CS162
Operating Systems and Systems Programming
Lecture 21

Reliability & Distributed Systems

Professor Natacha Crooks
https://cs162.org/

Recall: File System Ideas

File Number is index into set of inode arrays
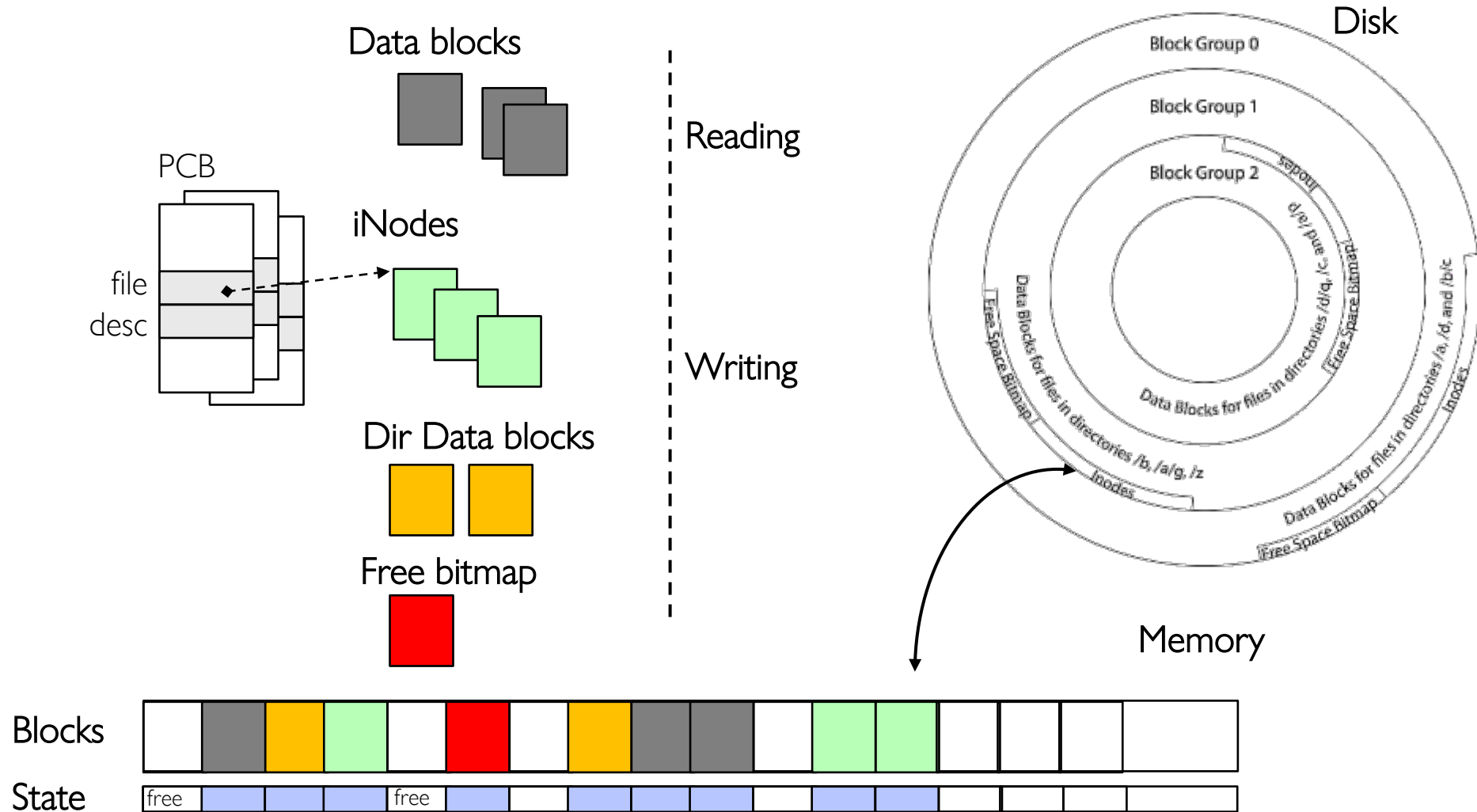
Index structure is an array of *inodes*

Each inode corresponds to a file and contains its metadata

Inode maintains a multi-level tree structure to find storage blocks for files

Original *inode* format appeared in BSD 4.1
Berkeley Standard Distribution Unix!

File System Buffer Cache

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

Data blocks

Reading

PCB

iNodes

file desc

Writing

Dir Data blocks

Free bitmap

Disk

Block Group 0

Block Group 1

Block Group 2

Memory

Blocks

State

free

free

Recall: Dealing with Persistent State

Buffer Cache: write back dirty blocks periodically, even if used recently

- Why? To minimize data loss in case of a crash
- Linux does periodic flush every **30** seconds

