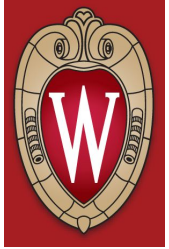
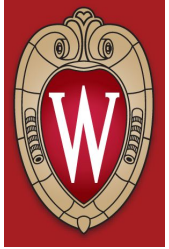


CS 739 Distributed Systems

Project 2 : Client/Server File System

Group 10
Members:
Cody Tseng
Do Men Su
Vyom Tayal
Chahak Tharani



Part 1 : Functionality & Correctness

1.1, 1.2 POSIX Compliance & AFS Protocol

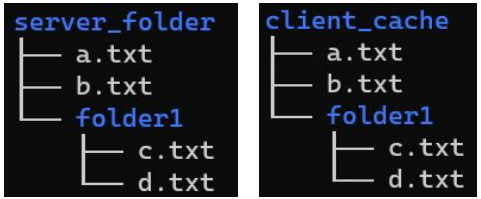
Whole file cache, use local copy when write and read (Do not cache directory information)
Flush on close & fsync, last writer wins
Check modified time during open
(Cache technically staled if other client update the file, but become un-staled once open is called)

POSIX	FUSE equivalent	RPC
open()	open()	getattr() download()*
creat()	create()	create()
unlink()	unlink()	unlink()
mkdir()	mkdir()	mkdir()
rmdir()	rmdir()	rmdir()
stat()	getattr()	getattr()

POSIX	FUSE equivalent	RPC
read()/ pread()	read()	X (Use local)
write()/pwrite()	write()	X (Use local)
close()	flush()	upload()
	release()	X (Might not be called in the case of fork())
fsync()	fsync()	upload()
rename()	rename()	rename()

Cache folder structure

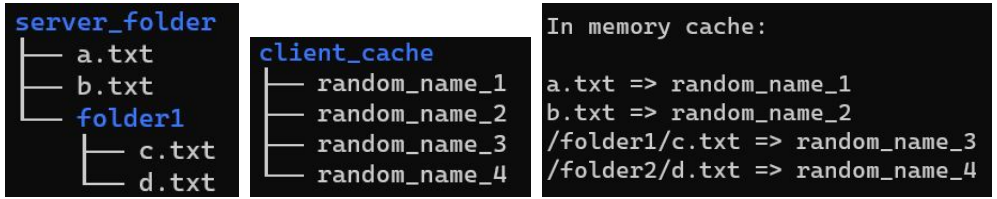
Mirror



Issue #1: Needs to make all parent directories when opening file and remove all empty parent directories when unlink

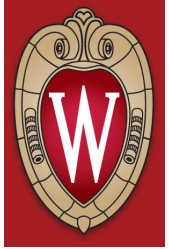
Issue #2: Client might not use the same file system as the server and might be impossible to name a file exactly the same

Flat + mapping



Issue: Either somehow persist the in memory cache every operation or give up the entire cache after reboot/crash

Cache folder structure (contd.)



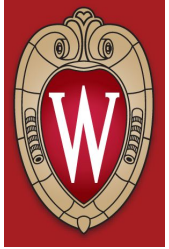
Flat + Hashing the pathname

```
server_folder
├── a.txt
├── b.txt
└── folder1
    ├── c.txt
    └── d.txt
```

```
client_cache
├── 18b7cb099a9ea3f50ba899b5ba81e0d377a5f3b16f8f6eeb8b3e58cd4692b993
├── 291f4e016dbf9d689e1b3ef9cb492662e1cccef9f61f524bb15b0f58326613bd
├── 8b4e065549e1618388fac5e4145bcf4bdb60a41b2c03daeac8391381b5a03a4b
└── ffa0da5d885fba09d903c782713b6b098c8cf21f56a3a35d9aa920613220d2e1
```

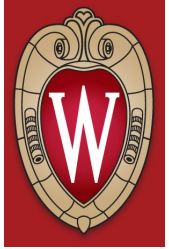
- Filename in cache = SHA256(pathname)
- Easy to manage
- No need to persist any mapping
- Need to sync the modify time for cache files with server

