PERSISTENCE: DISTRIBUTED FILE SYSTEMS

Kai Mast
CS 537
Fall 2022

RECAP: FSCK

- File system check (FSCK) verifies correctness of a file system and (tries to) repair it if needed
- Scans entire file system to find inconsistencies, e.g.,
  - References to invalid data blocks
  - Incorrect link counts
  - Inconsistencies in the inode or data block bitmaps
- Scan is usually performed when file system is not in use or in read-only mode

RECAP: JOURNALING

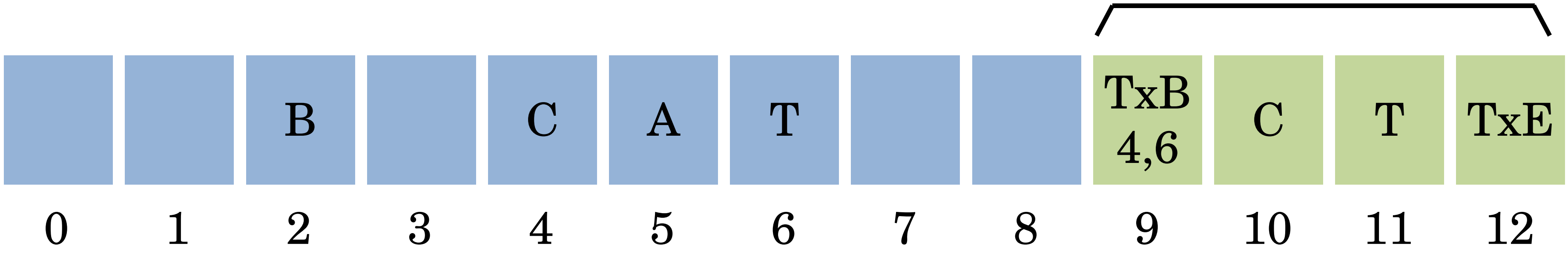
Keep a log (journal) of writes

- Writes that logically belong together are bundled into transactions
- Allows to recover in-progress transactions after crashes
- Replay writes from log
  - Will overwrite any partial writes
  - Resolves any potential inconsistencies

Clean up log after flushing writes (checkpoint)

- Log is reused by other other transactions after checkpointing
- Flushing writes can be batched to increase performance

ORDERING FOR CONSISTENCY



What operations can proceed in parallel and which must be strictly ordered?

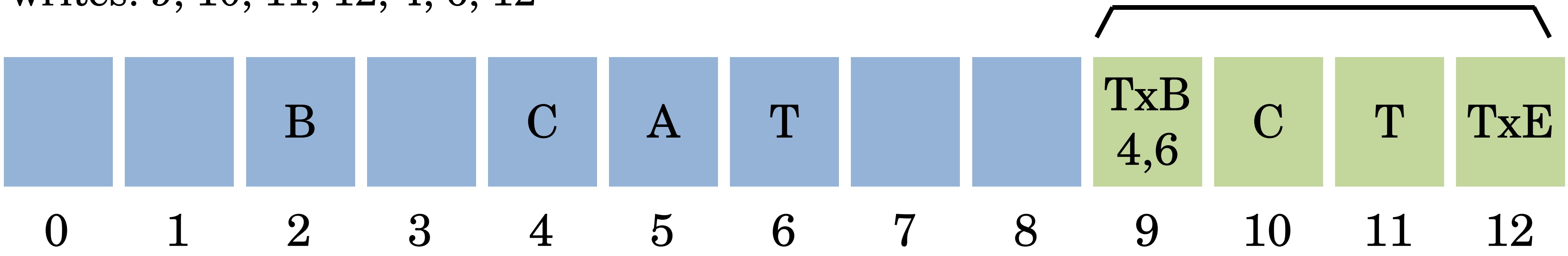
Strict ordering is expensive:

