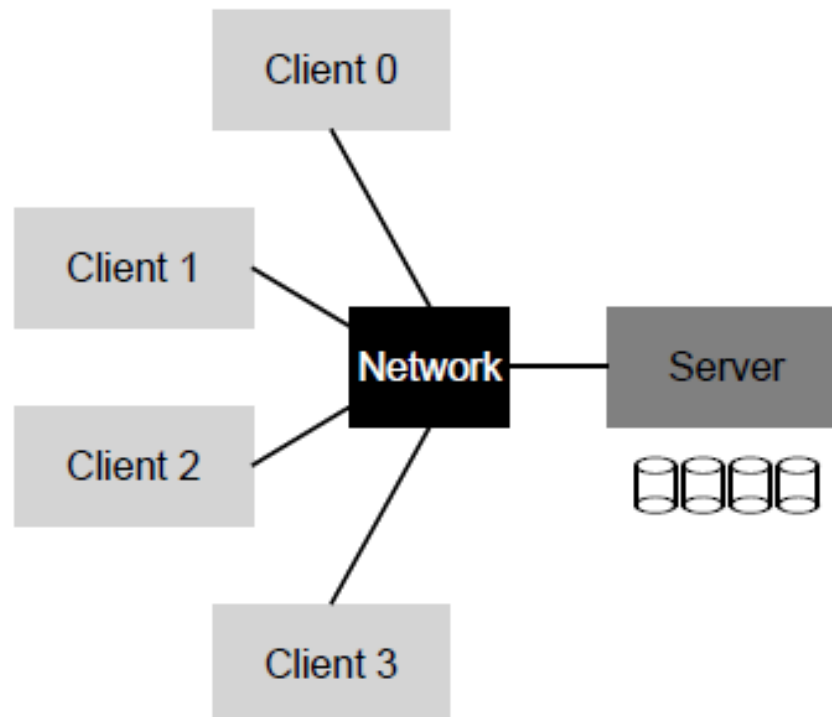


Network File Systems

INF 551

Wensheng Wu

Client/server architecture



Advantages

- Easy sharing
 - Of data across clients
- Centralized administration
 - E.g., backup done on server, instead of individual clients
- Security
 - Server is located in a locked room

Disadvantages

- Network overhead
- More components to fail

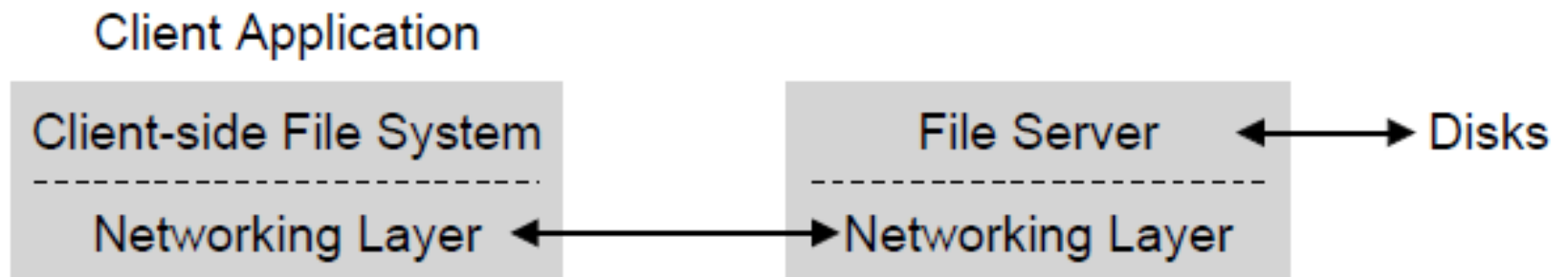
A basic distributed file system (DFS)

- Client/server architecture
- Applications interact with client-side file system (FS)
 - Issue system calls (i.e., calls that request services from OS kernel)
 - E.g., `open()`, `read()`, `write()`, `rmdir()`, etc.
 - Same interface as standalone file system

Transparent remote access

- Same operations but access remote files
 - Details of accessing are made transparent to clients
- Read()
 - Client-side FS sends a message to server-side file system (file server): read block xyz
 - File server reads the block from cache/disk
 - Server sends a message back to client with data

DFS architecture



Sun's NFS (network file system)