Read-Write sharing semantic in the same client



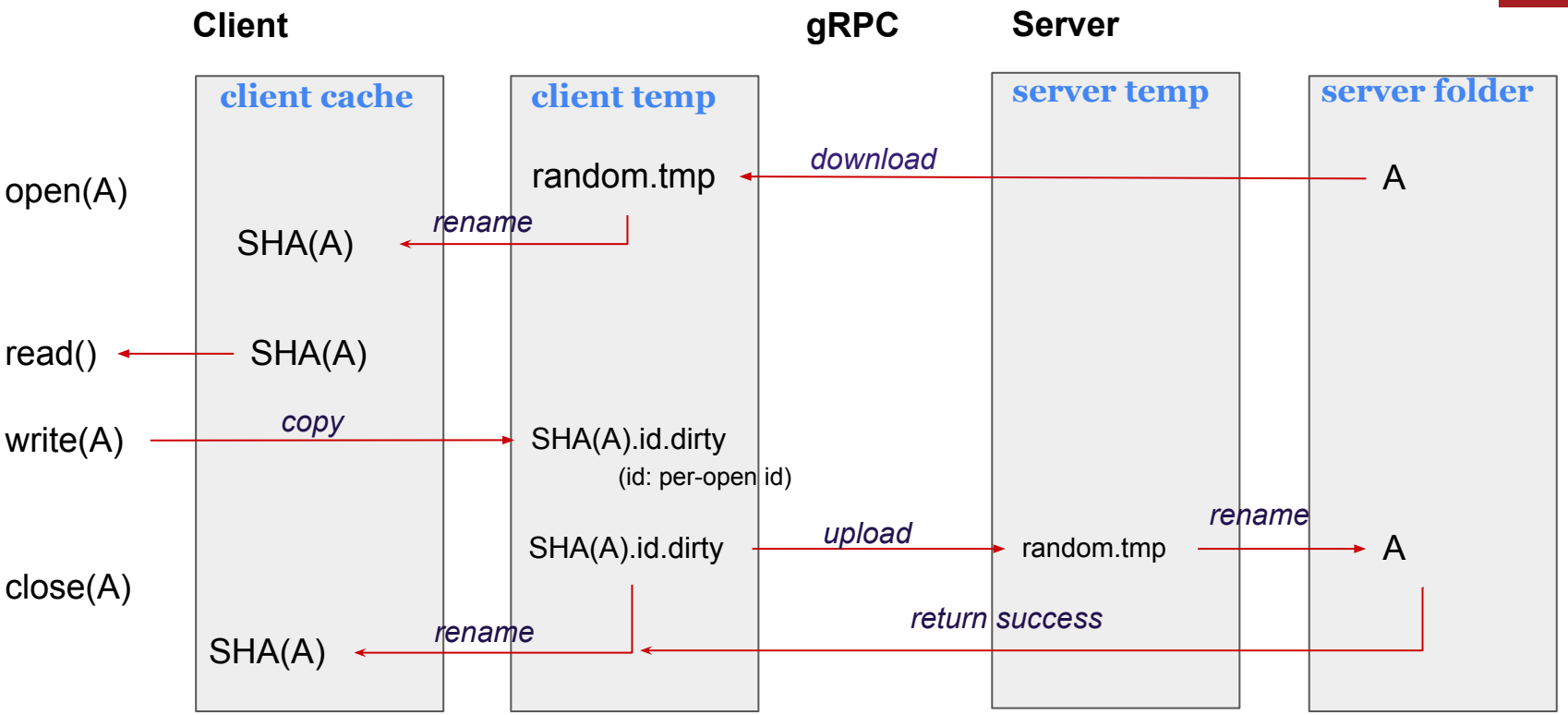
- Semantics
  - Write sharing (POSIX) vs Last writer wins (AFS)
  - Reader immediately see update (POSIX) vs see after writer flush and reader reopen (AFS)
- Results
  - Choose to align with AFS behavior
    - More consistent across the system
    - Experiments on CSL AFS show later result
- Imply that we need a copy of the file (dirty files) for every open and write
  - *(or journal but we use copy-on-write in our implementation)
  - Rename dirty file to the original file after flush