- must flush from memory to disk
- tell disk not to reorder
- tell disk can't cache, must persist to final media

writes: 9, 10, 11, 12, 4, 6, 12

ORDERING FOR CONSISTENCY

writes: 9, 10, 11, 12, 4, 6, 12



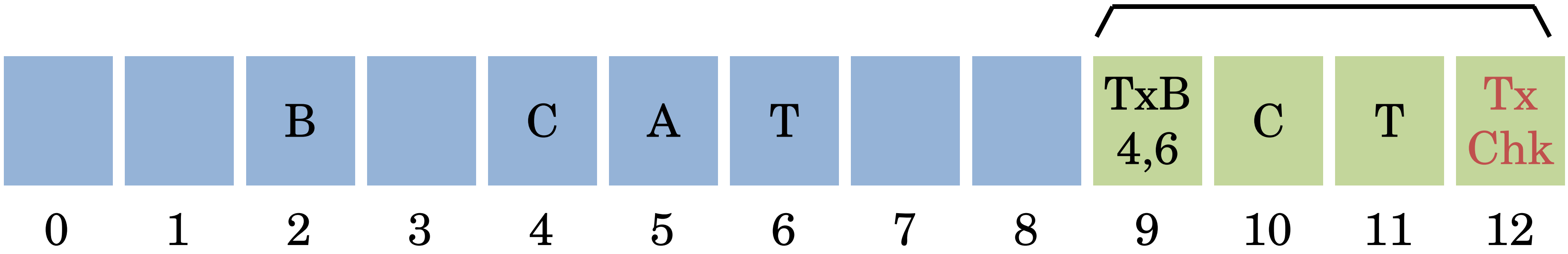
transaction: write C to block 4; write T to block 6

Barriers
1) Before journal commit, ensure journal transaction entries complete
2) Before checkpoint, ensure journal commit complete
3) Before free journal, ensure checkpoint (in-place updates) complete

write order: 9,10,11 | 12 | 4,6 | 12

CHECKSUM OPTIMIZATION

Can we get rid of barrier between (9, 10, 11) and 12 ?

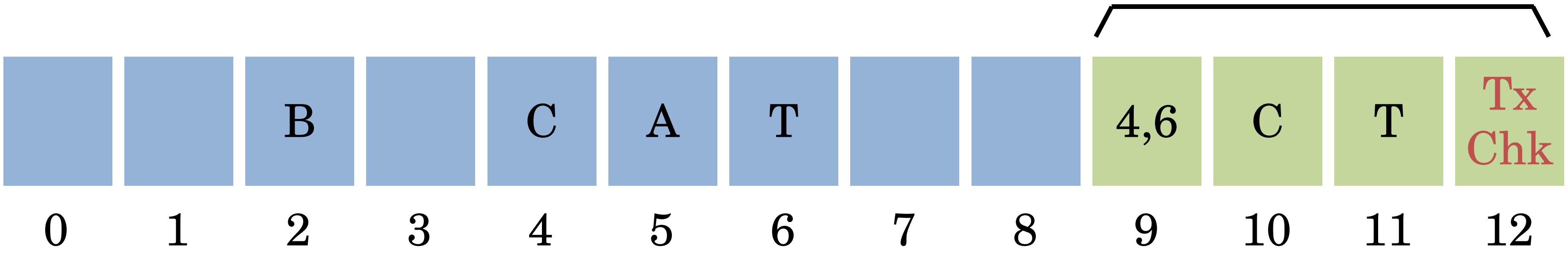


write order: 9,10,11,12 | 4,6 | 12

In last transaction block, store checksum of rest of transaction (Calculate over blocks 9, 10, 11)

How does recovery change?
During recovery: If checksum does not match, treat as not valid

WRITE BUFFERING OPTIMIZATIONS



Batched updates

- If two files are created, inode bitmap, inode etc. are written twice
- Mark as dirty in-memory and batch many updates into one transaction