- Example of DFS
- Open protocol
 - Specifies only the format of messages btw client and server
 - Permits different implementations
 - Enhances interoperability among different vendors
- NFSv2
 - Basis for later versions
 - Focus: simple and fast server crash recovery

Key: statelessness


- Server does not keep track of states of clients
 - Which clients have read/cached which blocks
 - Which files are currently open at which clients
 - Current position/offset of file
- Requests from clients must make sure:
 - the server can deliver all the information needed to complete the requests
 - & do not rely on previous requests

Stateless file handle

- Contains 3 parts (note it is richer than fd)
 1. Volume identifier
 - Which volume? (e.g. partition C or D if NTFS)
 2. Inode number
 - Which file in the volume?
 3. Generation number
 - Needed since inode number may be reused at the server

Client-side file open table

File handle



Server	File descriptor	File system (volume)	File name	Inumber	Generation #	Position offset	Status	...
Server1	3	vol1	"/foo"	32382	1	4096	open	
Server1	4	vol2	"/foo/more"	48482	1	0	close	...

Client uses file handle to communication with server

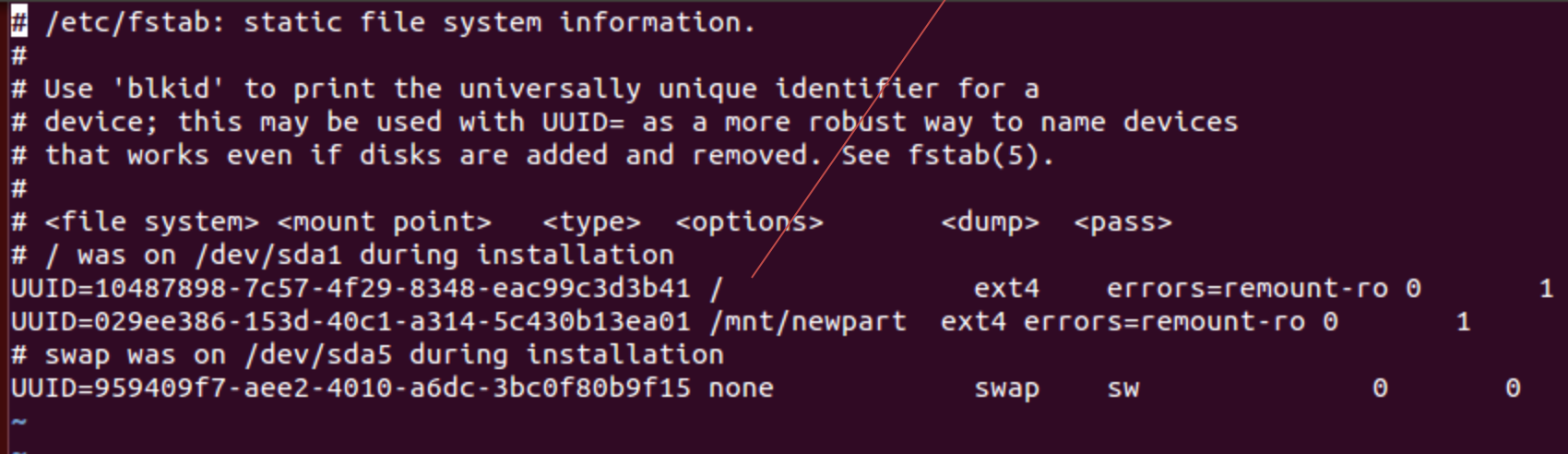
Obtain file handle via lookup

- Lookup
 - Input: parent directory file handle + name of file/directory to look up
 - Output: file handle of lookup file/directory
- E.g., `lookup(root file handle, "foo.txt")`
 - obtain handle of `"/foo.txt"`
- File handle for the root directory may be obtained via the **mount** protocol

File system table in Linux

- Each line represents a file system to be mounted when machine starts
 - UUID is the ID of file system
- Each FS gets mounted to a different part of directory tree

Root directory



```
## /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point> <type> <options> <dump> <pass>
# / was on /dev/sda1 during installation
UUID=10487898-7c57-4f29-8348-eac99c3d3b41 / ext4 errors=remount-ro 0 1
UUID=029ee386-153d-40c1-a314-5c430b13ea01 /mnt/newpart ext4 errors=remount-ro 0 1
# swap was on /dev/sda5 during installation
UUID=959409f7-aee2-4010-a6dc-3bc0f80b9f15 none swap sw 0 0
~
~
```

