N/A	Java, Python	No
ASN.1	ISO, IEC, ITU-T	N/A	Yes	ISO/IEC 8824; X.680 series of ITU-T Recommendations	Yes (BER, DER, PER, OER, or custom via ECN)	Yes (XER, JER, GSER, or custom via ECN)	Partial[f]	Yes (built-in)	N/A	Yes (OER)
Bencode	Bram Cohen (creator) BitTorrent, Inc. (maintainer)	N/A	*De facto* standard via BitTorrent Enhancement Proposal (BEP)	Part of BitTorrent protocol specification	Partially (numbers and delimiters are ASCII)	No	No	No	No	N/A
Binn	Bernardo Ramos	N/A	No	Binn Specification	Yes	No	No	No	No	Yes
BSON	MongoDB	JSON	No	BSON Specification	Yes	No	No	No	No	N/A
CBOR	Carsten Bormann, P. Hoffman	JSON (loosely)	Yes	RFC 7049	Yes	No	Yes through tagging	Yes (CDDL)	No	Yes
Comma-separated values (CSV)	RFC author: Yakov Shafranovich	N/A	Partial (myriad informal variants used)	RFC 4180 (among others)	No	Yes	No	No	No	No
Common Data Representation (CDR)	Object Management Group	N/A	Yes	General Inter-ORB Protocol	Yes	No	Yes	Yes	ADA, C, C++, Java, Cobol, Lisp, Python, Ruby, Smalltalk	N/A
D-Bus Message Protocol	freedesktop.org	N/A	Yes	D-Bus Specification	Yes	No	No	Partial (Signature strings)	Yes (see D-Bus)	N/A
Efficient XML Interchange (EXI)	W3C	XML, Efficient XML	Yes	Efficient XML Interchange (EXI) Format 1.0	Yes	Yes (XML)	Yes (XPointer, XPath)	Yes (XML Schema)	Yes (DOM, SAX, StAX, XQuery, XPath)	N/A
FlatBuffers	Google	N/A	No	flatbuffers github page Specification	Yes	Yes (Apache Arrow)	Partial (internal to the buffer)	Yes [2]	C++, Java, C#, Go, Python, Rust, JavaScript, PHP, C, Dart, Lua, TypeScript	Yes
Fast Infoset	ISO, IEC, ITU-T	XML	Yes	ITU-T X.891 and ISO/IEC 24824-1:2007	Yes	No	Yes (XPointer, XPath)	Yes (XML schema)	Yes (DOM, SAX, XQuery, XPath)	N/A
FHIR	Health_Level_7	REST basics	Yes	Fast Healthcare Interoperability Resources	Yes	Yes	Yes	Yes	Hapi for FHIR[1] JSON, XML, Turtle	No
Ion	Amazon	JSON	No	The Amazon Ion Specification	Yes	Yes	No	No	No	N/A
Java serialization	Oracle Corporation	N/A	Yes	Java Object Serialization	Yes	No	Yes	No	Yes	N/A
JSON	Douglas Crockford	JavaScript syntax	Yes	STD 90/RFC 8259 (ancillary: RFC 6901, RFC 6902), ECMA-404, ISO/IEC 21778:2017	No, but see BSON, Smile, UBJSON	Yes	Yes (JSON Pointer (RFC 6901); alternately: JSONPath, JPath, JSPON, json:select()), JSON-LD	Partial (JSON Schema Proposal, ASN.1 with JER, Kwalify, Rx, Itemscript Schema), JSON-LD	Partial (Clarinet, JSONQuery, JSONPath), JSON-LD	No
MessagePack	Sadayuki Furuhashi	JSON (loosely)	No	MessagePack format specification	Yes	No	No	No	No	Yes
Netstrings	Dan Bernstein	N/A	No	netstrings.txt	Yes	Yes	No	No	No	Yes
OGDL	Rolf Veen	?	No	Specification	Yes (Binary Specification)	Yes	Yes (Path Specification)	Yes (Schema WD)		N/A
OPC-UA Binary	OPC Foundation	N/A	No	opcfoundation.org	Yes	No	Yes	No	No	N/A
OpenDDL	Eric Lengyel	C, PHP	No	OpenDDL.org	No	Yes	Yes	No	Yes (OpenDDL Library)	N/A
Pickle (Python)	Guido van Rossum	Python	*De facto* standard via Python Enhancement Proposals (PEPs)	[3] PEP 3154 -- Pickle protocol version 4	Yes	No	No	No	Yes ([4])	No
Property list	NeXT (creator) Apple (maintainer)	?	Partial	Public DTD for XML format	Yes[a]	Yes[b]	No	?	Cocoa, CoreFoundation, OpenStep, GnuStep	No
Protocol Buffers (protobuf)	Google	N/A	No	Developer Guide: Encoding	Yes	Partial[d]	No	Yes (built-in)	C++, C#, Java, Python, Javascript, Go	No
	John McCarthy					Yes				