Dirty files handling after a crash



- Option 1
  - Option 1.1: Flush all latest dirty files to the server
    - Potentially long startup time
  - Option 1.2: Keep dirty files, flush on the next close()
    - A read of a file has a side effect of updating server data
  - Potentially flush corrupted data (Which might be OK?)
  - Persistence properties depend on client file system
    - Feel weird? Also we don’t want to force clients be on specific file system
  - Or make the dirty file consistent with another c-o-w or journal
    - Performance hit
- Option 2: Drop them
  - What we go with in the end

Design Summary



1.3 Durability - File system

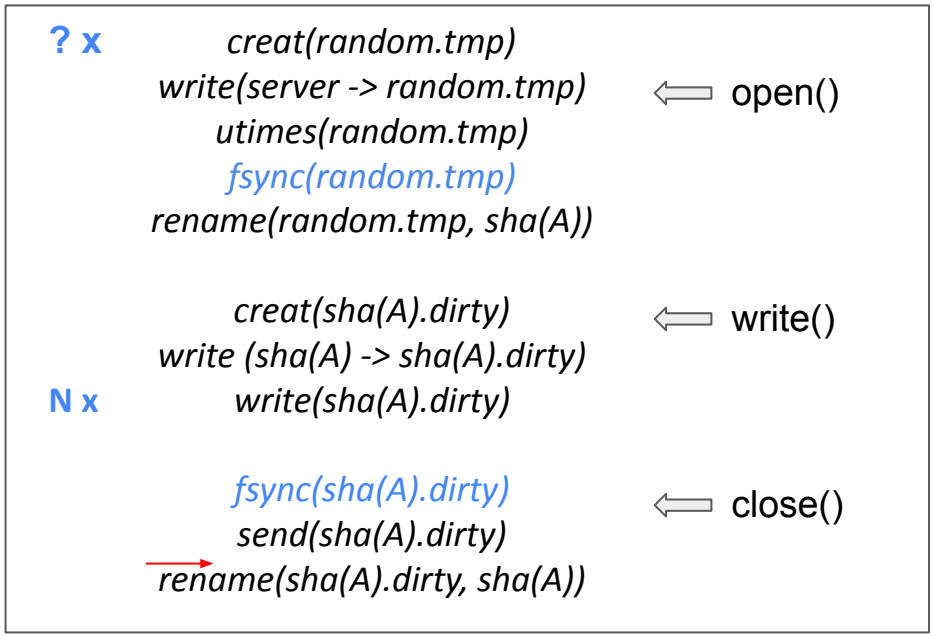
Persistence Property	File system														
	ext2	ext2-sync	ext3-writeback	ext3-ordered	ext3-datajournal	ext4-writeback	ext4-ordered	ext4-nodelalloc	ext4-datajournal	btrfs	xfs	xfs-wsync	reiserfs-nolog	reiserfs-writeback	reiserfs-ordered	reiserfs-datajournal
Atomicity																
Single sector overwrite																
Single sector append	×		×			×								×		
Single block overwrite	×	×	×	×		×	×	×			×	×	×	×	×	
Single block append	×	×	×			×							×	×		
Multi-block append/writes	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Multi-block prefix append	×	×	×			×							×	×		
Directory op	×	×											×			
Ordering																
Overwrite → Any op	×		×	×		×	×	×			×	×	×	×	×	
[Append, rename]→ Any op	×		×			×							×	×		
O_TRUNC Append → Any op	×		×			×							×	×		
Append → Append (same file)	×		×			×							×	×		
Append → Any op	×		×			×	×			×	×		×	×		
Dir op → Any op	×									×			×			

- Do not want to force client to use specific file system
- Insert fsync when necessary to prevent ordering issue
- Still need atomic directory op
  - Any FS that support atomic directory op should be find with our file system
- Use ext4-ordered in our benchmark

1.3 Durability - Open, Write, Close

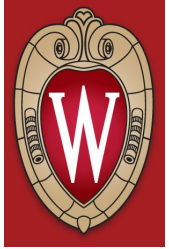
We guarantee atomic writes

- changes visible on client only after successful close()/fsync()
- may be visible on server and not on client (crash at red arrow) however, updated on client on next open
- rename ops are guaranteed by ext4