Not foolproof! Can still crash with dirty blocks in the cache

- What if the dirty block was for a directory?
  - »Lose pointer to file's inode (leak space)
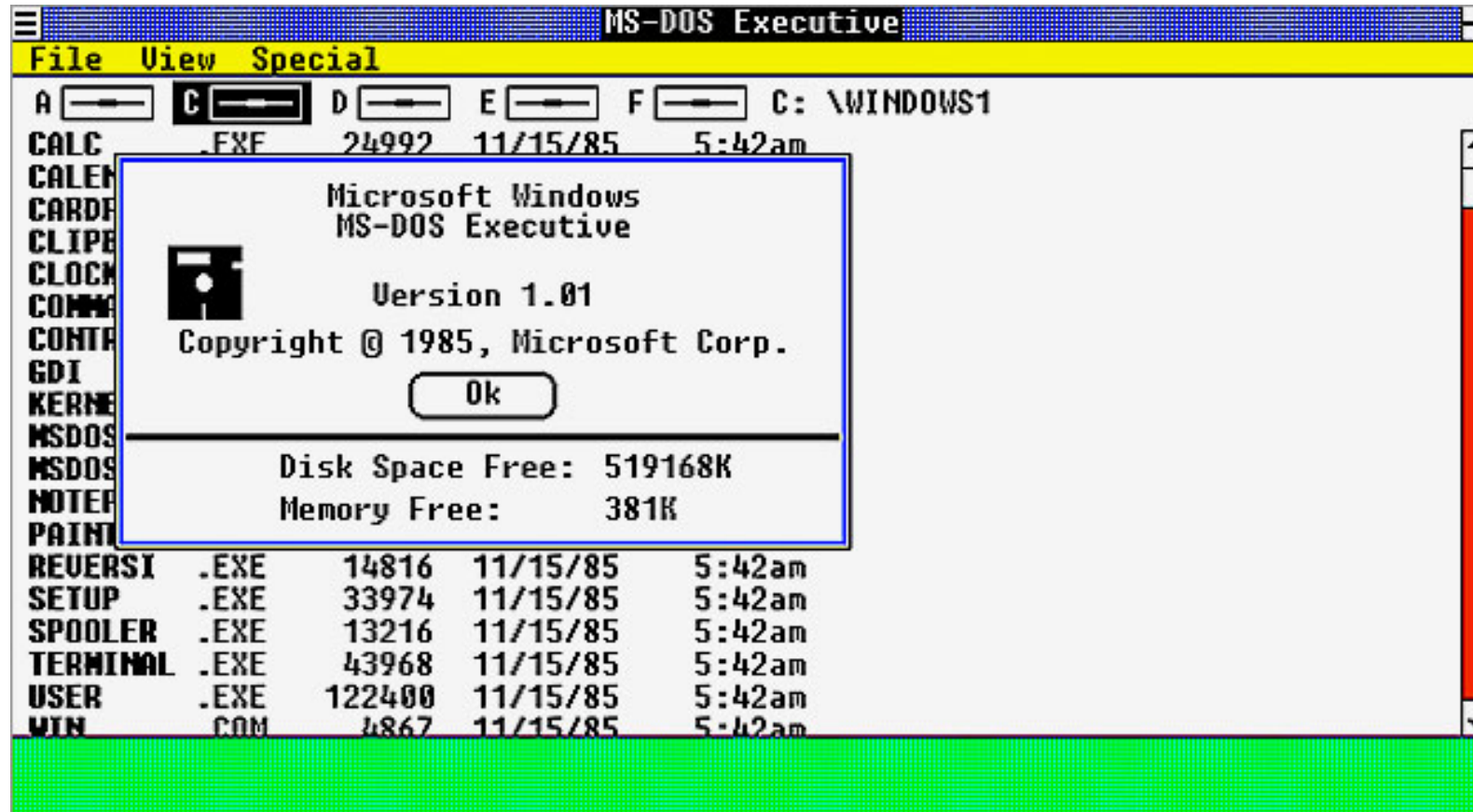 - »File system now in inconsistent state

Recall: Boom!

Happy Birthday Windows!

Happy 40th Birthday to Windows! Was first announced on Nov 10th 1983

Recall: 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?

Two Reliability Approaches

Careful Ordering and Recovery

FAT & FFS + (fsck)

Each step builds structure,
Data block $\Leftarrow$ inode $\Leftarrow$ free $\Leftarrow$ 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

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., **W**ord, emacs autosaves)

Berkeley 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

Reliability Approach #2: Copy on Write File Layout

Create a new version of the file with the updated data
- Reuse blocks that don't change much of what is already in place

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

More General Reliability Solutions

Use Transactions for atomic updates
- Ensure that multiple related updates are performed atomically
- i.e., if a crash occurs in the middle, the state of the systems reflects either all or none of the updates
- Most modern file systems use transactions internally to update filesystem structures and metadata
- Many applications implement their own transactions

Provide Redundancy for media failures
- Redundant representation on media (Error Correcting Codes)
- Replication across media (e.g., RAID disk array)

Transactions

Closely related to critical sections for manipulating shared data structures

They extend concept of atomic update from memory to stable storage
- Atomically update multiple persistent data structures

Many ad-hoc approaches
- FFS carefully ordered the sequence of updates so that if a crash occurred while manipulating directory or inodes the disk scan on reboot would detect and recover the error (fsck)
- Applications use temporary files and rename

Key Concept: Transaction

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

```
┌─────────────────────┐   transaction    ┌─────────────────────┐
│                     │                  │                     │
│ consistent state 1  │─────────────────▶│ consistent state 2  │
│                     │                  │                     │
└─────────────────────┘                  └─────────────────────┘
```

Recall: Code in a critical section appears atomic to other threads

Transactions extend the concept of atomic updates from *memory* to *persistent storage*

