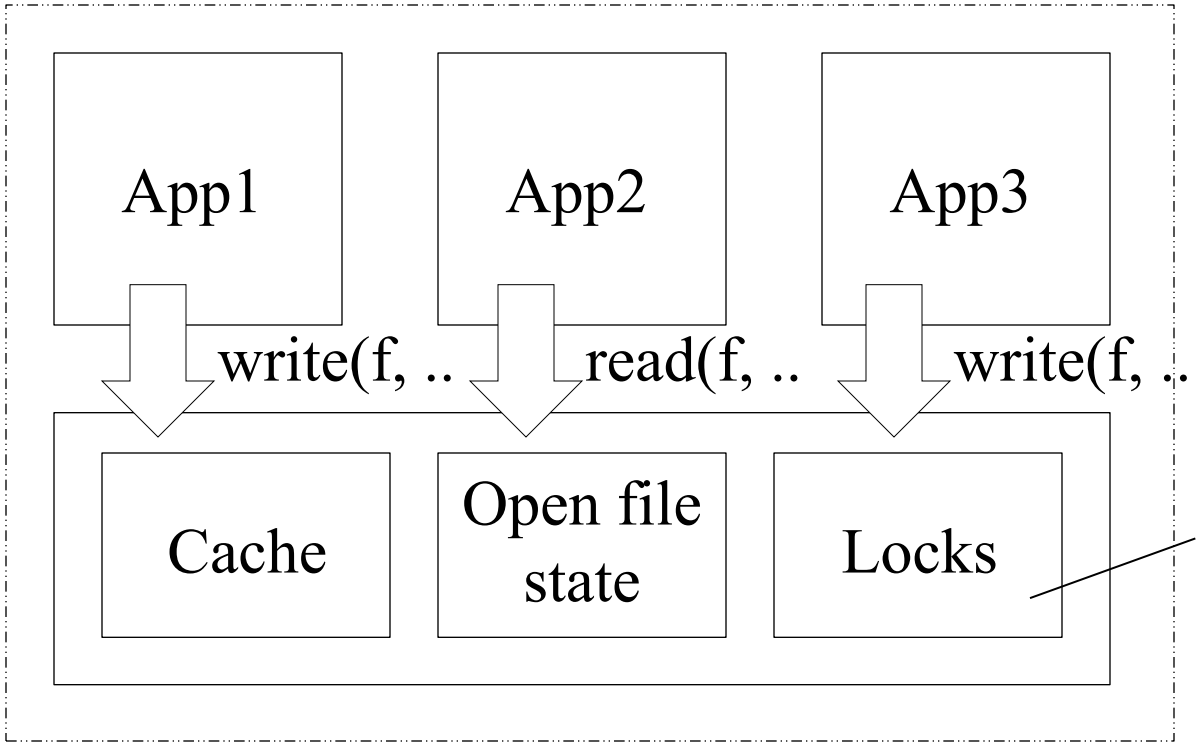
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
UNIVERSITY OF CRETE

HY590.45
Modern Topics in Scalable Storage Systems

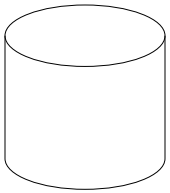
Kostas Magoutis
magoutis@csd.uoc.gr
http://www.csd.uoc.gr/~hy590-45

File sharing in a single system



- File lock
- Range lock
  - Byte range
- Type of lock (op)
  - R/W

Single system

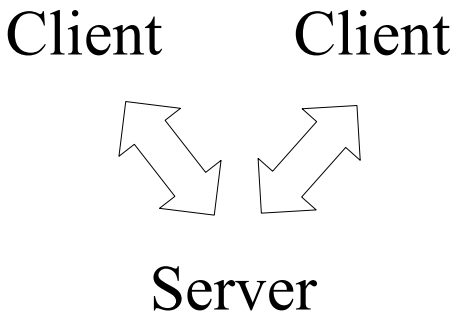


Data + metadata

Crash recovery? Not an issue in a single system

Network file sharing

- Benefits
  - Ability to access files from many locations
    - E.g., home directories
  - Consolidate storage management



- Makes it possible to share files
  - Often concurrent readers or single writer
  - Less often, concurrent writers
    - Exclusive access to non-overlapping parts of file
    - Several data producers, concurrent append to shared file
    - Infrequent in engineering/office type workloads