(i) open(), write() and close() in AAFS

1.3 Durability - Rename

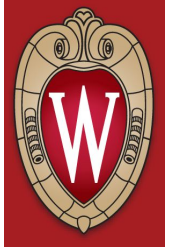


Final implementation

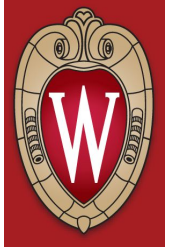
CLIENT	SERVER
rename(sha(old), tmp)	
Call RPC	rename(old, new)
	Respond
? x *rename(tmp, sha(new))*	

Prevent if server crash after rename and another client rename another file to new

1.4 Crash Recovery Protocol



- Client crash
  - Remove all dirty cache and temporary files
  - (keep clean cache)
  - Dirty files are renamed to client cache files only on successful write on server
- Server crash
  - Remove temporary files (generated when uploading file to server)
  - No special message exchanges
  - Client will response ENONET for all gRPC failures (return normal errno for normal errors)
- No additional metadata information needed to be persisted

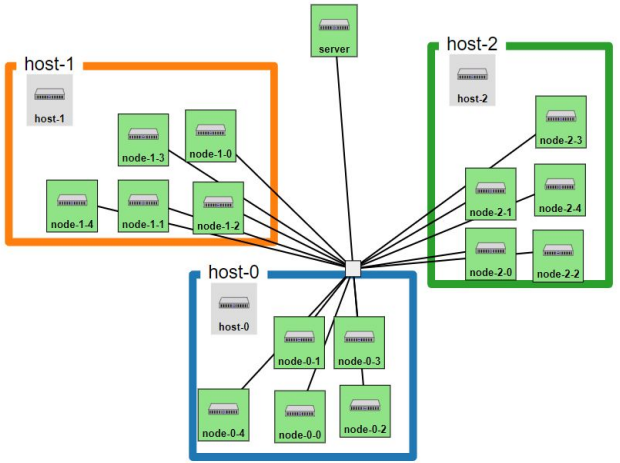


Part 2 : Measurement

Hardware used:

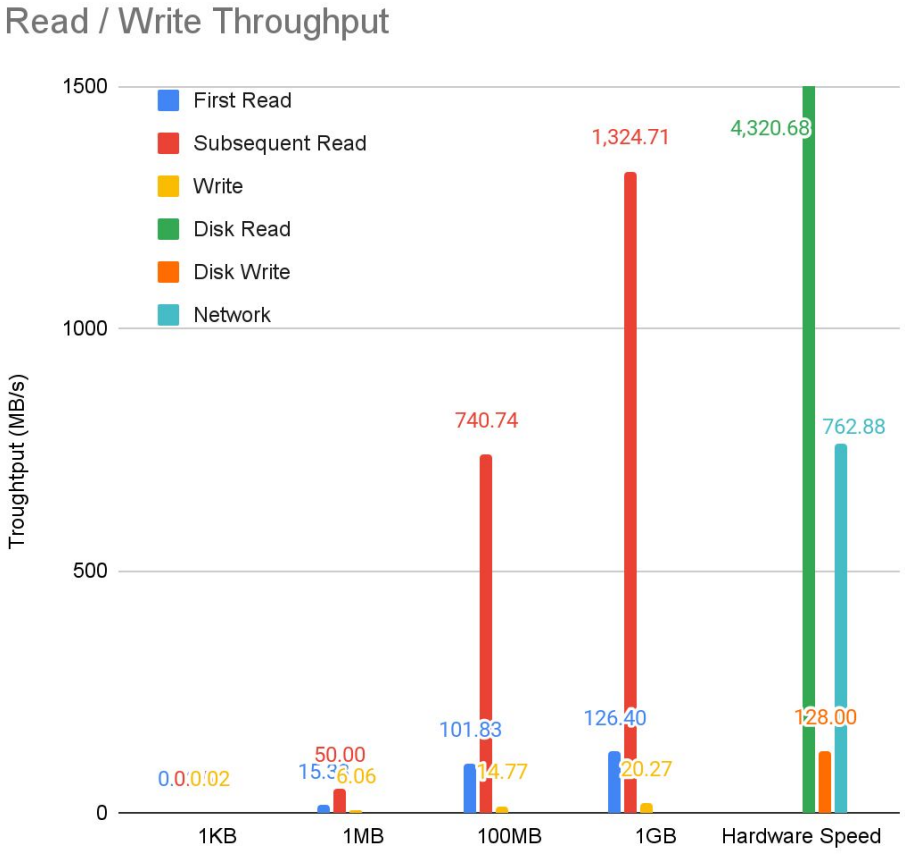
Cloudlab

- 4 Physical machine
- Server on its own physical machine
- At most 5 clients VM share a host
- Ext4-ordered for clients and servers

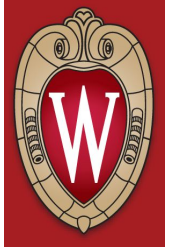


All cache are cleared unless specified

2.1 Performance : Read/Write Performance



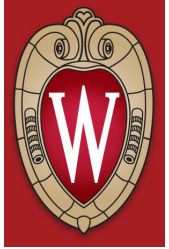
Workload



Copy	Copy an entire vscode (v1.65) source tree to fuse ~80MB, ~5000 files
Remove	rm -rf the vscode source tree
SQLite	Insert 2 rows in a large SQLite database (~3GB)
Make	Compile htop 3.1.2