Remote procedure call (RPC)

- Remote server publishes a set of procedures
 - E.g., $f(\text{args})$
- In making RPC calls,
 - Client notifies remote server of executing f
 - Client sends over arguments args for f
 - Server executes $f(\text{args}) \Rightarrow \text{results}$
 - Server sends back results

RPC in NFS

- NFS server publishes a set of RPCs
 - E.g., `NFSPROC_LOOKUP` for lookup file handle
 - Others include: read, write, create, remove, etc.

`NFSPROC_LOOKUP`

expects: directory file handle, name of file/directory to look up
returns: file handle

Lookup with long path

- Multiple lookup requests needed for long path
 - One component of path at a time
- Avoid additional parsing if sending over separators
 - plus different OS's may have different separators
 - e.g., "/" (Linux) or "\" (Windows)

Example

- Look up `"/foo/more/bar.txt"`
 - First, use `/` file handle to obtain `foo`'s handle
 - Next, use `foo`'s handle to obtain `more`'s handle
 - Finally, use `more`'s handle to obtain `bar.txt` handle


CRUD

- All CRUD operations use file handle
 - Instead of file descriptors as in local file system

Read

Explicit offset



- NFSPROC_READ(file handle, **offset**, count)
 - Return: data + file attributes
 - File attributes include modification time, useful for client-size cache validation
 - Compared to local FS
 - $n = \text{read}(\text{fd}, \text{buffer}, \text{size})$ 
 - n is the number of bytes actually read
- Offset is implicit here (current location)
 - Buffer is provided

Write

- NFSPROC_WRITE(file handle, **offset**, count, data)
 - Return: file attributes
 - **Note again explicit offset is specified in the call**
- Compared to local FS
 - $n = \text{write}(\text{fd}, \text{buffer}, \text{size})$
 - **Offset is again implicit (current position)**

Create and remove files

- NFSPROC_CREATE(directory file handle, name of file in the directory, attributes)
 - Return file handle (note difference from reading material, see more at:
<https://tools.ietf.org/html/rfc1094>)
- NFSPROC_REMOVE(directory file handle, name of file to be removed)
 - Return nothing

Working with directories

- NFSPROC_MKDIR & NFSPROC_RMDIR
 - Similar to create & remove files
 - but create & remove a directory instead
- NFSPROC_READDIR
 - Similar to read file, but here read directory content

Get/set attributes of files

- GetAttr
 - Obtain file attributes, e.g., last modified time
 - Important for client-side caching

NFSPROC_GETATTR

expects: file handle

returns: attributes

NFSPROC_SETATTR

expects: file handle, attributes

returns: nothing

Example: opening a file for read

1. Obtain file handle + client-side book-keeping

Client

Server

```
fd = open("/foo", ...);  
Send LOOKUP (rootdir FH, "foo")
```

```
Receive LOOKUP request  
look for "foo" in root dir  
return foo's FH + attributes
```

```
Receive LOOKUP reply  
allocate file desc in open file table  
store foo's FH in table  
store current file position (0)  
return file descriptor to application
```

Example: reading the file

2. Start to read

- Obtain file handle & offset from client-side open table
-

`read(fd, buffer, MAX);`

Index into open file table with fd

get NFS file handle (FH)

use current file position as offset

Send READ (FH, offset=0, count=MAX)

Receive READ request

use FH to get volume/inode num

read inode from disk (or cache)

compute block location (using offset)

read data from disk (or cache)

return data to client

Receive READ reply

update file position (+bytes read)

set current file position = MAX

return data/error code to app

Example: reading a file

3. Continue to read

4. Done and clean up

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

Same except offset=MAX and set current file position = 2*MAX

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

Same except offset=2*MAX and set current file position = 3*MAX

```
close(fd);
```