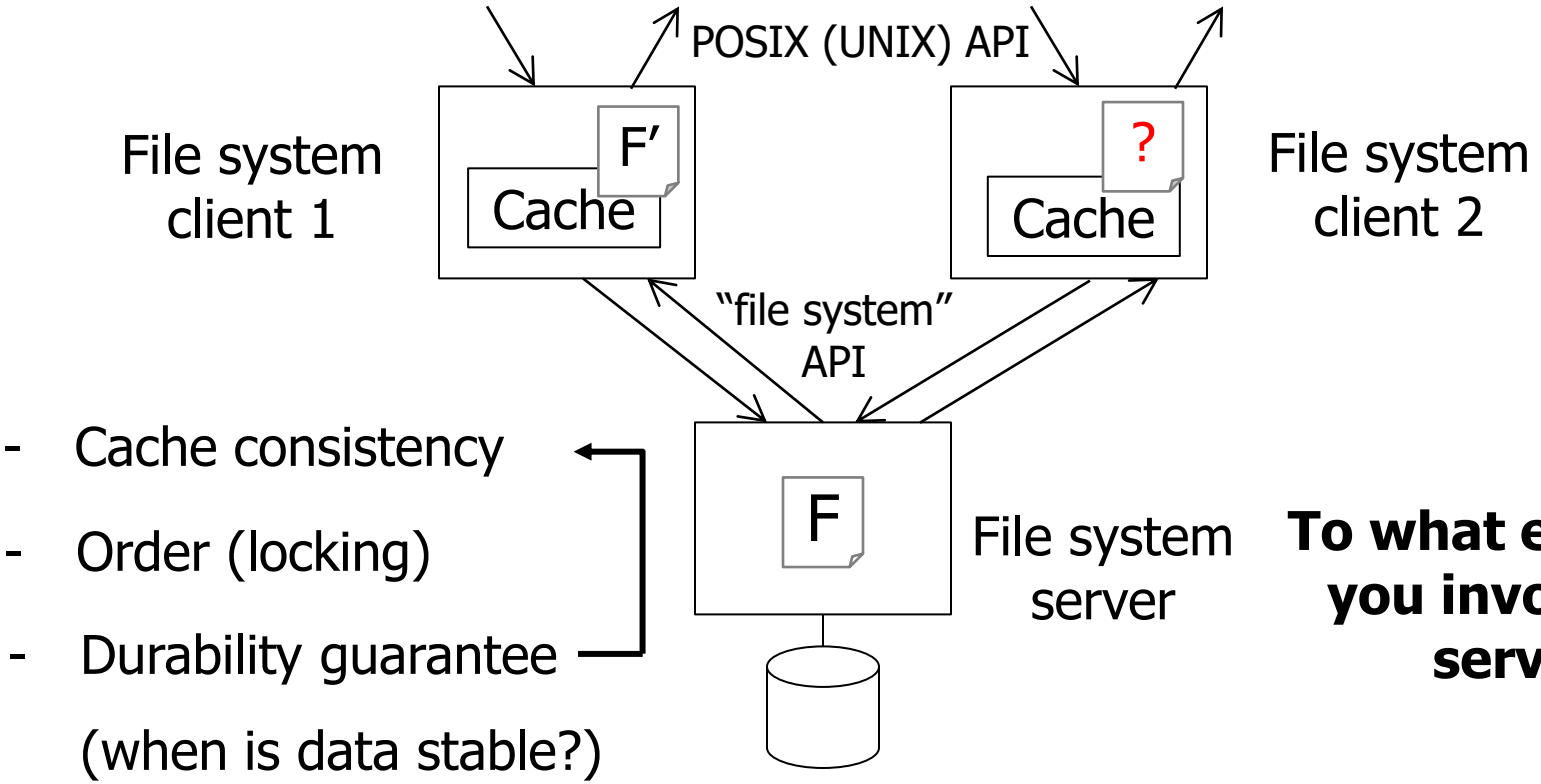
Network file sharing

Application 1

```
fd = open(F)
lseek(fd, 128, SEEK_SET)
write(fd, buf, 16384)
```

Application 2

```
fd = open(F)
lseek(fd, 128, SEEK_SET)
read(fd, buf, 16384)
```



- Cache consistency
- Order (locking)
- Durability guarantee

(when is data stable?)

To what extent do you involve the server?