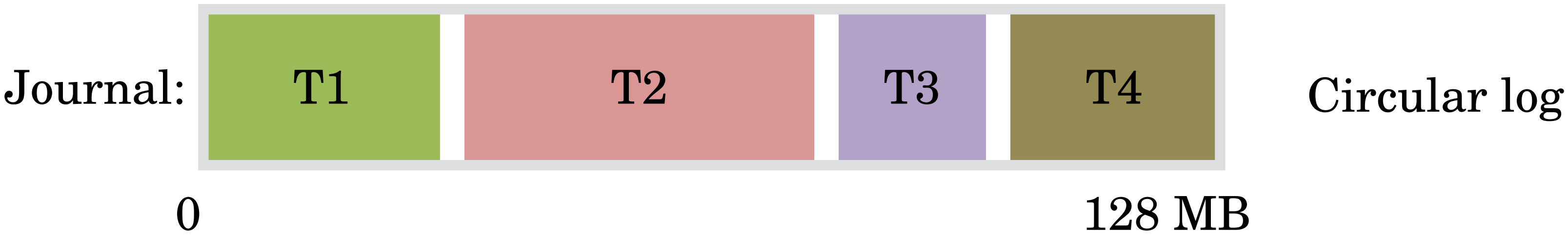
Delay checkpoints

- Note: after journal write, there is no rush to checkpoint
- If system crashes, we still have persistent copy of written data!
- Journaling is sequential (fast!), checkpointing is random (slow!)
- Solution? Delay checkpointing for some time

CIRCULAR LOG

Difficulty: need to reuse journal space

Solution: keep many transactions for un-checkpointed data



Must wait until transaction T1 is checkpointed before it can be freed and reused for next transaction, T5

Problem: Journaling Writes Twice!

How to avoid writing all disk blocks Twice?

Observation:

- Most of writes are user data (especially sequential writes)
- If user data is not consistent, file system still operates correctly

Strategy: Journal all metadata, including superblock, bitmaps, inodes, indirects, directories

- Guarantees metadata is consistent if crash occurs
- Will not leak space or re-allocate blocks to multiple files

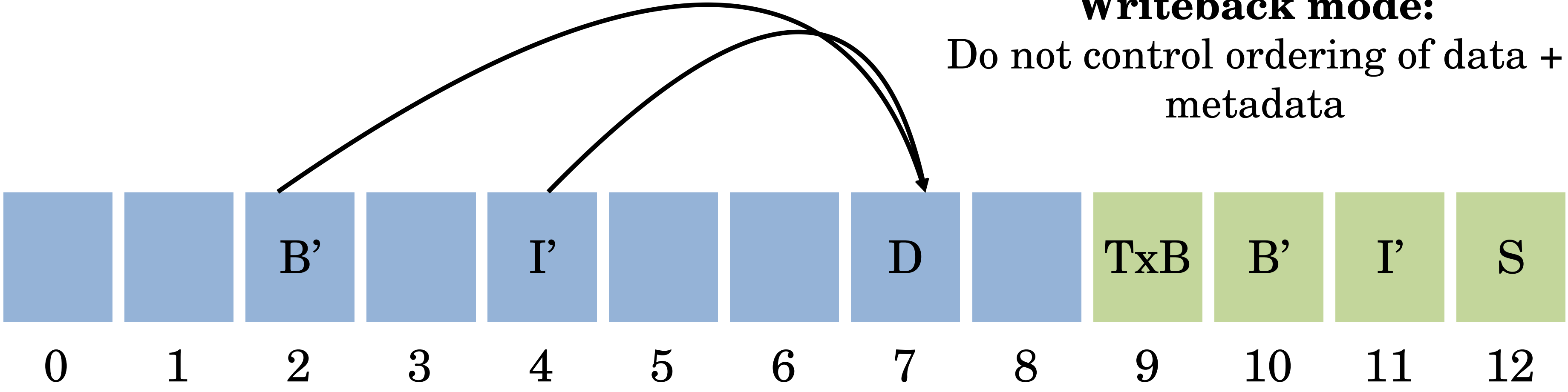
For regular user data, write it to in-place location whenever convenient

Implication: Files may contain garbage (partial old, partial new) if we crash and recover