Typical Structure

Begin a transaction – get transaction id

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

Commit the transaction

"Classic" Example: Transaction

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

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

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

 UPDATE branches SET balance = balance + 100.00 WHERE
    name = (SELECT branch_name FROM accounts WHERE name
    = 'Bob');
COMMIT;    --COMMIT WORK
```
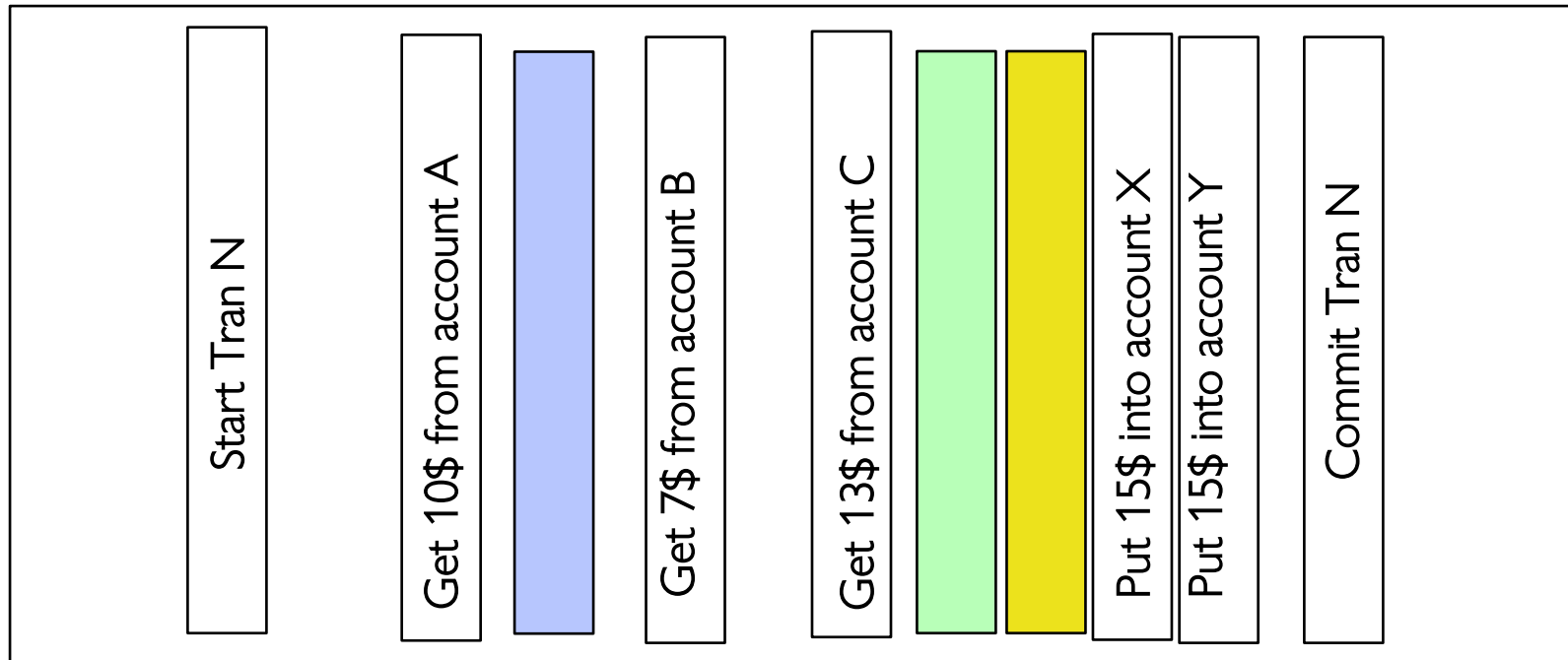
Transfer $100 from Alice's account to Bob's account

Concept of a log

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

Start Tran N | Get 10$ from account A | | Get 7$ from account B | Get 13$ from account C | | | Put 15$ into account X | Put 15$ into account Y | Commit Tran N

Write-Ahead Logging

Better reliability through use of log
- Changes are treated as transactions