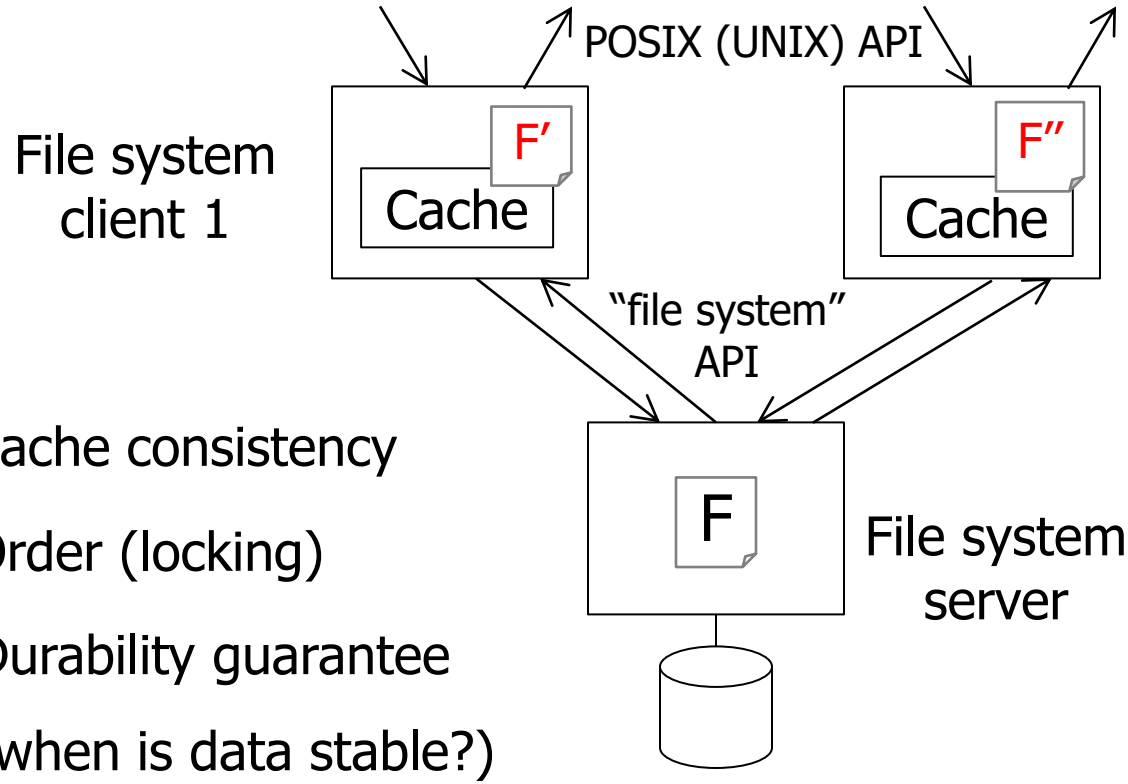
Network file sharing

Application server 1

```
fd = open(F, O_APPEND)
write(fd, buf, 16384)
```

Application server 2

```
fd = open(F, O_APPEND)
write(fd, buf, 16384)
```

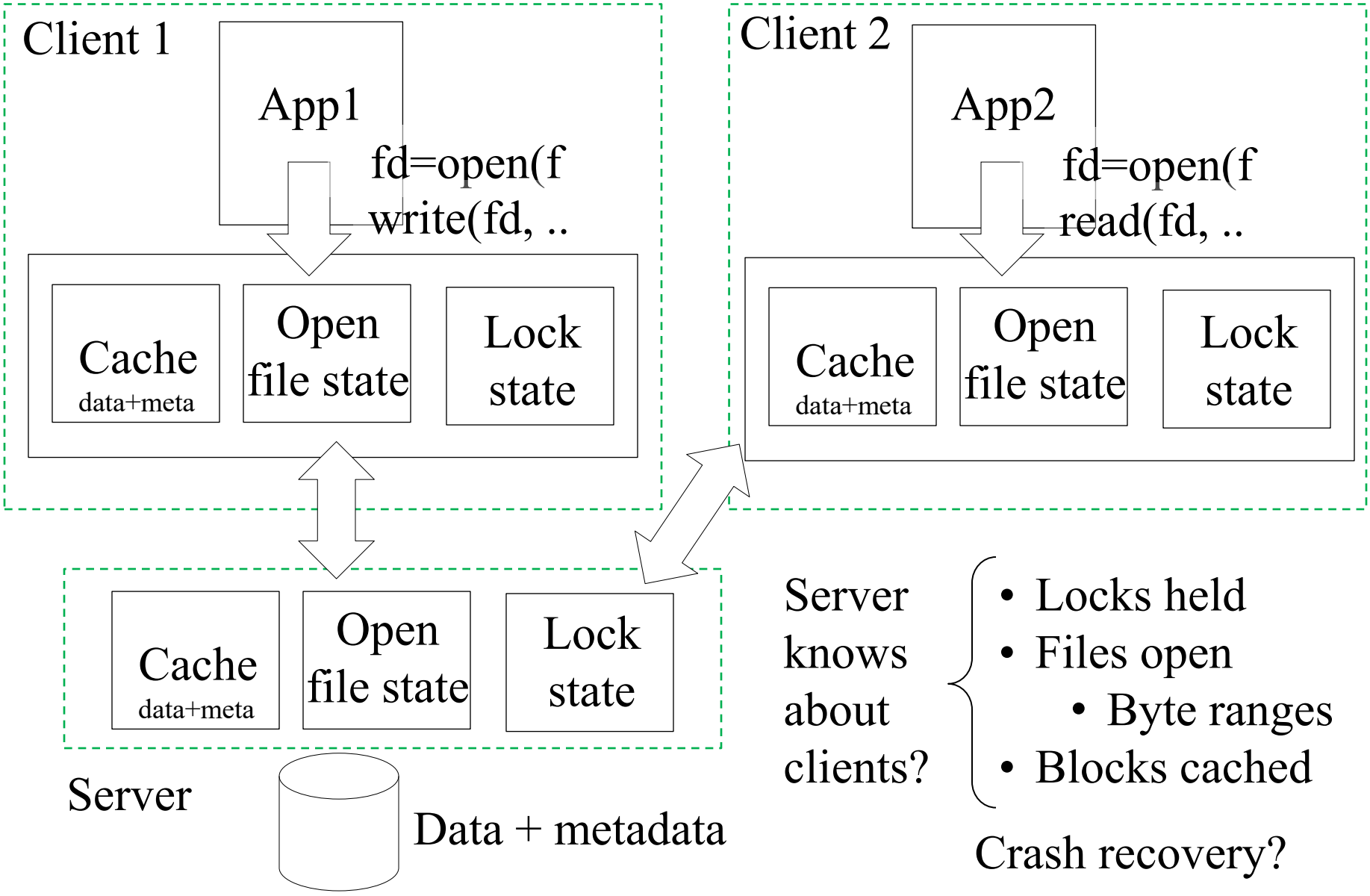


- Cache consistency
- Order (locking)
- Durability guarantee

  (when is data stable?)

To what extent do you involve the server?

Extending to a distributed setting



Server knows about clients?

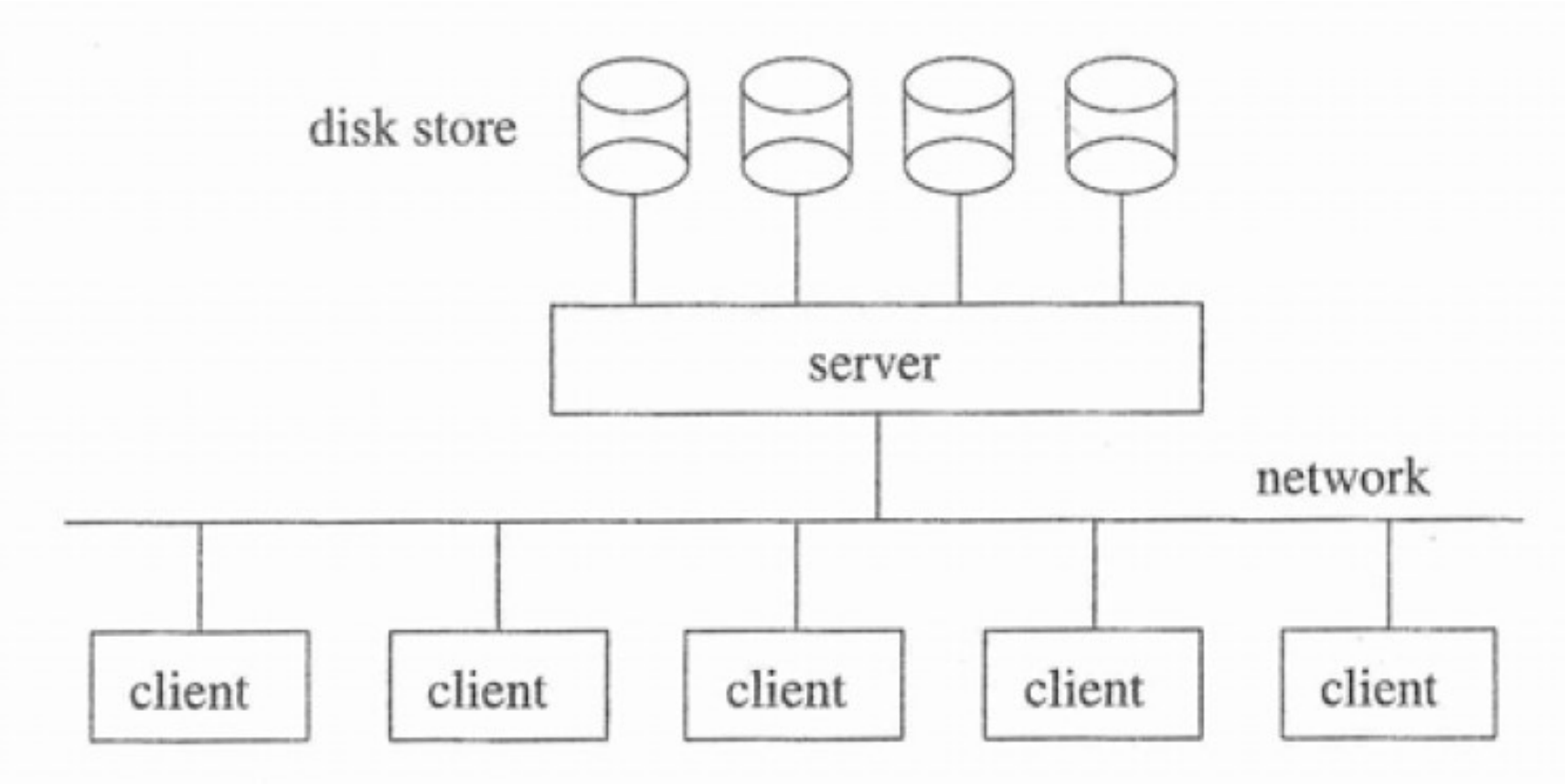
- Locks held
- Files open
  - Byte ranges
- Blocks cached

Crash recovery?

Network File System (NFS)

- History
  - UNIX United
  - SUN Network Disk
  - RFS
  - Andrew File System (AFS)
- Overview of NFS
  - Early editions (v2) aimed for statelessness
    - Server maintains no other information except for actual fs data
    - Writes are stable per-operation (stable store requirement)
    - Everything goes through server (no client caching)
  - Aims to offer UNIX semantics
  - Transport independent
  - UNIX security and access control