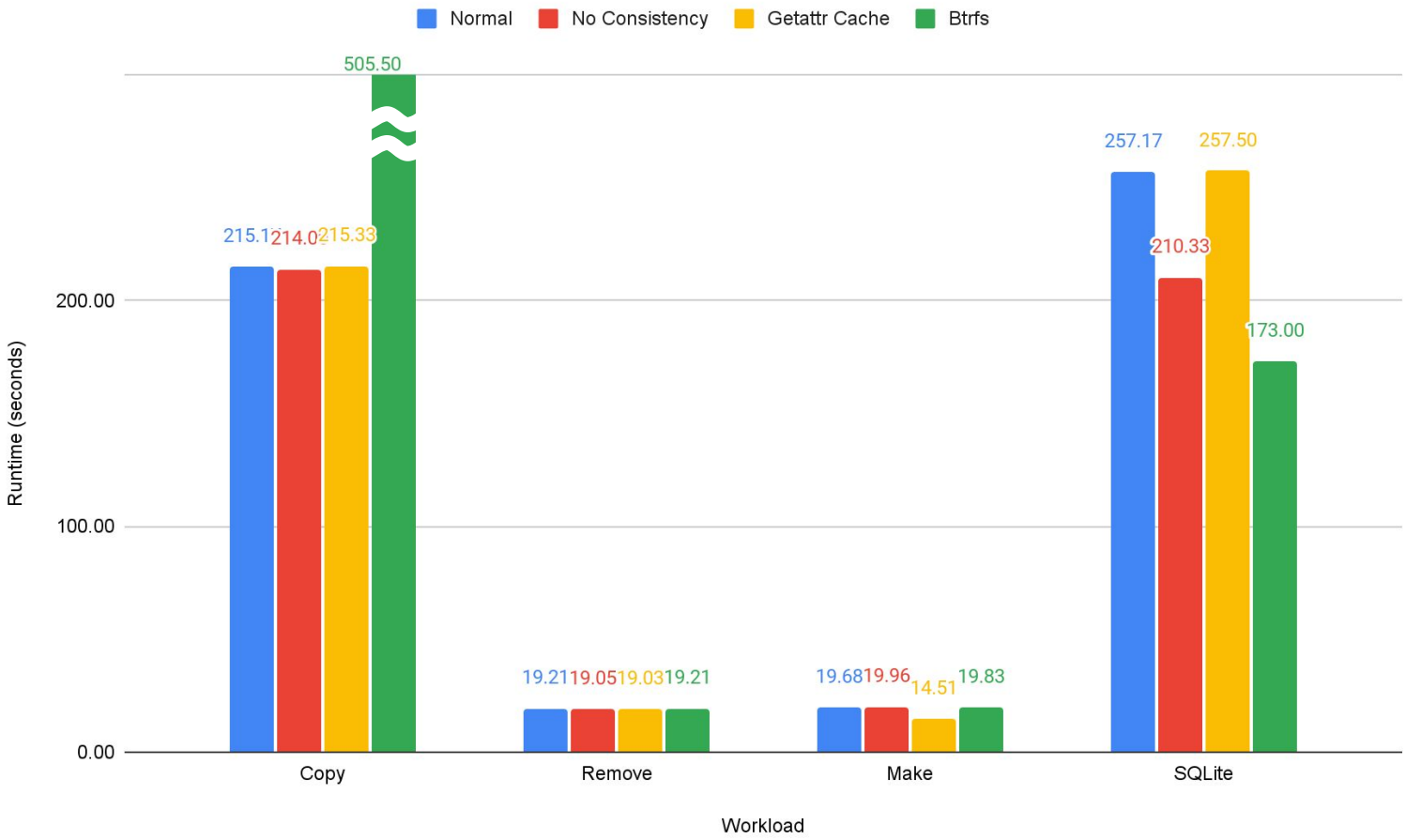
Performance of different workloads

Optimizations & Comparison

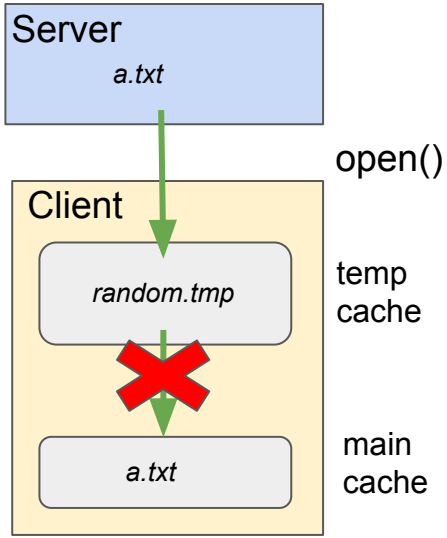


- Optimization 1: Use Btrfs to avoid copying whole file on writes.
- Optimization 2: Cache getattr results for 1 second. Do not send multiple getattr RPC calls for consecutive getattr.
- Baseline 1: Normal Implementation
- Baseline 2: Reduce consistency guarantee. Do not create a dirty file for writing.

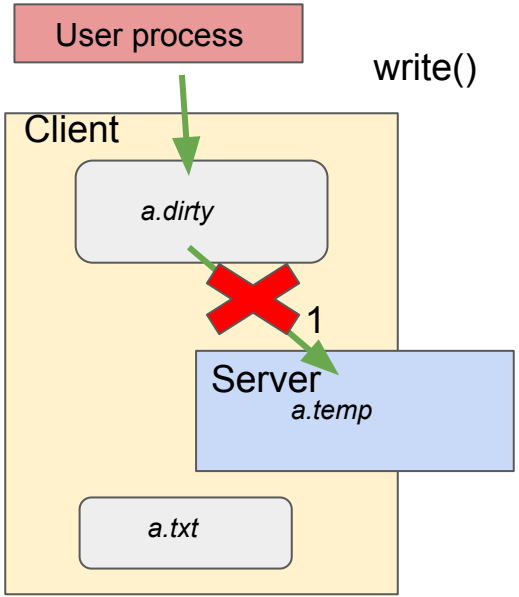
Optimizations & Comparison



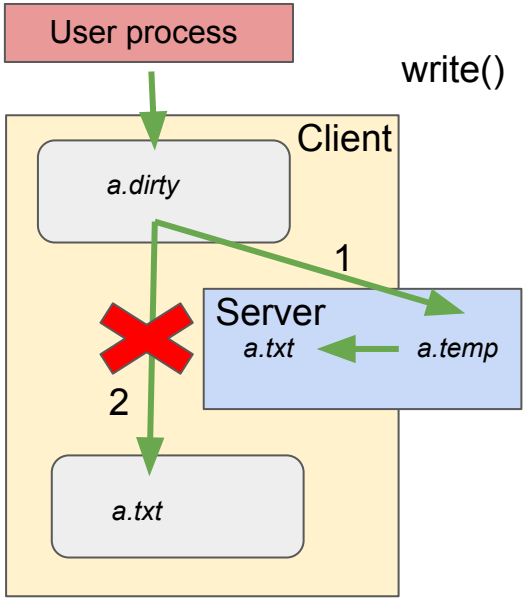
2.2 Reliability (Client crash)