- A transaction is committed once it is written to the log
  - » Data forced to disk for reliability
  - » Process can be accelerated with NVRAM
- Although File system may not be updated immediately, data preserved in the log

Difference between "Log Structured" and "Journaled"
- In a Log Structured filesystem, data stays in log form
- In a Journaled filesystem, Log used for recovery

Journaling File Systems

Don't modify data structures on disk directly

Write each update as transaction recorded in a log
- Commonly called a journal or intention list
- Also maintained on disk (allocate blocks for it when formatting)

Once changes are in the log, they can be safely applied to file system
- e.g. modify inode pointers and directory mapping

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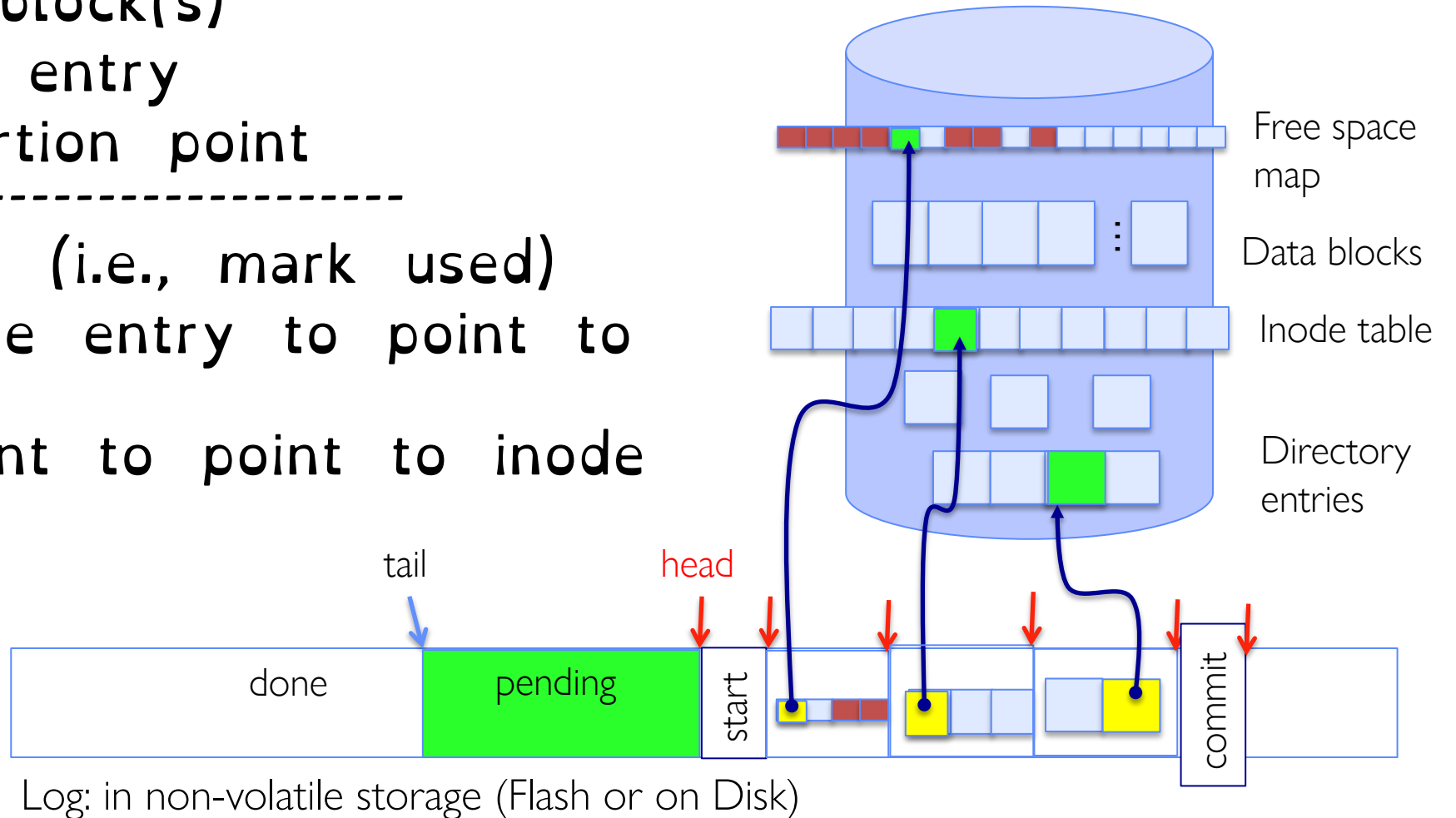Find dirent insertion point
----
Write map (i.e., mark used)