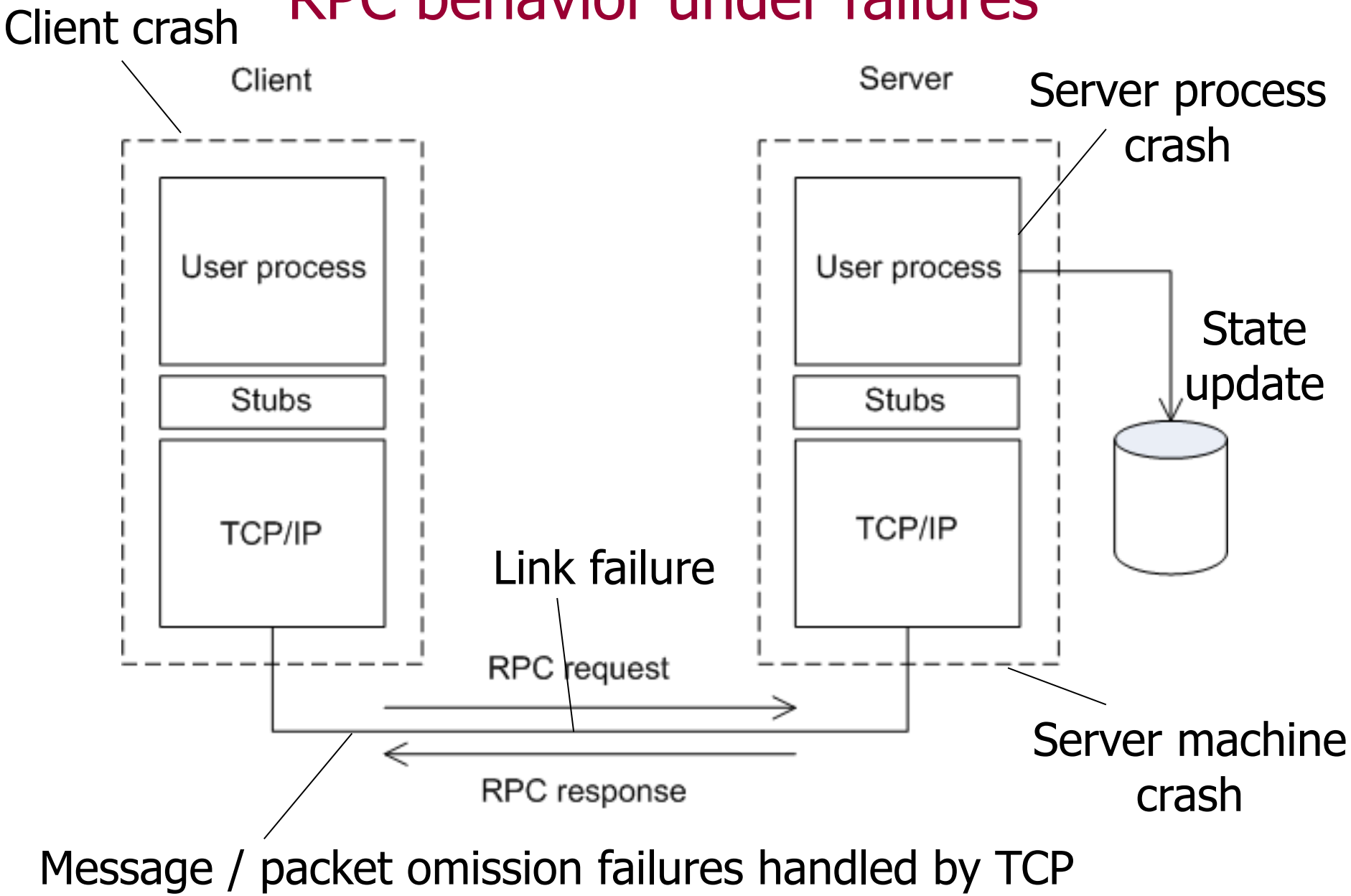
NFS division between clients and server



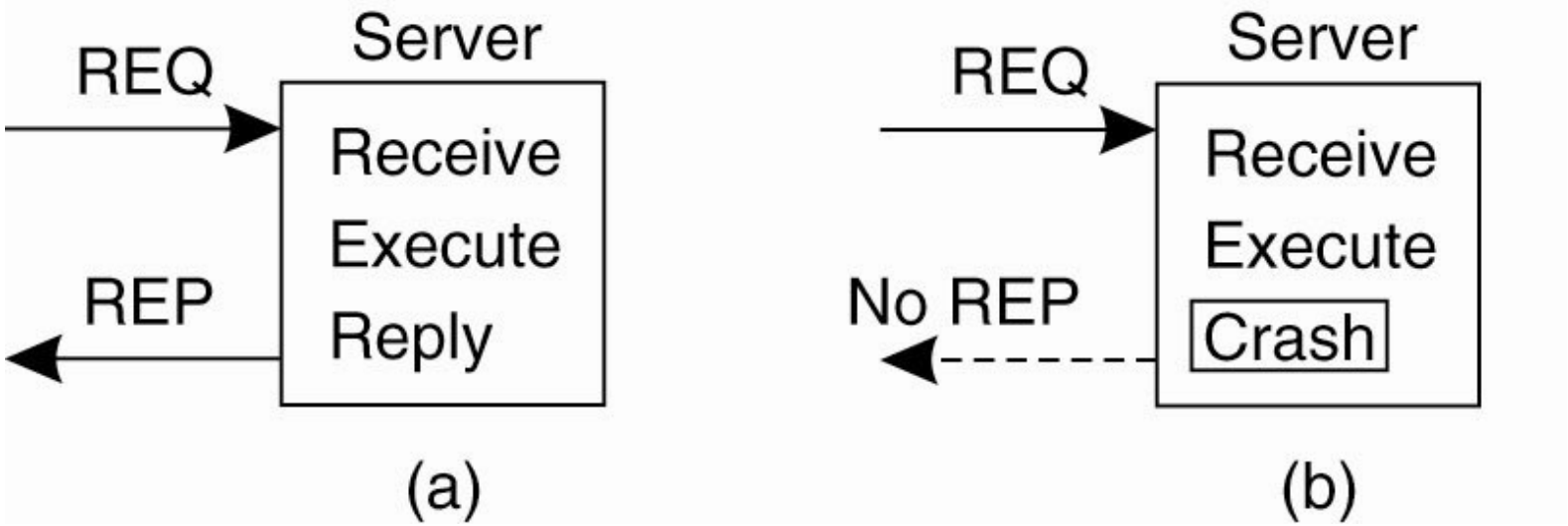
NFS structure and operation

- Based on Remote Procedure Calls (RPCs)
  - Handle problems that may occur due to crashes
- Files identified by NFS handle
  - Comprises inode id, file system id, generation number
- VFS/Vnode layer

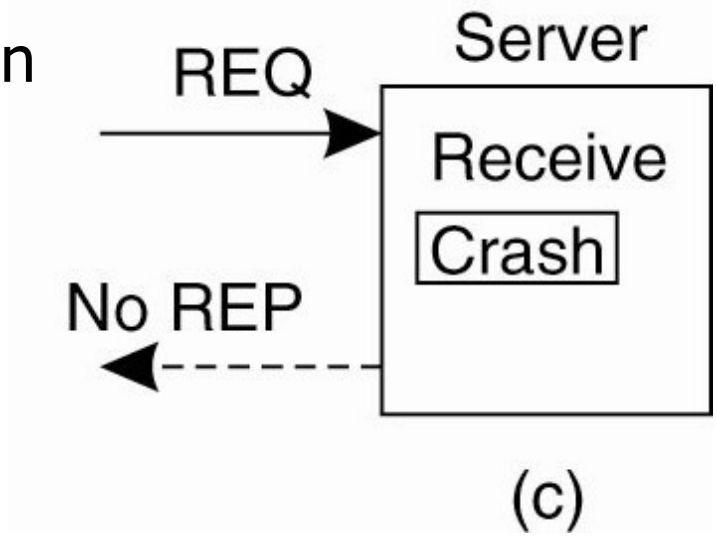
RPC behavior under failures



Server crashes



A server in client-server communication
(a) The normal case
(b) Crash after execution
(c) Crash before execution



RPC semantics

- At-least-once
  - Retry after an exception/timeout until successful
  - Good choice with idempotent operations (e.g., reads)
  - How about non-idempotent operations (e.g., writes)?