Remote Procedure Call (RPC)

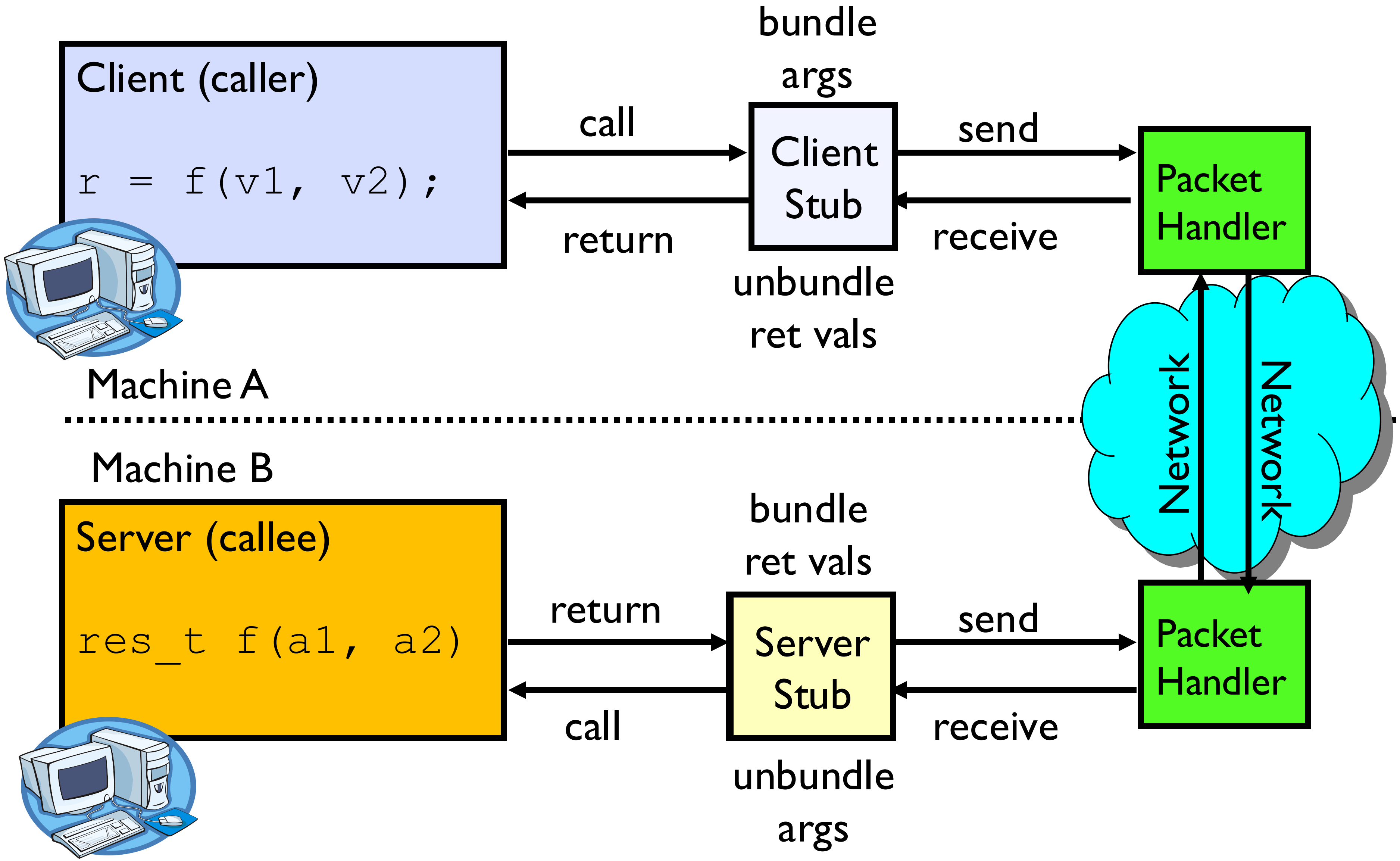
Raw messaging is a bit too low-level for programming