Write inode entry to point to block(s)
Write dirent to point to inode

Free space map

Data blocks

Inode table

Directory entries
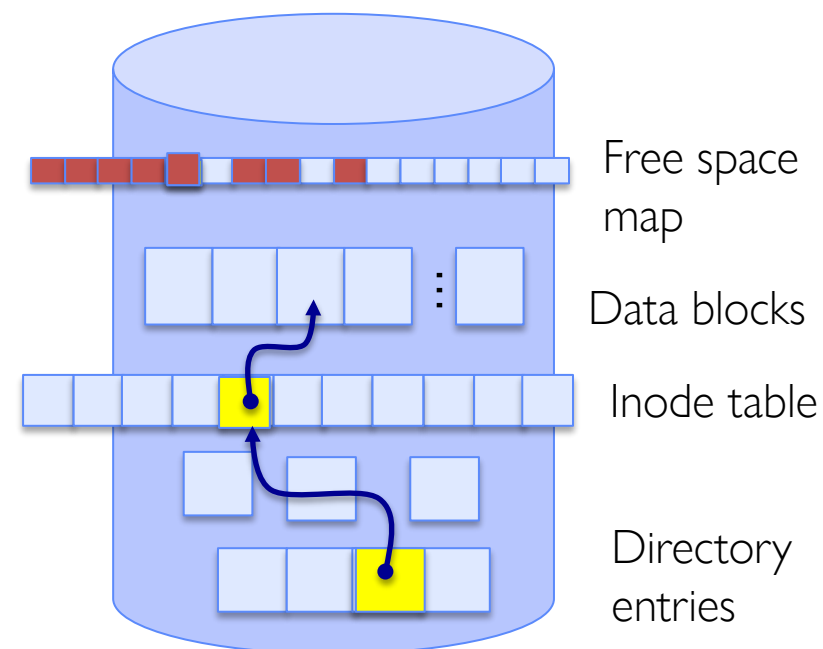
Creating a File (With Journaling)

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

Free space map

Data blocks

Inode table

Directory entries

tail   head

done   pending   start   commit

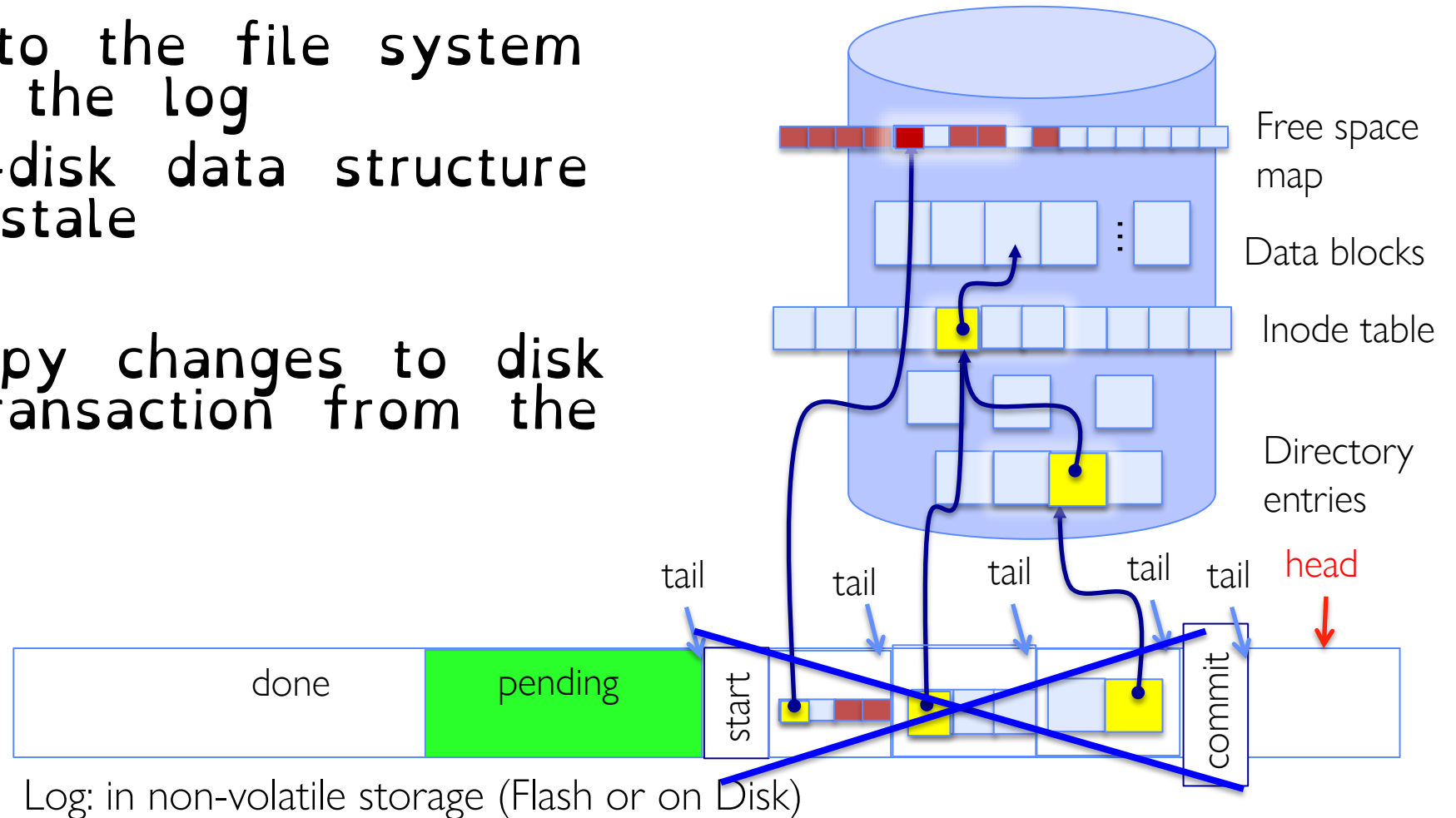Log: in non-volatile storage (Flash or on Disk)

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

Free space map

Data blocks

Inode table

Directory entries

tail   tail   tail   tail   tail   head

| done | pending | start | | | | commit | |

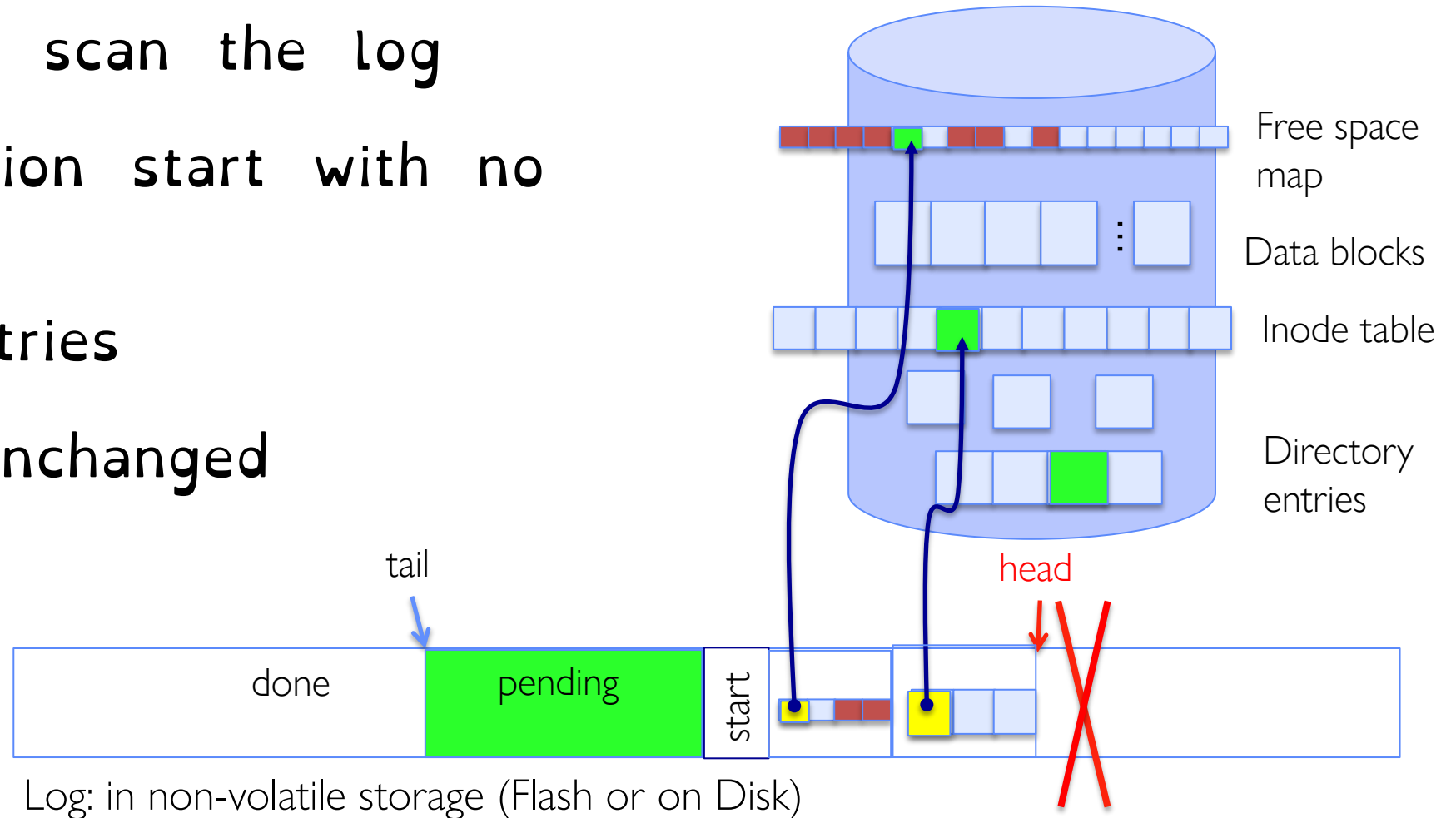Log: in non-volatile storage (Flash or on Disk)

Crash Recovery: Discard Partial Transactions

Upon recovery, scan the log

Detect transaction start with no commit
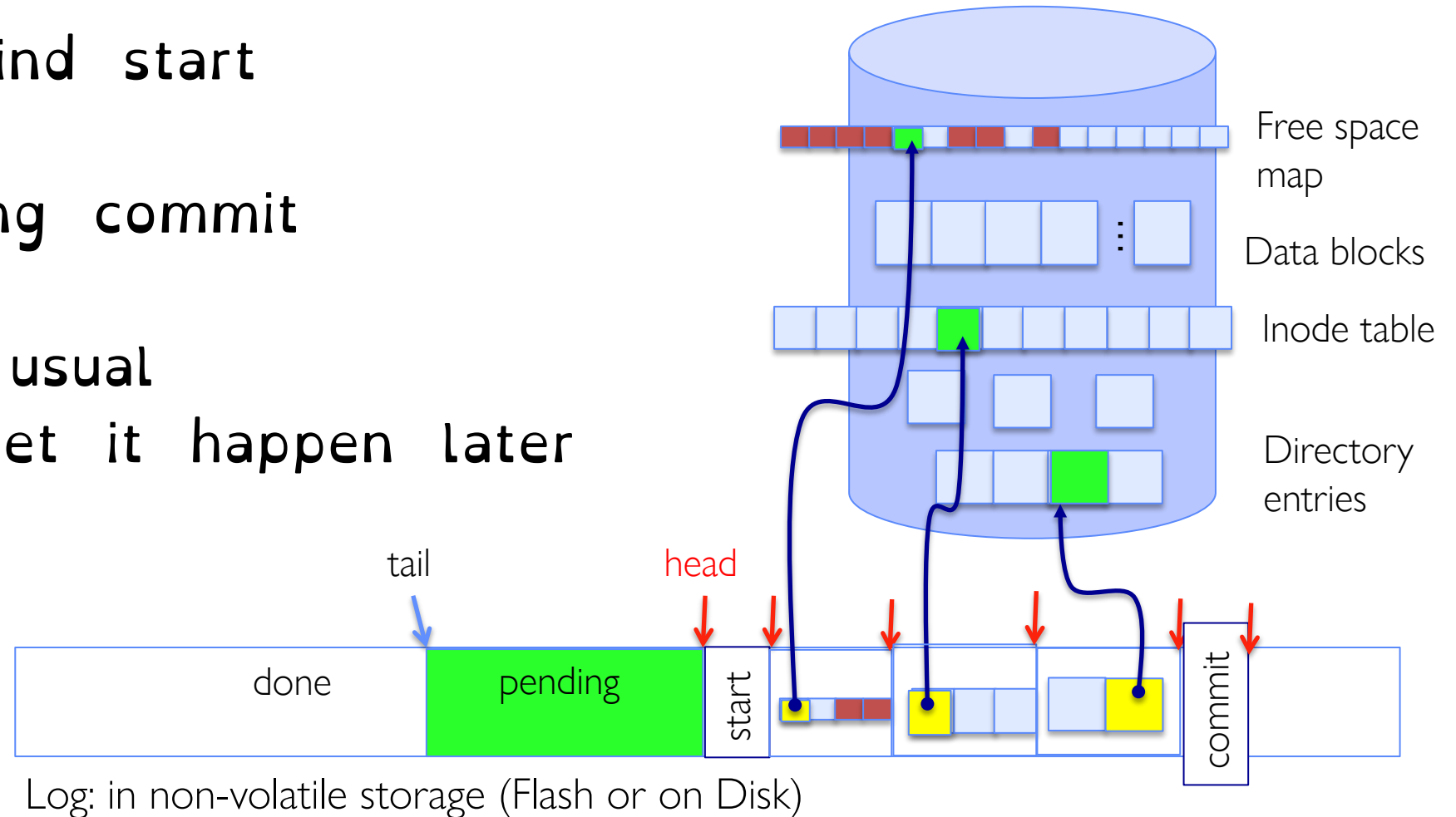
Discard log entries