- At-most-once
  - Do not retry an operation or try to avoid duplicates

NFS Version 2 RPC requests

RPC request	Action	Idempotent
GETATTR	get file attributes	yes
SETATTR	set file attributes	yes
LOOKUP	look up file name	yes
READLINK	read from symbolic link	yes
READ	read from file	yes
WRITE	write to file	yes
CREATE	create file	yes
REMOVE	remove file	no
RENAME	rename file	no
LINK	create link to file	no
SYMLINK	create symbolic link	yes
MKDIR	create directory	no
RMDIR	remove directory	no
READDIR	read from directory	yes
STATFS	get filesystem attributes	yes

Stable-store requirement (NFS v2)

- All procedures in NFS v2 are synchronous
- When a procedure returns to the client, it assumes that the operation has completed and any data associated with the request is now on stable storage
- If it didn't, the client would have to save the data and retransmit it to the server, if the server crashed
- A WRITE may cause the server to update data blocks, indirect blocks, attributes (size, modify times)

Statelessness

- Server maintains no (recoverable) session state
  - Client IDs, open files, locks, cached blocks, RPCs in progress, etc.
- Pros
  - Simpler server design
  - Easier recovery
- Cons
  - Need session state to correctly implement file system API
  - Bad performance
    - Too expensive to be writing each request to server disk
    - Need to be able to read from client cache
    - Addressed by NFS v3 (stability per session, not per op)

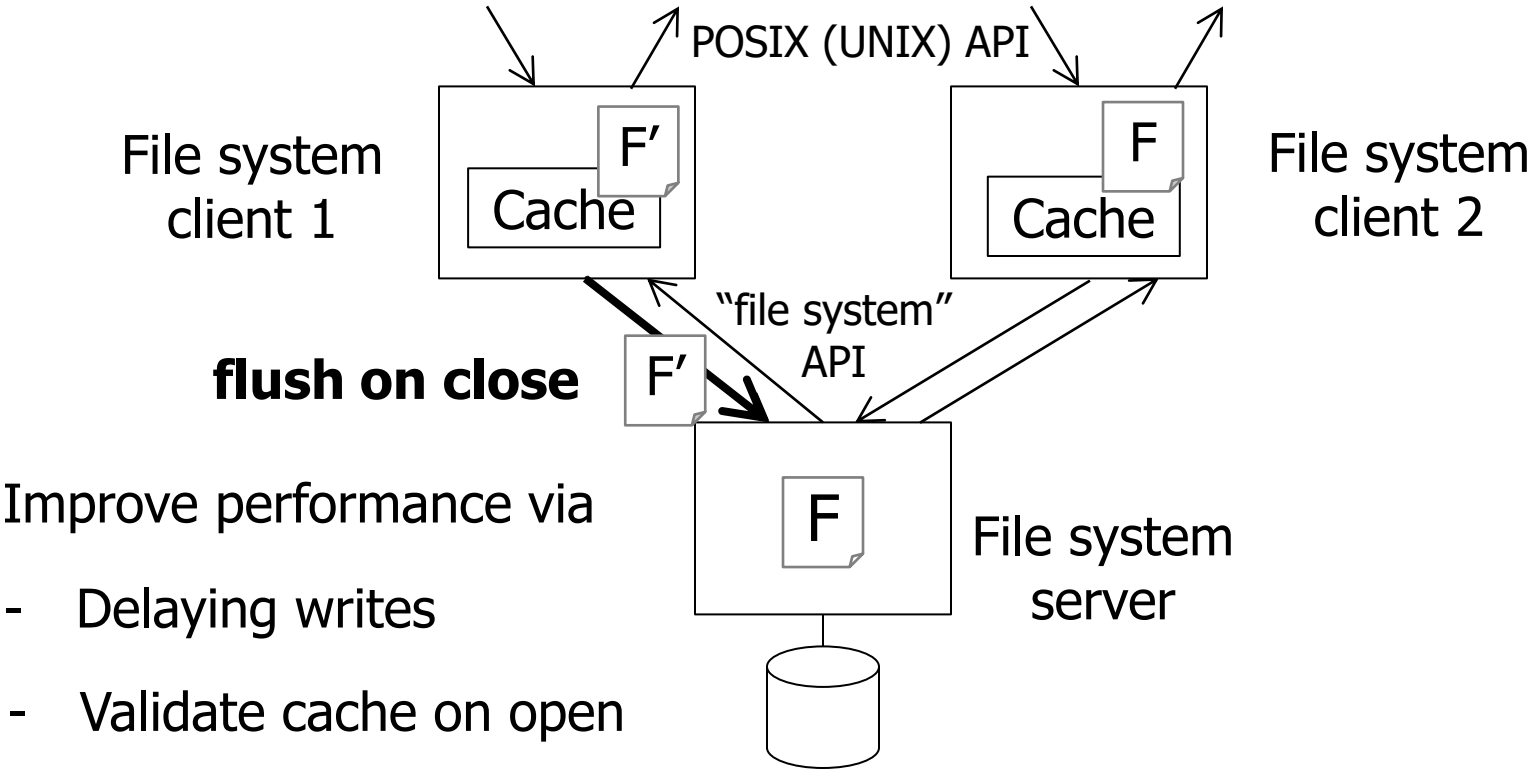
NFS v3: close-to-open consistency

Application 1

```
fd = open(F)
lseek(fd, 128, SEEK_SET)
write(fd, buf, 16384)
close(fd)
```