- Application needs to deal with corrupted files after crash

METADATA JOURNALING

Writeback mode:
Do not control ordering of data + metadata



Transaction: Append to inode I

- Ensures B’ and I’ are updated atomically

Write Order: 9,10,11,7,12 | 2,4 | 12

What if we crash? B’ or I’ might point to garbage data (file contents are corrupted)

- But inodes and bitmaps are still correct (file system integrity maintained)

ORDERED JOURNAL

Ordered-mode journaling: Still only journal metadata
But write data **before** the transaction!

transaction:
append to inode I

		B’		I’			D		TxB	B’	I’	TxE
0	1	2	3	4	5	6	7	8	9	10	11	12

Write Order: 7 | 9,10,11,12 | 2,4 | 12

1. What happens if crash **before commit** (before update B and I)?
   - B indicates D currently free, I does not point to D
   - Lose D, but that is problem for application, not file system
2. **Crash after commit?** Can replay journal, update in-place B’ and I’ which will point to D, as desired
3. **After checkpoint?** Everything fine; if didn’t clear TxE, will replay transaction, extra work but no problems

SO FAR: PHYSICAL JOURNALS

Example: Append 10 bytes to a file (new size is 1010)

TxB id=1337 length=2	**inode** … size=1010	**data block** (4 kB)	**TxE** id=1337 checksum

Write out lots of information, but how much was really needed?

Actual changed data is much smaller!

ANOTHER APPROACH: LOGICAL JOURNAL

TxB id=1337 length=1	• *"set size in inode 1 to 1010"* • *"append 10 bytes to data block 4"*	**TxE** id=1337 checksum

Logical journals record **changes** to bytes, not contents of new blocks

On recovery: Need to read existing contents of in-place data and (re-)apply changes

DISTRIBUTED (FILE) SYSTEMS

(Book Chapter 49)

WHAT IS A DISTRIBUTED SYSTEM?

A distributed system is one where a machine I've never heard of can cause my program to fail. — Leslie Lamport

Definition: More than 1 machine working together to solve a problem

Issues:

- Concurrency of components
- Lack of global clock
- Independent failure of components

Examples

- Client/Server: web client and web server
- Cluster or Data Center: page rank computation, running massively parallel map-reduce
- Peer-to-Peer Network: file sharing, blockchains

WHY GO DISTRIBUTED?

More computing power (Better performance)

- Increase throughput
- Reduce latency

More storage capacity

- Data sharing

Fault-Tolerance

- Distributed systems continues to operate even if some machines fail

NETWORKING 101

Layer	Protocols
Application	HTTPS, SSH, **NFS**, …
Transport	TCP, **UDP**, …
Network	Mostly IP
Link	Ethernet, WiFi, …

Transmission Control Protocol (TCP)

- Creates a bidirectional stream of bytes between two parties
- Data is guaranteed to be delivered in order

User Datagram Protocol (UDP)

- Transmits fixed-size packets between two parties
- Delivery is unreliable
- Packets might get reordered
- Used by NFS

NEW CHALLENGES

System failure: need to worry about partial failure

Communication failure: network links unreliable

- bit errors
- packet loss
- link failure

Individual **nodes (machines)** crash and recover

- Some of our focus today

TYPES OF FILE SYSTEMS

Local FS (FFS, ext3/4, LFS):
Processes on same machine access shared files on that machine

Network FS (NFS, AFS):
Processes on different machines access shared files on a different machine

Many clients with a (nearby) single server…

GOALS FOR DISTRIBUTED FILE SYSTEMS

Fast + simple crash recovery

- Clients or servers might crash
- Keep protocol simple to minimize potential bugs

Transparent access

- Application will not "notice" that accesses are over the network
- Provide normal UNIX semantics

Reasonable performance

- Scale with number of clients?