Disk remains unchanged

Free space map

Data blocks

Inode table

Directory entries

tail

head

done

pending

start

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

Free space map

Data blocks

Inode table

Directory entries

tail   head

done   pending   start   commit

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

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,
The Godfather of Distributed Systems

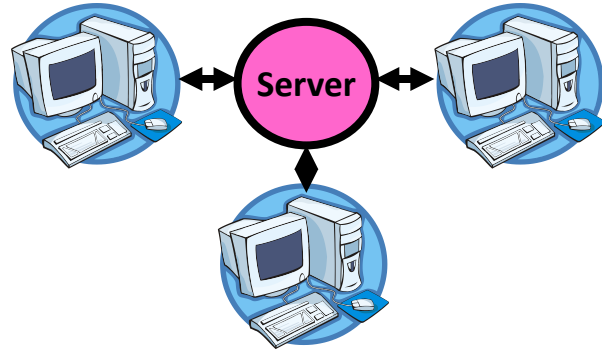# Centralised vs Distributed Systems

Considered a single computer!
All computation was done on
the local computer in isolation

The world is a large
distributed system
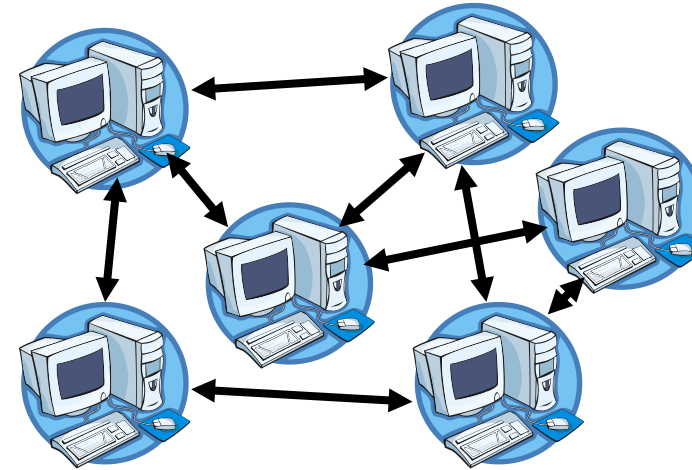
Two types of distributed systems

Client/Server Model

Peer-to-Peer Model

One or more server provides *services* to clients

Clients makes *remote procedure calls* to server
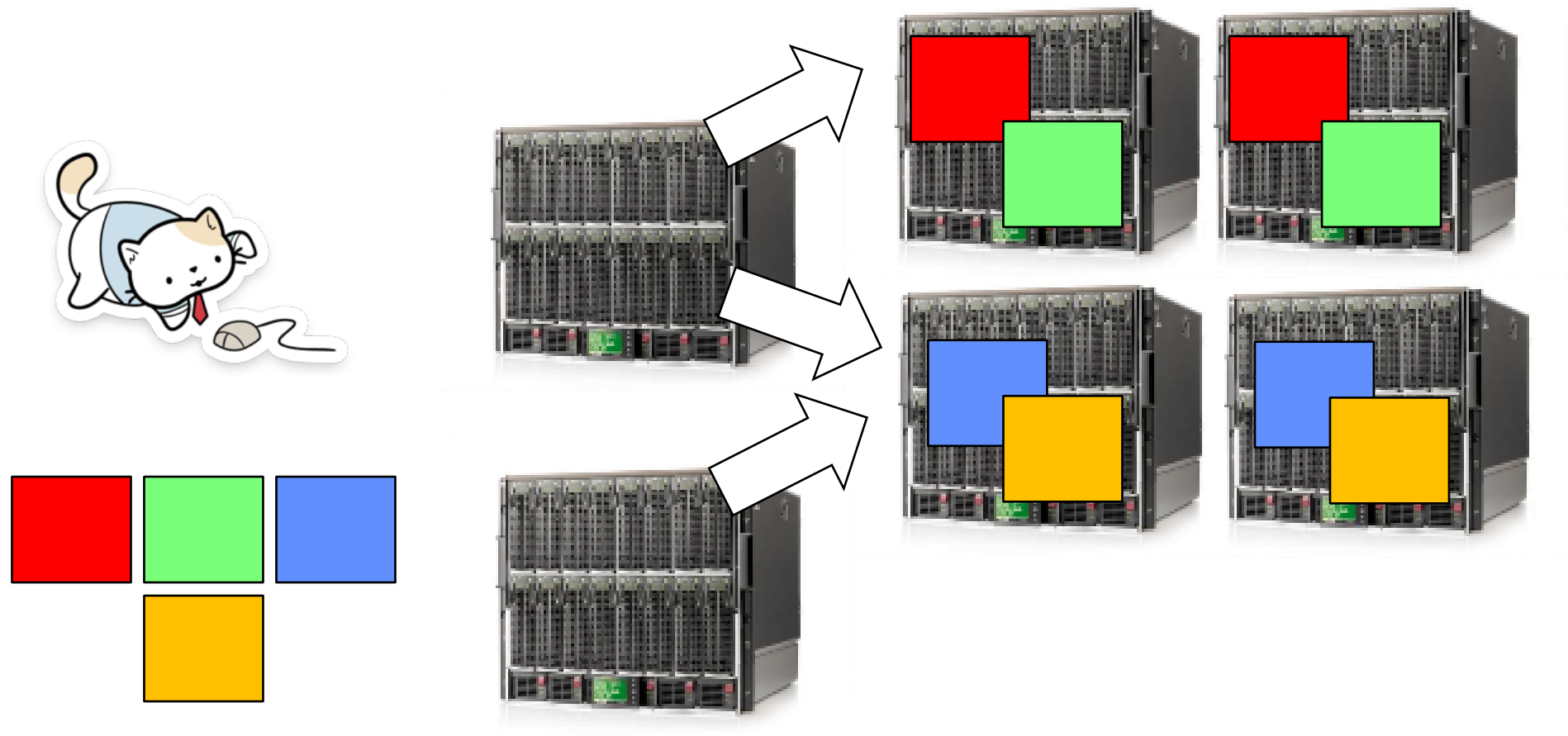
Server serves *requests* from clients