Application 2

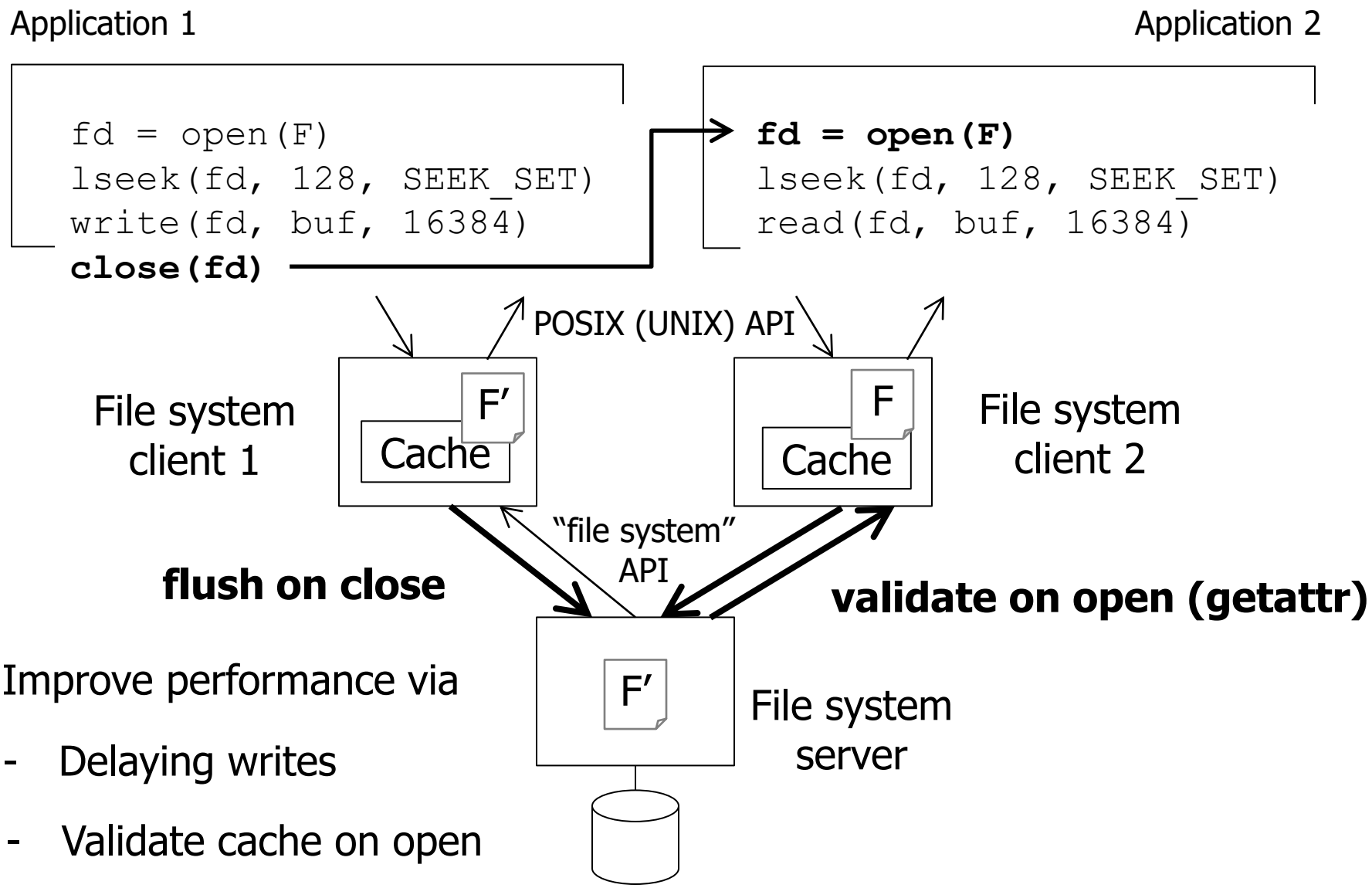
```
fd = open(F)
lseek(fd, 128, SEEK_SET)
read(fd, buf, 16384)
```



Improve performance via

- Delaying writes
- Validate cache on open

NFS v3: close-to-open consistency



Improve performance via

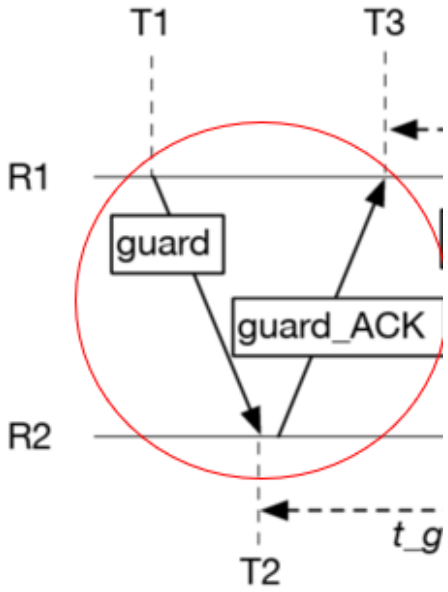
- Delaying writes
- Validate cache on open

Comparing network file systems

- NFS version 3
  - Weak consistency (close-to-open), relies on client applications being written for this consistency model
- Sprite [Nelson88]
  - File locks
  - Disable caching when detecting concurrent writes
- AFS [Howard88]
  - File locks
  - Callback-based invalidations

Background: Leases, fault tolerant locks

- Time-limited right to do something, a timed lock
  - Issued by a lease grantor, to lease holder
  - Leases expire **for both parties** if not renewed
- Examples
  - "You have the right to exclusively cache a file block"
- The holder should not believe it has the lease while the grantor thinks the opposite
  - Grantor should start the clock **after** the holder so that it expires after the holder!
  - Message exchange
    - guard: Start the clock @Holder
    - guard_ACK: Start the clock @Grantor
- *What if ACK is never received?*
  - Holder believes it has lock while grantor does not

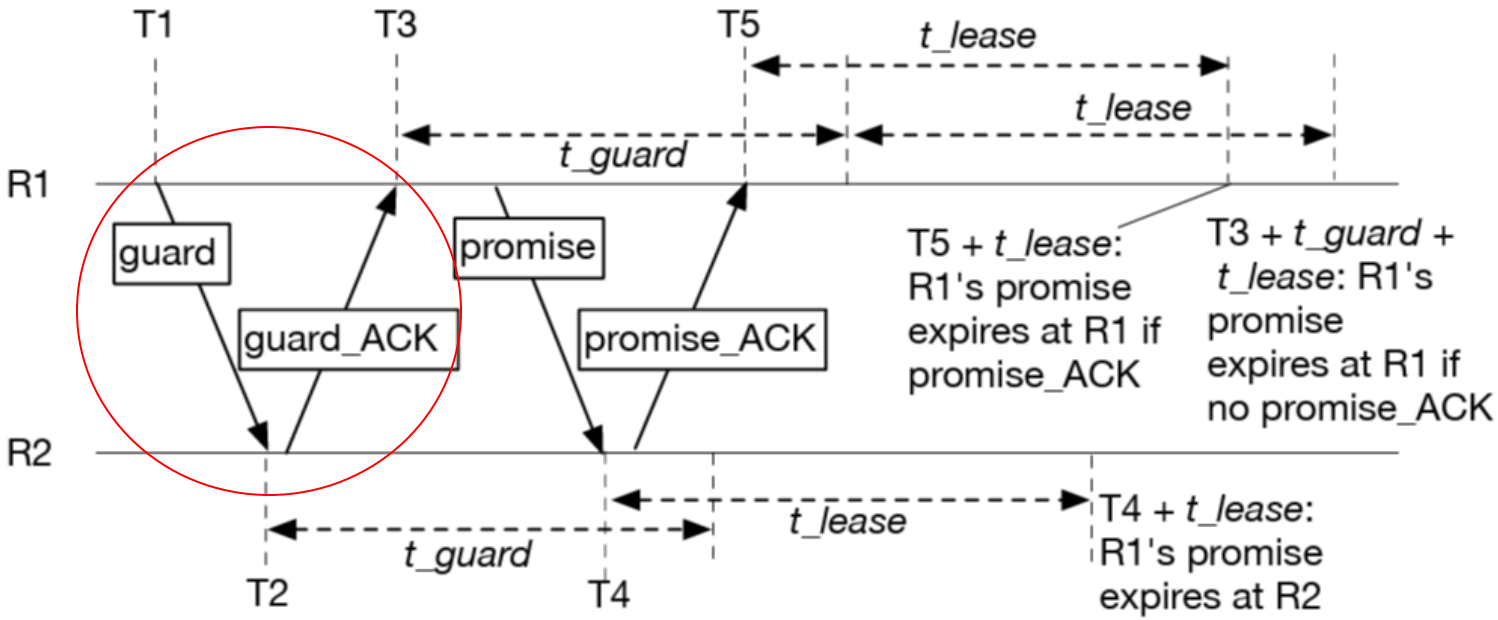


Granting a lease

Get into promise, only if guard is successful!