- Must wrap up information into message at source
- Must decide what to do with message at destination
- May need to sit and wait for multiple messages to arrive
- And must deal with machine representation by hand

Common abstraction: Remote Procedure Call (RPC)

- Calls a procedure on a remote machine
- Idea: Make communication look like an ordinary function call
- Automate all of the complexity of translating between representations
- Client calls:
  `remoteFileSystem→Read("rutabaga");`
- Translated automatically into call on server:
  `fileSys→Read("rutabaga");`

RPC Information Flow



RPC Implementation

Request-response message passing (under covers!)

"Stub" provides glue on client/server

- Client stub is responsible for "marshalling" arguments and "unmarshalling" the return values
- Server-side stub is responsible for "unmarshalling" arguments and "marshalling" the return values.

Marshalling involves (depending on system)

- Converting values to a canonical form, serializing objects, copying arguments passed by reference, etc.

RPC Details (1/3)

Equivalence with regular procedure call

- Parameters ⇔ Request Message
- Result ⇔ Reply message
- Name of Procedure: Passed in request message
- Return Address: mbox2 (client return mail box)

Stub generator: Compiler that generates stubs

- Input: interface definitions in an “interface definition language (IDL)”
  - » Contains, among other things, types of arguments/return
- Output: stub code in the appropriate source language
  - » Code for client to pack message, send it, wait for result, unpack result and return to caller
  - » Code for server to unpack message, call procedure, pack results, send them

RPC Details (2/3)

Cross-platform issues:

- What if client/server machines are different architectures/ languages?
  - » Convert everything to/from some canonical form
  - » Tag every item with an indication of how it is encoded (avoids unnecessary conversions)

How does client know which mbox (destination queue) to send to?

- Need to translate name of remote service into network endpoint (Remote machine, port, possibly other info)
- *Binding:* the process of converting a user-visible name into a network endpoint
  - » This is another word for “naming” at network level
  - » Static: fixed at compile time
  - » Dynamic: performed at runtime

RPC Details (3/3)

- Dynamic Binding
  - Most RPC systems use dynamic binding via name service
    - » Name service provides dynamic translation of service $\rightarrow$ mbox
  - Why dynamic binding?
    - » Access control: check who is permitted to access service
    - » Fail-over: If server fails, use a different one

- What if there are multiple servers?
  - Could give flexibility at binding time
    - » Choose unloaded server for each new client
  - Could provide same mbox (router level redirect)
    - » Choose unloaded server for each new request
    - » Only works if no state carried from one call to next

- What if multiple clients?
  - Pass pointer to client-specific return mbox in request

Problems with RPC: Non-Atomic Failures

Different failure modes in dist. system than on a single machine

Consider many different types of failures

- User-level bug causes address space to crash
- Machine failure, kernel bug causes all processes on same machine to fail
- Some machine is compromised by malicious party

Can easily result in inconsistent view of the world

- Did my cached data get written back or not?
- Did server do what I requested or not?