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 behaviour 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 Systems
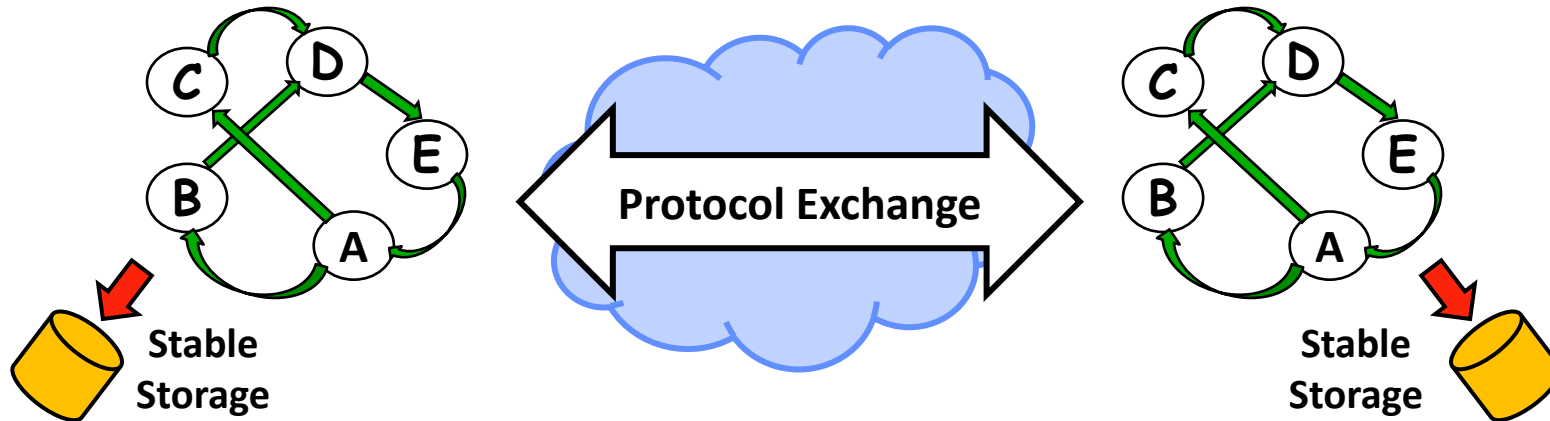
Peer-To-Peer System:
The Internet

Distributed Data Processing

Coordination
(Atomic Commit and
Consensus)

How do machines communicate?

A protocol is an agreement on how to communicate,

–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 Natacha…."
   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

Message Passing

How do you actually program a distributed application?
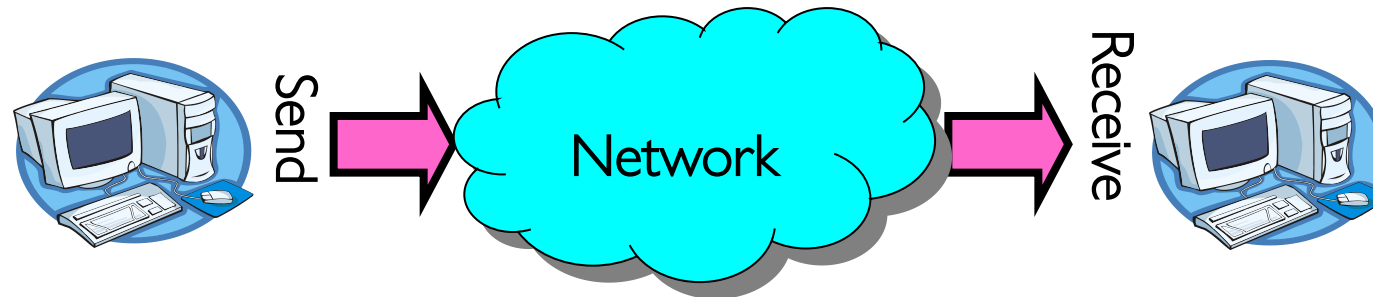
Interface:

- 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

Which end of a machine-recognized object (e.g., int) does its byte-address refer to?

Big Endian: address is the most-significant bits

Little Endian: address is the least-significant bits

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