NFS: NETWORK FILE SYSTEM

Think of NFS as more of a protocol than a particular file system

Many companies have implemented NFS since 1980s:
- Oracle/Sun, NetApp, EMC, IBM

We're looking at NFSv2
- NFSv4 has many changes

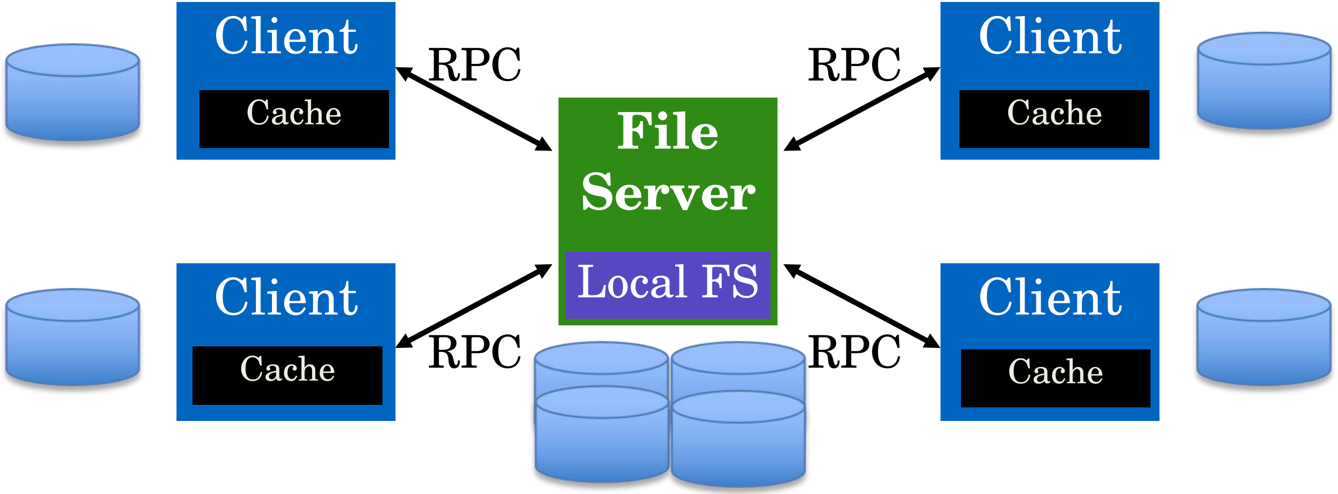
Why look at an older protocol?
- Simpler, focused goals (simple crash recovery, stateless)
- To compare and contrast NFS with AFS