Answer? Distributed transactions/2PC

Problems with RPC: Performance

RPC is *not* performance transparent:

– Cost of Procedure call << same-machine RPC << network RPC

– Overheads: Marshalling, Stubs, Kernel-Crossing, Communication

Programmers must be aware that RPC is not free

– Caching can help, but may make failure handling even harder

Topic Roadmap

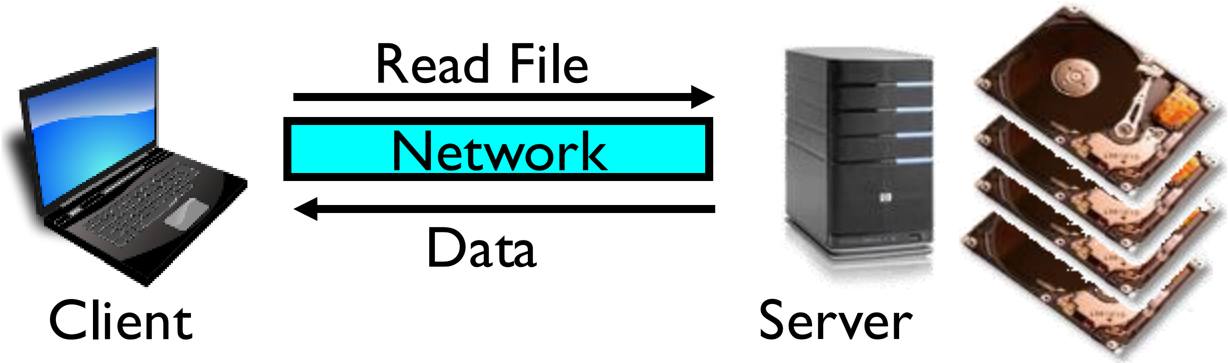
Distributed File System: NFS

Peer-To-Peer System: The Internet

Distributed Data Processing
(MapReduce and Spark)

Coordination
(Atomic Commit and Consensus)

Distributed File Systems



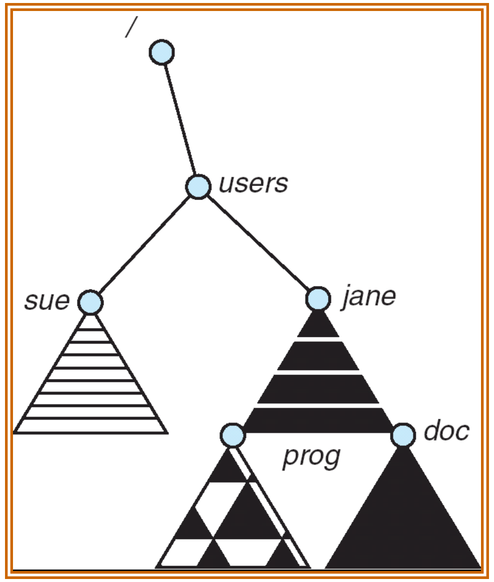
Transparent access to files stored on a remote disk

Mount remote files into your local file system

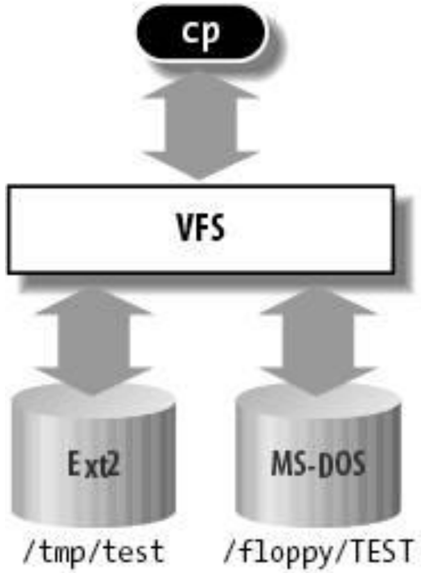
- Directory in local file system refers to remote files
- e.g., `/users/jane/prog/foo.c` on laptop actually refers to `/prog/foo.c` on `fs.cs.berkeley.edu`