```c
int main(int argc, char *argv[])
{
  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
```

What About Richer Objects?

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

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

`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

Google Protobuffers, **JSON** and **XML** are commonly used in web applications

Lots of ad-hoc formats

```
{ " faculty":
    [
            {id: 1,
             "name": "Anthony",
             "lastname": "Joseph"
             },
            {id: 2,
             "name": "Natacha",
             "lastname": "Crooks"
            }
    ]
}
```

Data Serialization Formats

| Name	Creator-maintainer	Based on	Standardized?	Specification	Binary?	Human-readable?	Supports references?[e]	Schema-IDL?	Standard APIs	Supports 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

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

Another option: 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
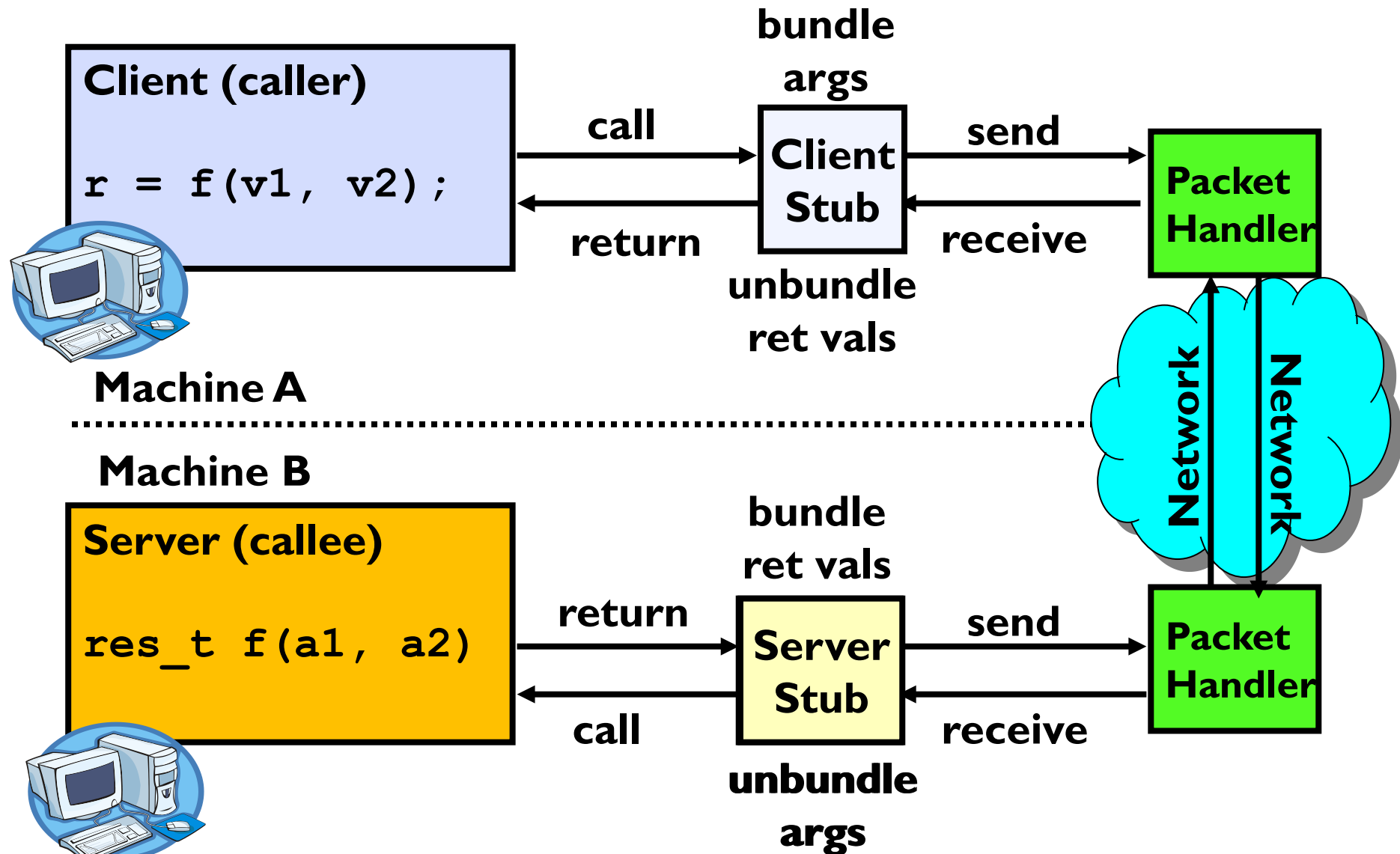
RPC Implementation

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 off, wait for result, unpack result and return to caller
  - » Code for server to unpack message, call procedure, pack results, send them off

RPC Details (2/3)

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

How does client know which mbox (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 → 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

- ifferent 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 complex