NFS OVERVIEW

1. Architecture + Network API

2. Caching

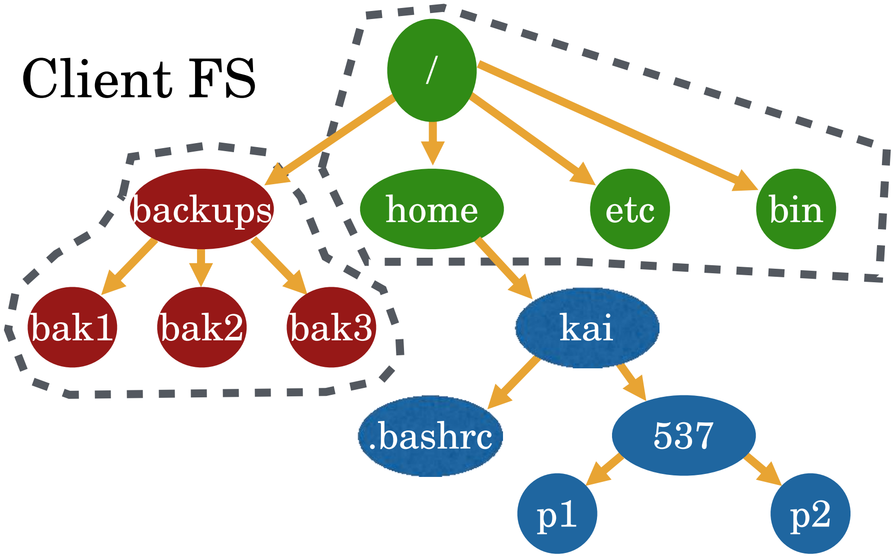
NFS ARCHITECTURE



RPC: Remote Procedure Call
Cache individual blocks of NFS files

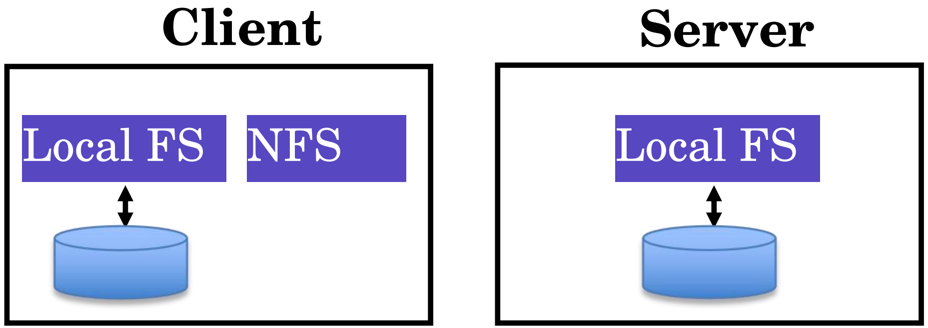
EXPORT FILE SYSTEM TO CLIENTS

Client FS



/dev/sda1 **on** /
/dev/sdb1 **on** /backups
NFS **on** /home/kai

Where will read to `/backups/bak1` go?
Where will read to `/home/kai/.bashrc` go?

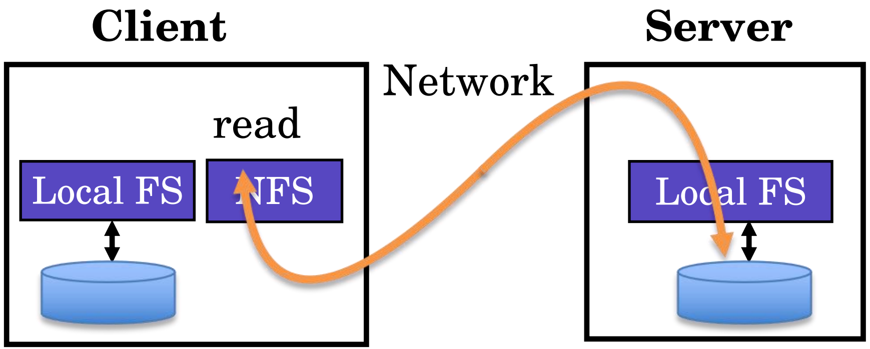


WHAT DO CLIENTS SEND TO SERVER? API STRATEGY 1

Attempt: Wrap regular UNIX system calls using RPC

- open() on client calls open() on server
- open() on server returns fd back to client

- read(fd) on client calls read(fd) on server
- read(fd) on server returns xdata back to client