- Grantor guaranteed to start lease timer after holder starts its own (either as a result of message received or pre-timer expiring)

Replicas do not synchronize clocks, but their clock rates are assumed to be similar, such that a modest guard time can account for clock drift over a short interval.



The guard specifies a time duration t_guard. The subsequent promise contains a lease duration t_lease. Importantly, **the promise is only valid if received by the holder before t_guard has expired**. This ensures that even if the holder does not respond to the promise, the grantor knows that the holder will not believe it has a lease subsequent to t_guard+t_lease seconds after it received the guard ACK

Figure courtesy of I. Moraru et al. "Paxos Quorum Leases: Fast Reads Without Sacrificing Writes", SoCC'14, https://doi.org/10.1145/2670979.2671001

Renewing a lease

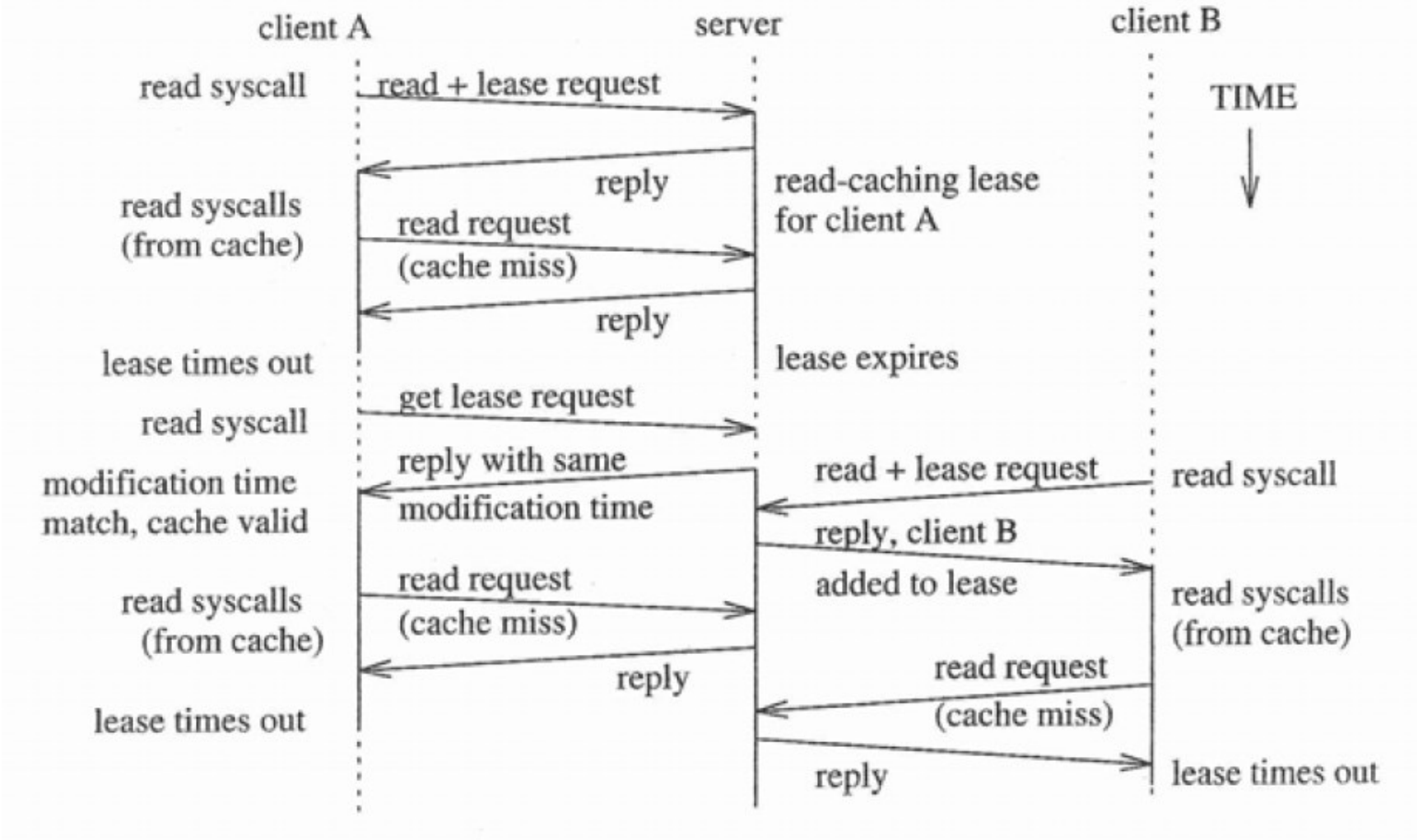
- When renewing active leases, there is no need to send the guard anymore
- The most recent acknowledged promise plays the role of the guard:
  - when sending a new promise, the grantor indicates that it must be received within a time t0+t_guard from the most recent received acknowledgment (the grantor indicates which ACK this was), where t0 is the time elapsed at the grantor since receiving this acknowledgment
- Therefore, the grantor will be able to safely relinquish its promise after t_guard + t_lease seconds from sending the renewal.

