A Low-bandwidth Network File System - SOSP'01

Athicha Muthitacharoen, Benjie Chen, and David Mazières

MIT Laboratory for Computer Science and NYU Department of Computer Science

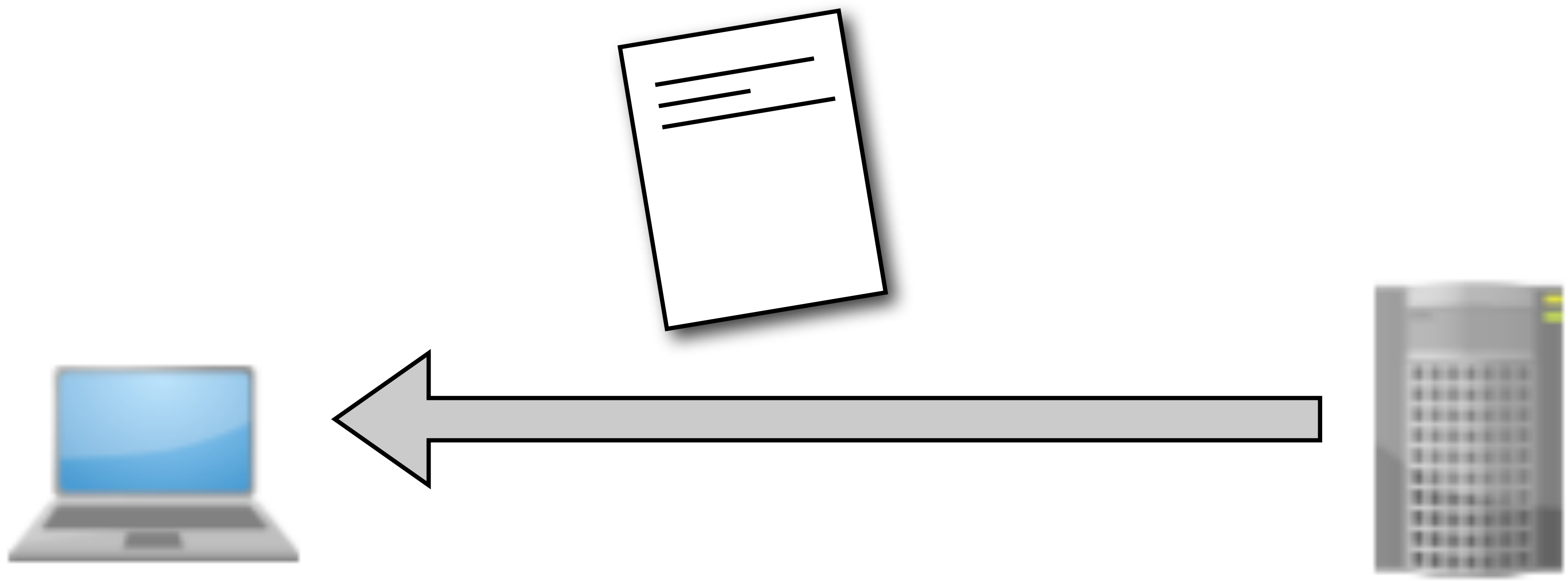
Presented by Mario Linares-Vásquez

A Low-bandwidth Network File System (LBFS)

Purpose: present a network file system that consumes less bandwidth than most current file systems

Intuition: avoid sending data over the network that is already in the file system or the client's cache

Remote access to a file system (Option 1)