Case 1: crash before rename file during open(), on recovery client will have to refetch from server

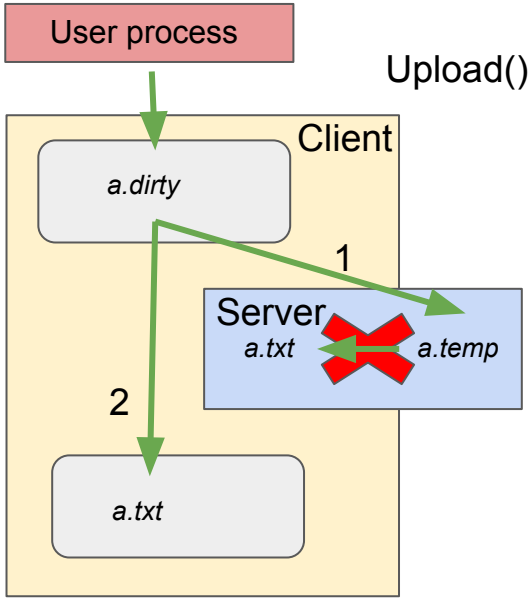


Case 2: crash before sending file to server, user will have to write again on client restart

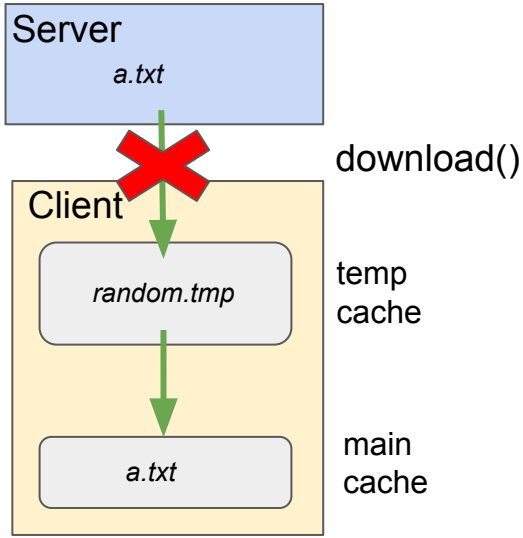


Case 3: crash before renaming the file to be written, client loses the file but server file got updated, client can read on open()

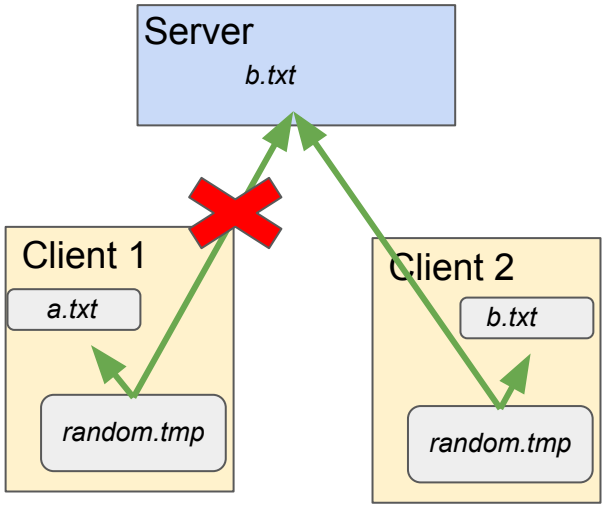
2.2 Reliability (Server crash)



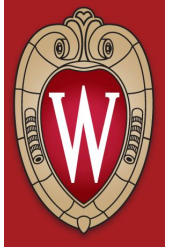
Case 1: Crash before renaming the file to be written on server side, on restarting, server loses the file and client has to resend the flush().



Case 2: Crash while getting the file from the server, on recovery client will have to refetch from server.



Case 3: Last Writer Wins.
During update from multiple clients, the client, which calls close() last will update the file last and that will be saved on server.



THANK YOU