Using leases for consistent fault-tolerant client file caching in NFS

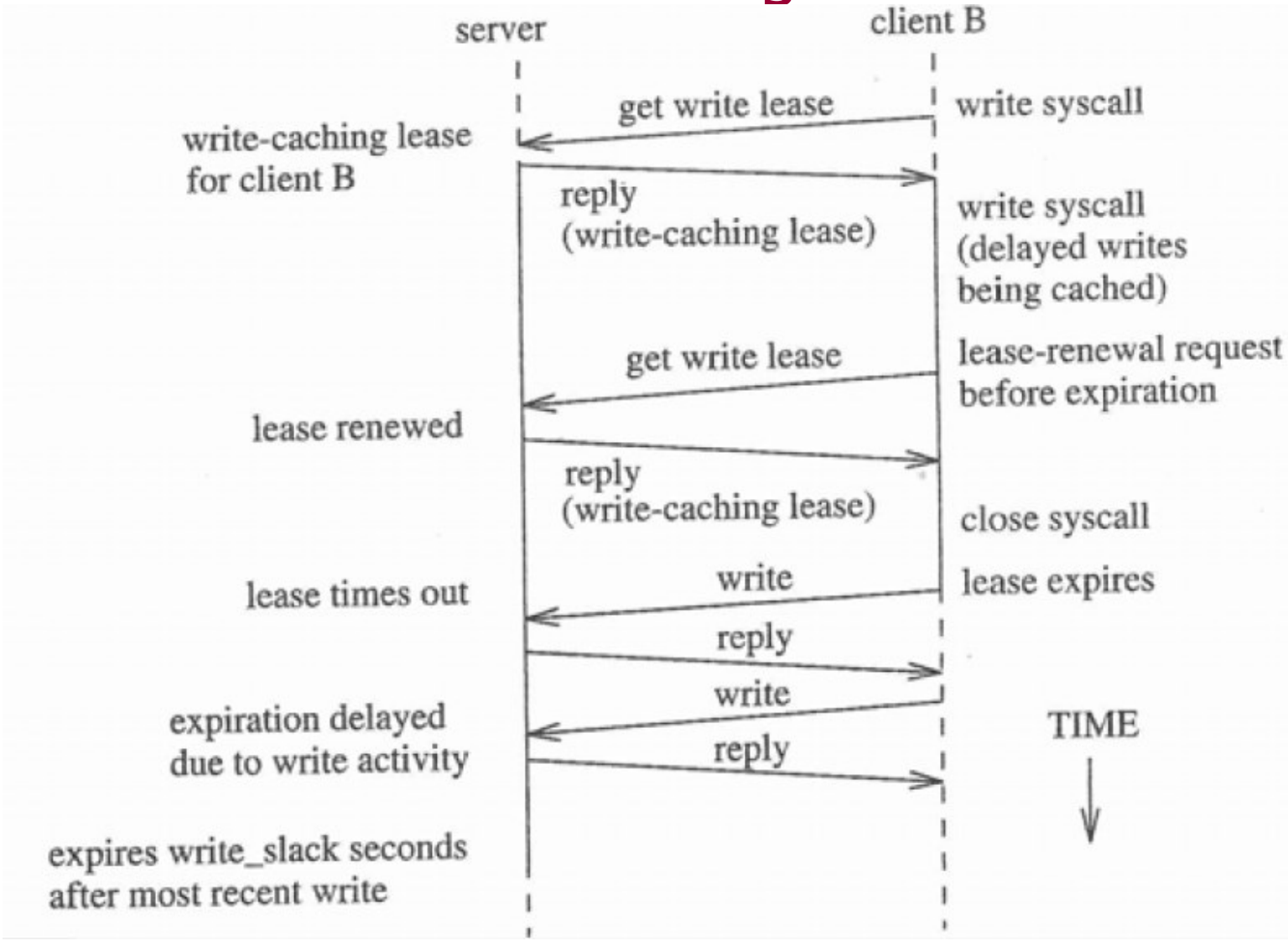
- Pros
  - Quick recovery without requiring hard state
  - Can tolerate partitions
  - Require moderate amount of soft state

- Important constants
  - `maximum_lease_term` (normally ~30”)
  - `clock_skew`
  - `write_slack`

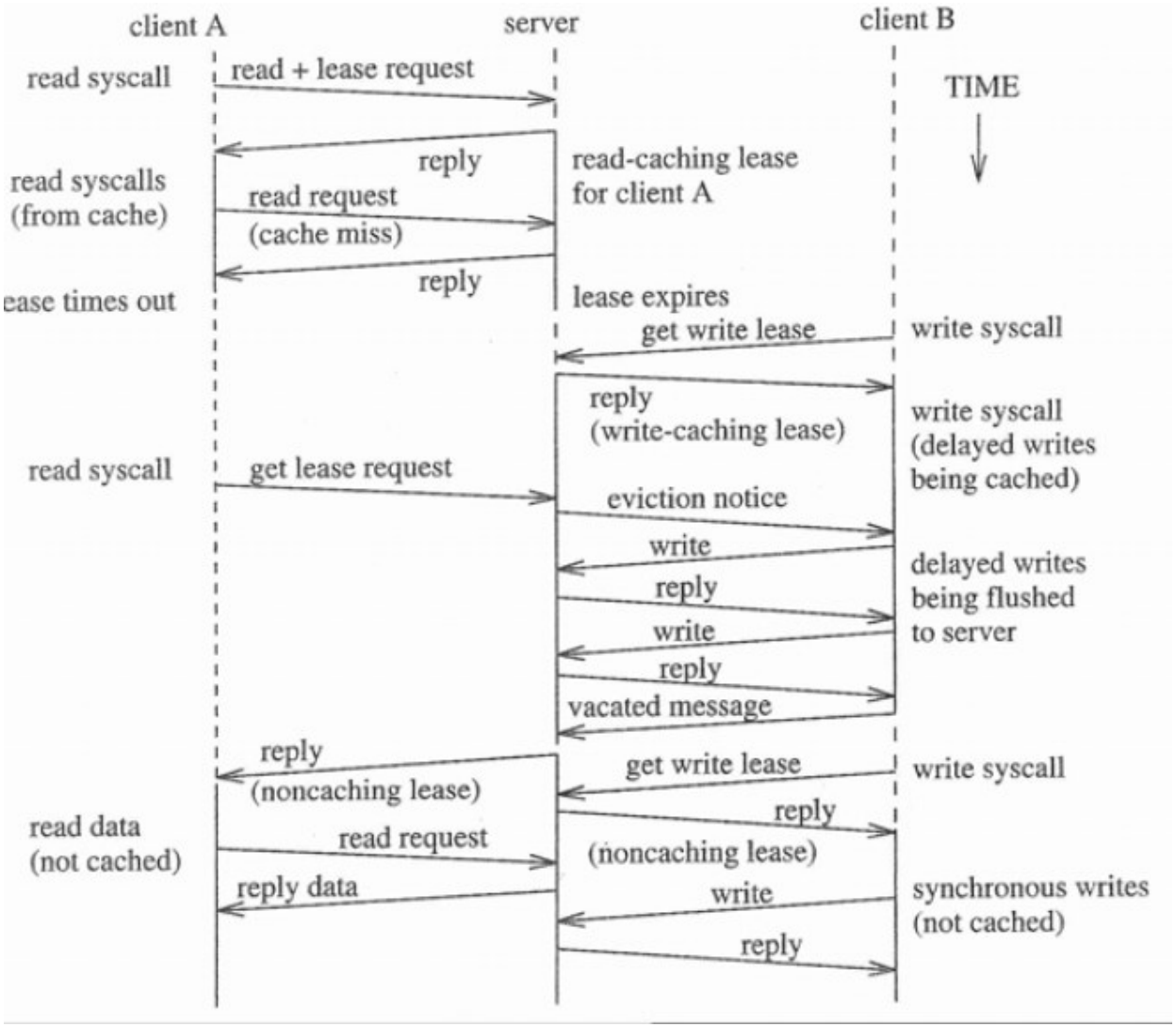
Read-caching leases



Write-caching lease



Write-sharing leases



Recovery

- `maximum_lease_term` seconds after the server stops issuing leases, there should be no leases left
- After rebooting, server accepts writes for up to `write_slack` seconds after final lease expired
- Server that do not know when they crashed estimate final-lease expiration time by adding up
  - `boot_time`
  - `maximum_lease_term`
  - `write_slack`
  - `clock_skew`