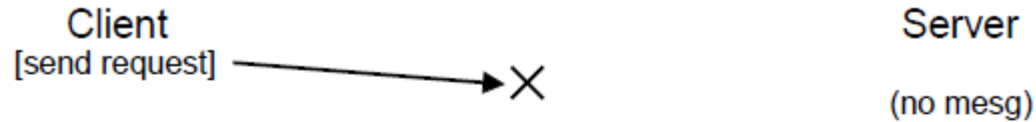
Just need to clean up local structures

Free descriptor "fd" in open file table

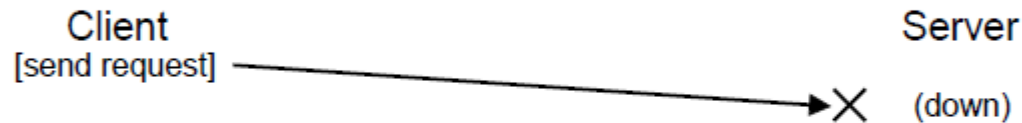
(No need to talk to server)

Deal with failures

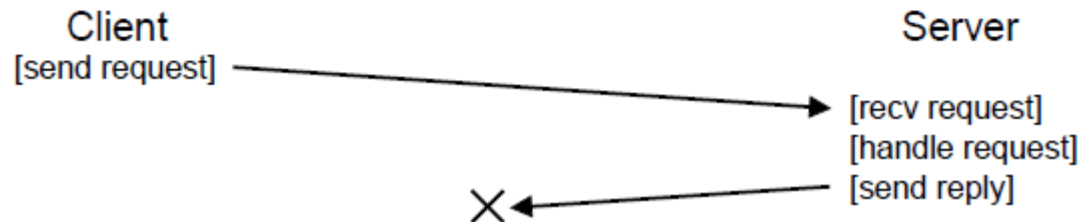
Case 1: Request Lost



Case 2: Server Down



Case 3: Reply lost on way back from Server



Idempotent operations

- Does not matter how many times you execute
 - Effect is the same as single execution
- All these **common** operations are idempotent
 - Lookup
 - Read
 - ReadDir
 - Write: since it specifies the exact offset

Power of idempotent operations

- Simplify handling of failure
- Client sets timer, when time-out but no reply
 - Simply retry the same request

Non-idempotent operations

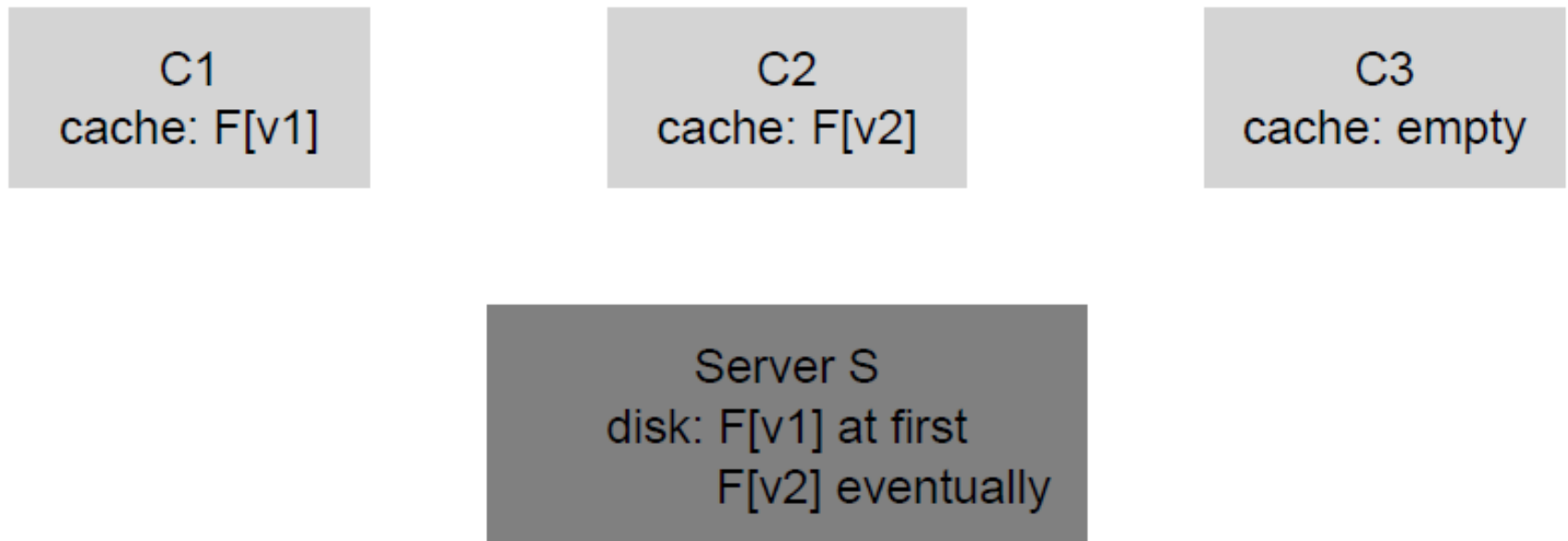
- These operations are not idempotent:
 - Create (file)
 - Remove (file)
 - Mkdir (create directory)
 - Rmdir
- Error message will return from server
 - E.g., when create is executed more than once