Naming Choices:

- [Hostname,localname]: Filename includes server
- A global name space: Filename unique in "world"



Virtual Filesystem Switch (VFS)




```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

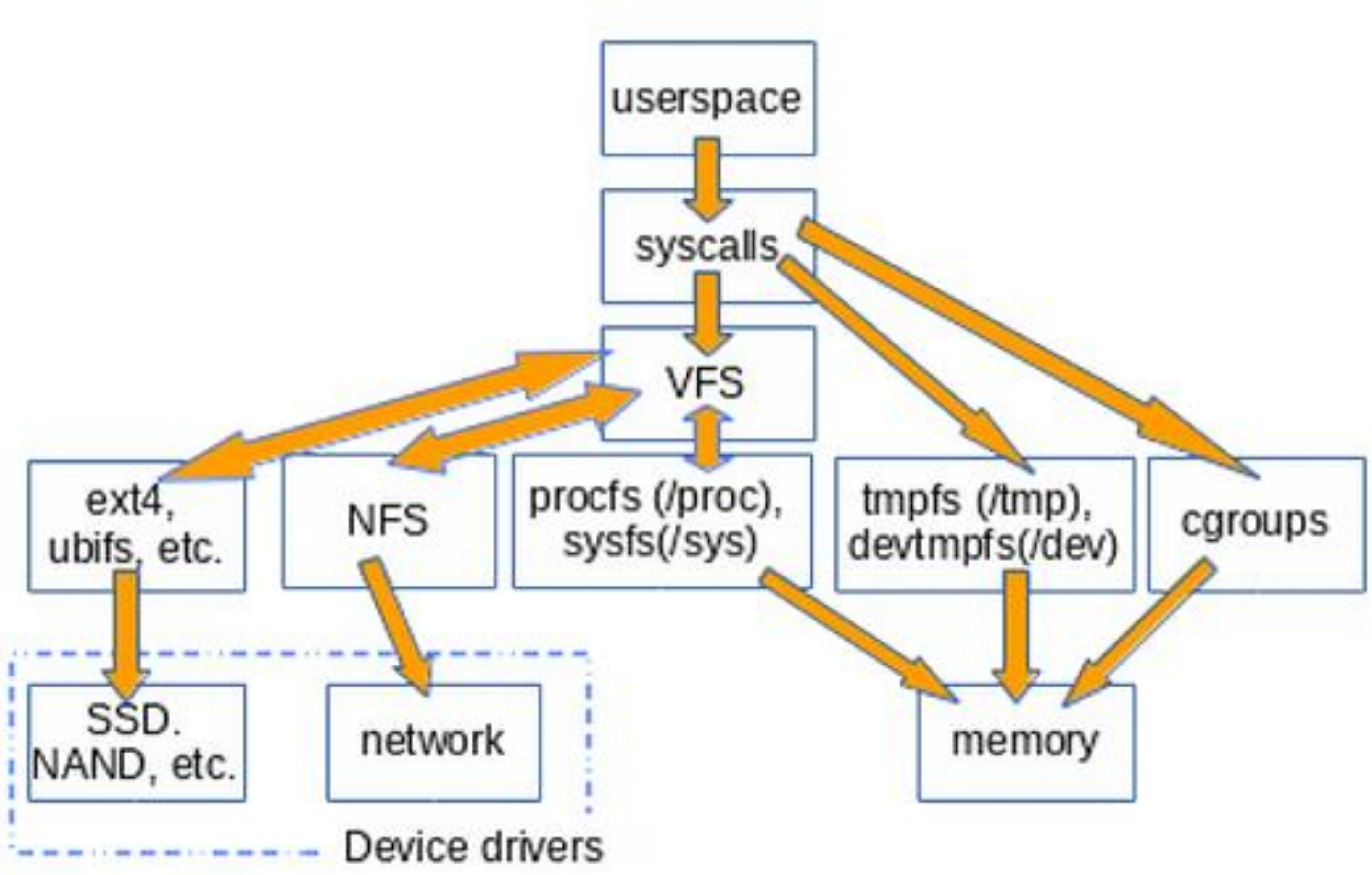
Virtual abstraction of file system in many OSes (including Linux)

- Provides virtual superblocks, inodes, files, etc
- Compatible with a variety of local and remote file systems

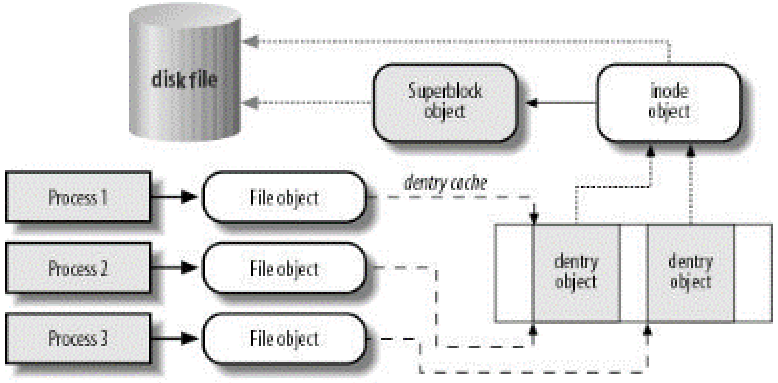
VFS allows the same system call interface (the API) to be used for different types of file systems in the same naming hierarchy