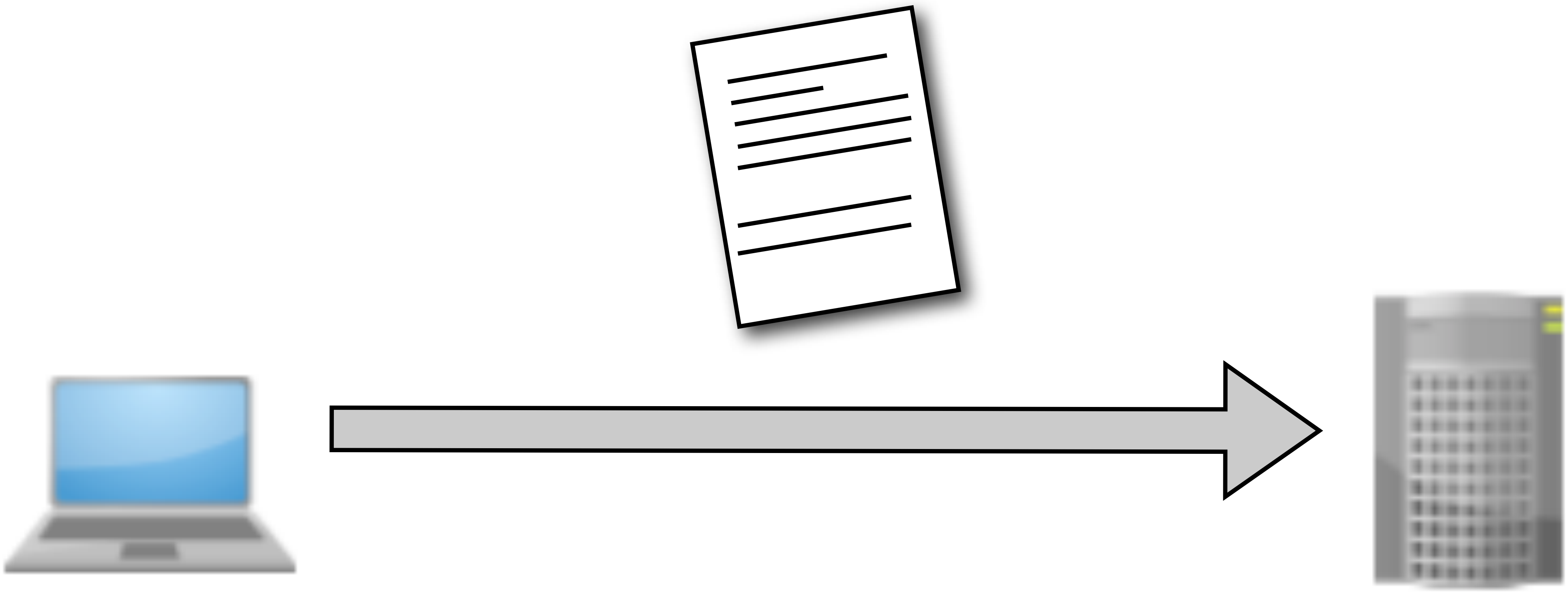


1. Copy

Remote access to a file system (Option 1)