Improving performance

- Client side
 - Read caching
 - Write buffering
- Similar to standalone file system
- Unique change: cache consistency problem
 - Due to multiple caches in several clients

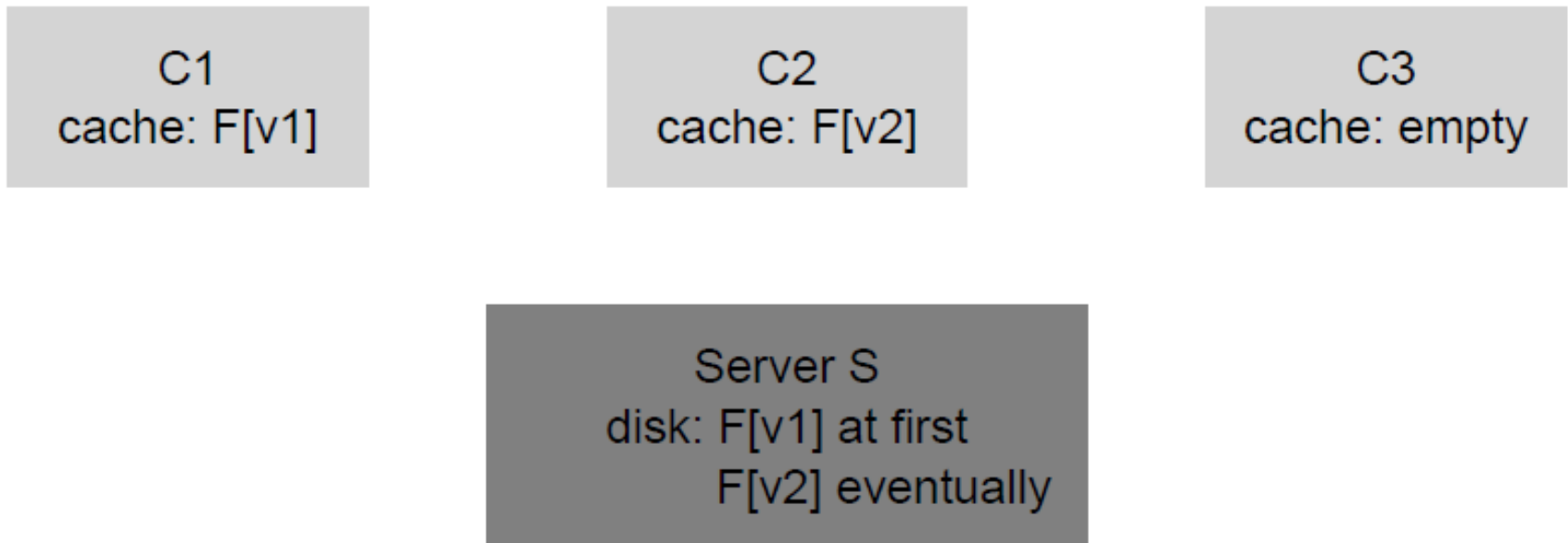
Cache consistency problems

- C2 updates file F to version 2: F[v2], but does not commit it to server when it closes the file
- Problems: update visibility & stale cache