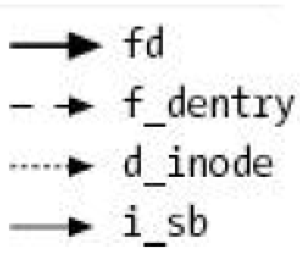
- The API is to the VFS layer, rather than any specific type of file system

Example Linux mounting tree



VFS Common File Model in Linux



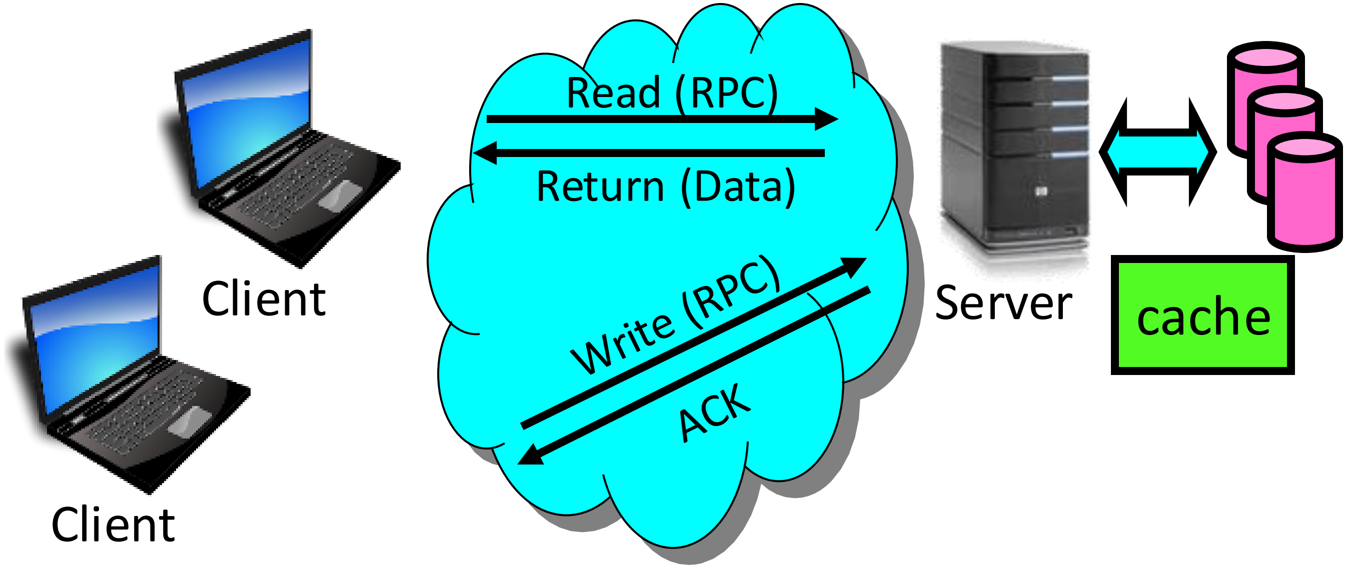


Four primary object types for VFS:

- superblock object: represents a specific mounted filesystem
- inode object: represents a specific file
- dentry object: represents a directory entry
- file object: represents open file associated with process

May need to fit the model by faking it

Simple Distributed File System

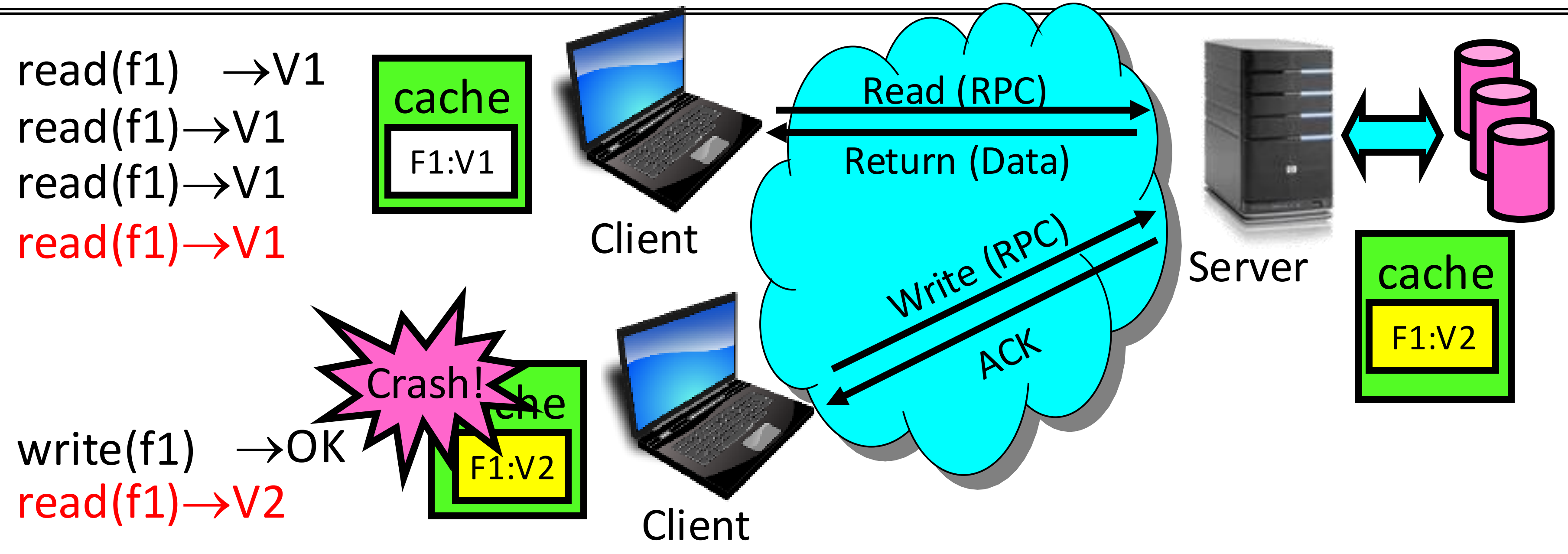


Remote Disk: Reads and writes forwarded to server. Use Remote Procedure Calls (RPC) to translate file system calls into remote requests

Advantage: Server provides consistent view of file system to multiple clients

Problems? Performance!
- Going over network is slower than going to local memory
- Lots of network traffic/not well pipelined
- Server can be a bottleneck

Use of caching to reduce network load



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…fast!

Problems:

- Failure:
  - » Client caches have data not committed at server
- Cache consistency! Client caches not consistent with server/each other