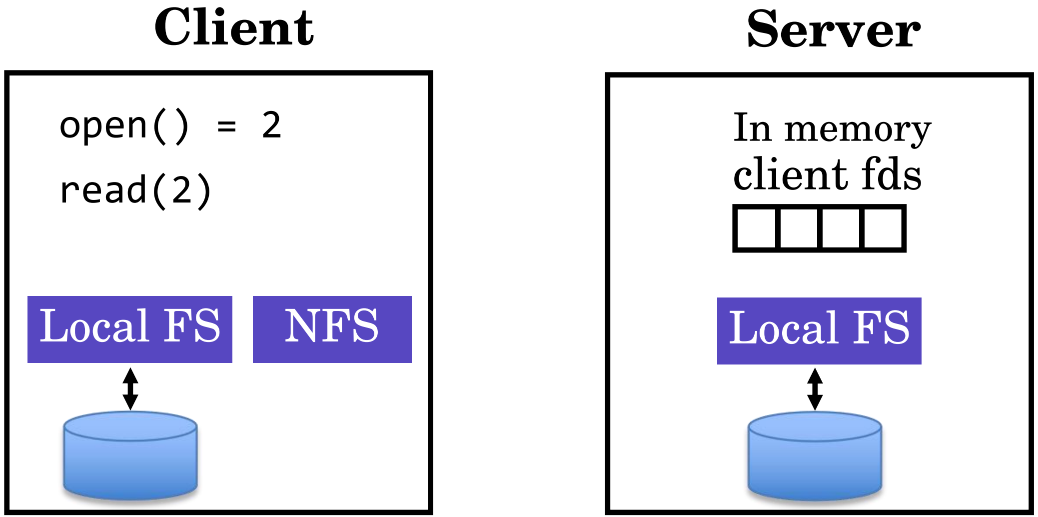
PROBLEM: FILE DESCRIPTORS ON SERVER

What about server crashes? (and reboots)

```
int fd = open("foo", O_RDONLY);
read(fd, buf, MAX);
read(fd, buf, MAX);
```

… **Server crash!**

```
read(fd, buf, MAX);
```



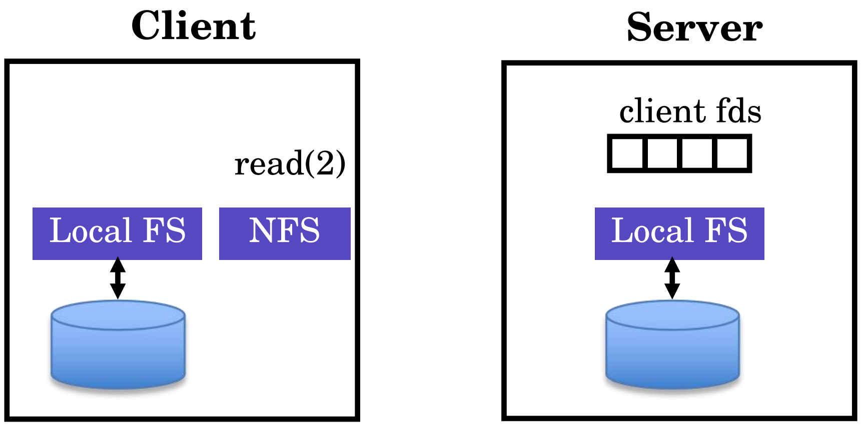
Remember: What is fd tracking?

Goal: behave like slow read

POTENTIAL SOLUTIONS

1. Run some crash recovery protocol when server reboots
   - Complex

2. Persist fds on server disk
   -
   - How long to keep fds? What if client crashes? misbehaves?

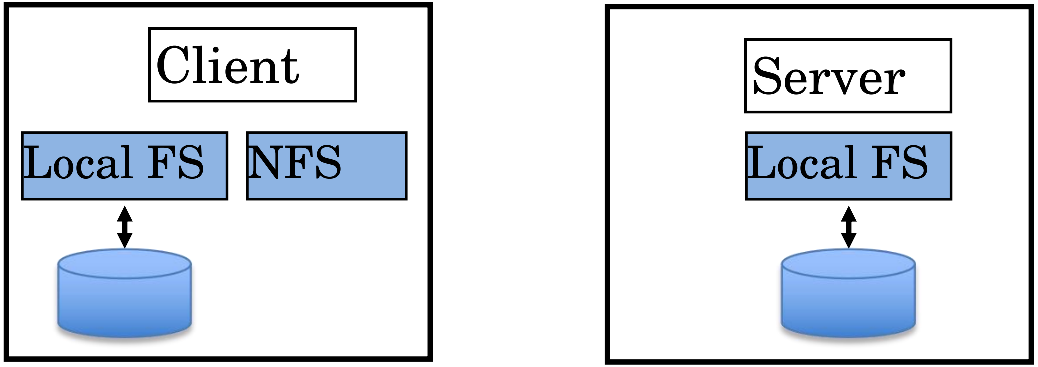


API STRATEGY 2: PUT ALL INFO IN REQUESTS

Every request from client completely describes desired operation

Use a "stateless" protocol!