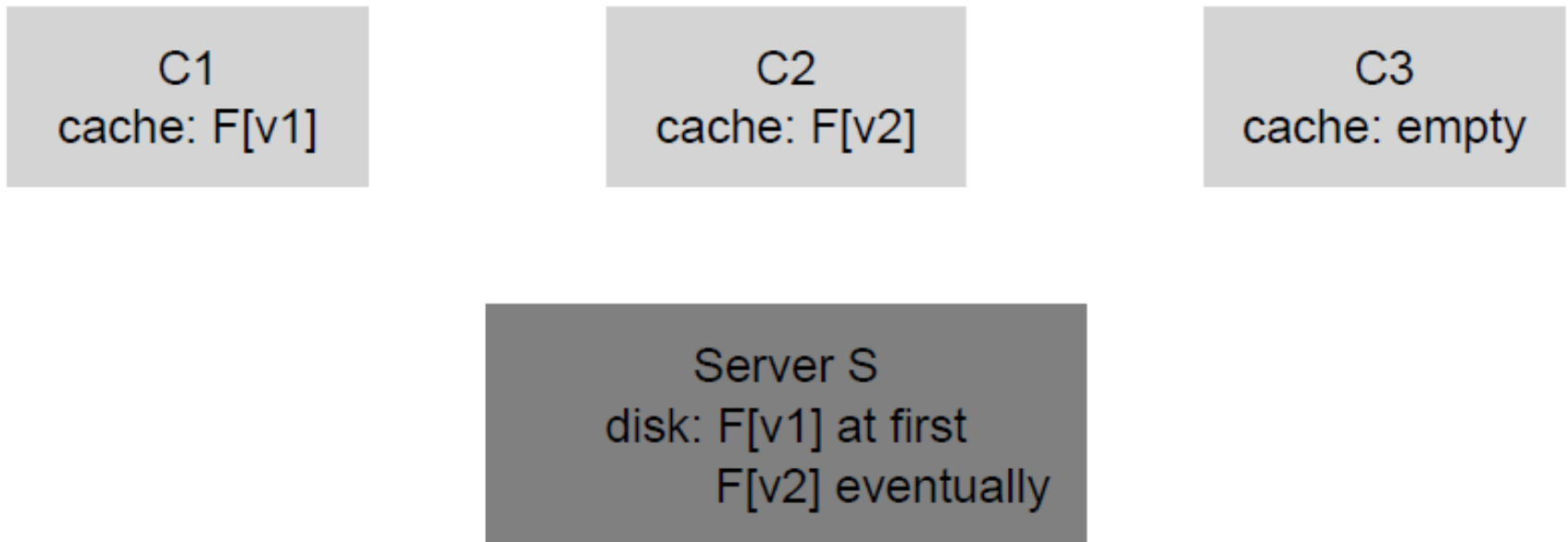
Update visibility problem

- C3 reads F from server, got old content



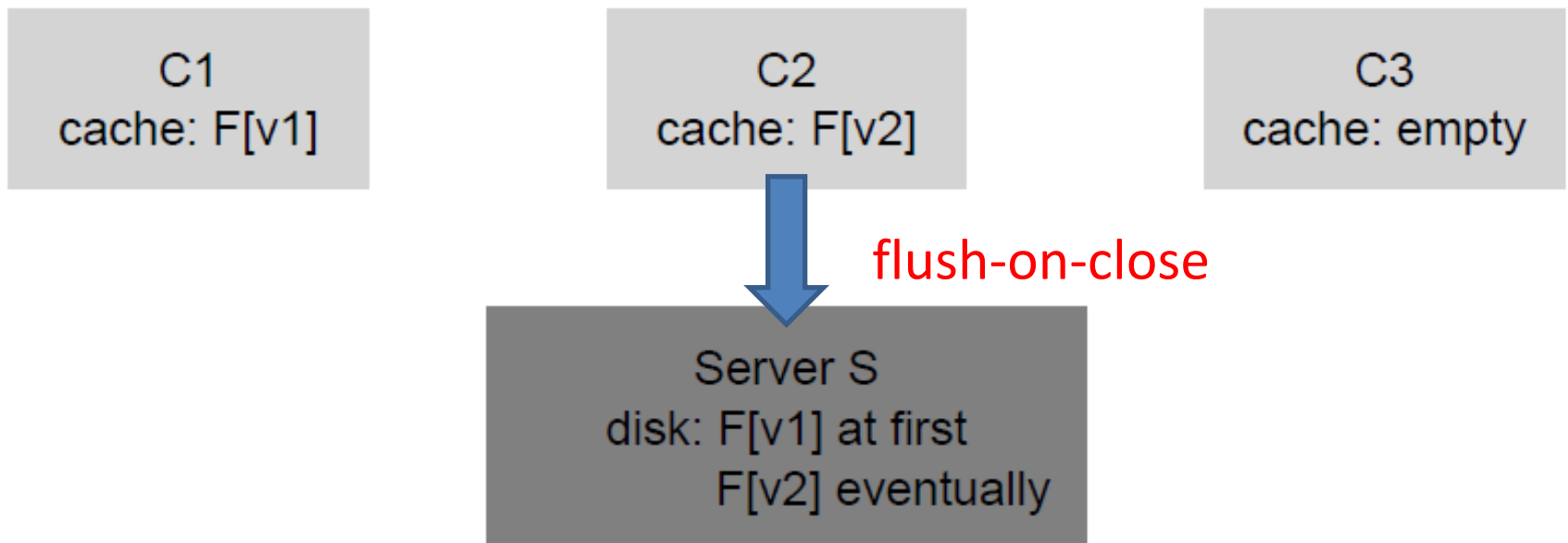
Stale cache problem

- $F[v2]$ finally flushed to server, but now caches C1 (& C3) are stale



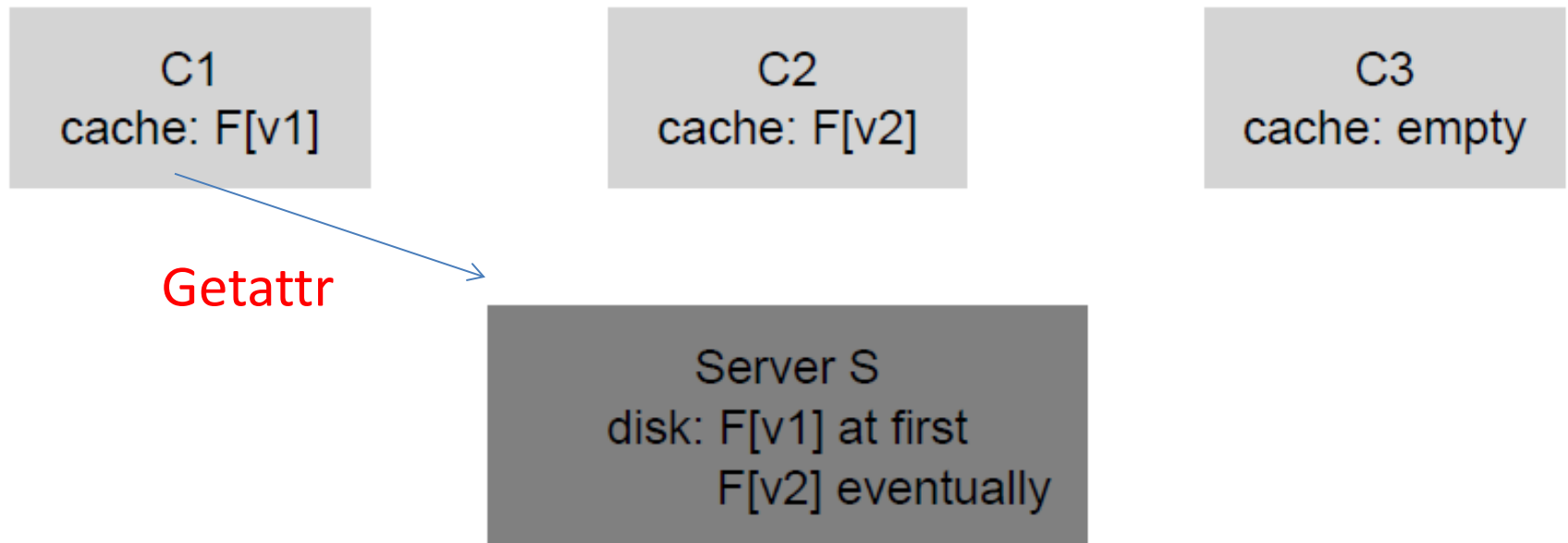
Solution: flush-on-close

- Client flushes all updates to server when file is closed
- Solve the **update visibility** problem



Solution: cache validation

- Client checks if cache is current by issuing GETATTR to server before opening/accessing the file



References

- NFS: Network File System Protocol Specification
 - <https://tools.ietf.org/html/rfc1094>