- Server maintains no state about clients
- Server can still keep state to improve
- Can crash and reboot with no correctness problems (just slower performance)
- Main idea of NFSv2



STRATEGY 2: PUT ALL INFO IN REQUESTS

"Stateless" protocol: server maintains no state about clients

Need API change. Get rid of fds; One possibility:

```
read(path, buf, size, c                ;
write(path, buf, size, c               ;
```

Specify path and offset in each message

Server need not remember anything from clients

Pros? Server can crash and reboot transparently to clients

Cons?

API STRATEGY 3: INODE REQUESTS

```
inode = open(path);
read(inode, buf, size, offset);
write(inode, buf, size, offset);
```

With some new interfaces on server for accessing by inode number, this is pretty good!

Any correctness problems?

API STRATEGY 4: FILE HANDLES

```
fh = open(path);
read(fh, buf, size, offset);
write(fh, buf, size, offset);
```

File Handle = <volume ID, inode #, **generation #**>

File handles are opaque to clients

- Client should not (need to) interpret internals

Can NFS Protocol Include Append?

```
fh = open(path);
read(fh, buf, size, offset);
write(fh, buf, size, offset);
append(fh, buf, size);
```

Problem with `append()`? RPC often has "at-least-once" semantics

-
- Implementing "exactly once" requires state on server, which we are trying to avoid

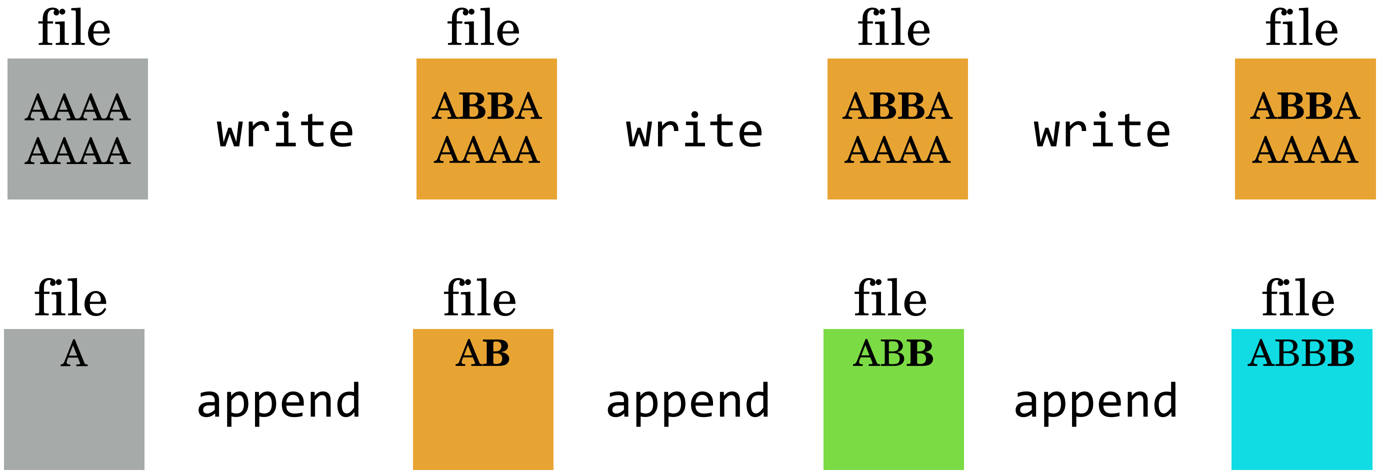
If RPC library replays messages, what happens when `append()` is retried on server?

-

IDEMPOTENT OPERATIONS

Solution: Design API so no that no harm is caused if we execute a function more than once

If f() is **idempotent**, then:



WHAT OPERATIONS ARE IDEMPOTENT?

Idempotent

- `write`
- any sort of read that doesn't change anything

Not idempotent

-

What about these?

- `mkdir`
- `create`

API STRATEGY 4: FILE HANDLES

Do not include append() in NFS protocol

```
fh = open(char *path);
read(fh, buf, size, offset);
write(fh, buf, size, offset);
append(fh, buf, size);
```

File Handle = <volume ID, inode #, generation #>

Can applications call append????