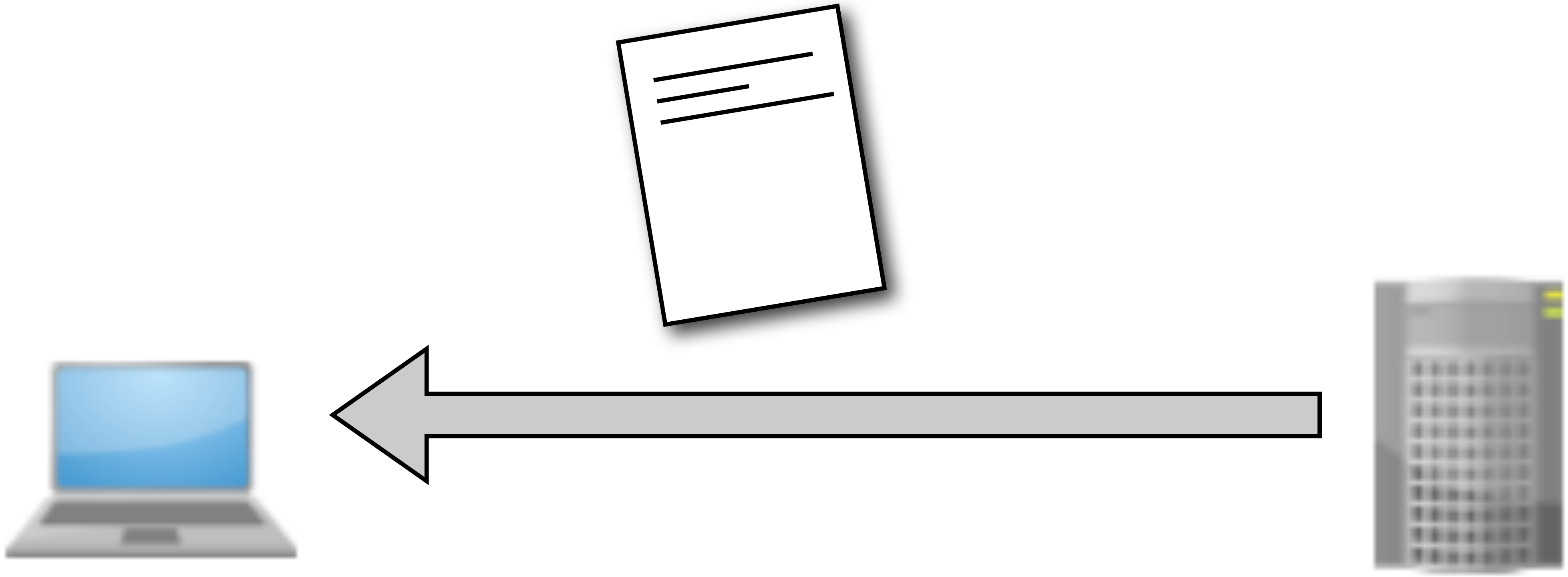
1. Copy
2. Modify remotely

Remote access to a file system (Option 1)



1. Copy
2. Modify remotely
3. Copy to server

Remote access to a file system (Option 2)



1. Copy

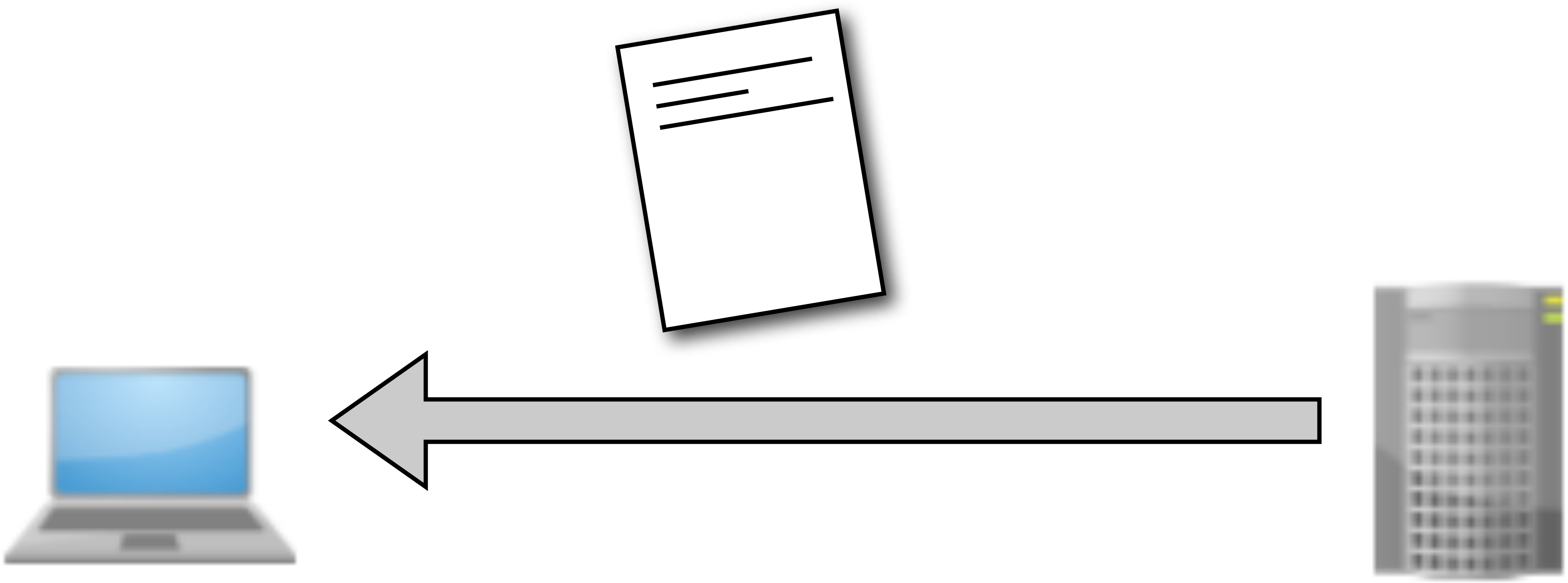
Remote access to a file system (Option 2)

1. Copy
2. Modify remotely

Remote access to a file system (Option 2)

1. Copy
2. Modify remotely
3. Save incrementally

Remote access to a file system (Option 3)

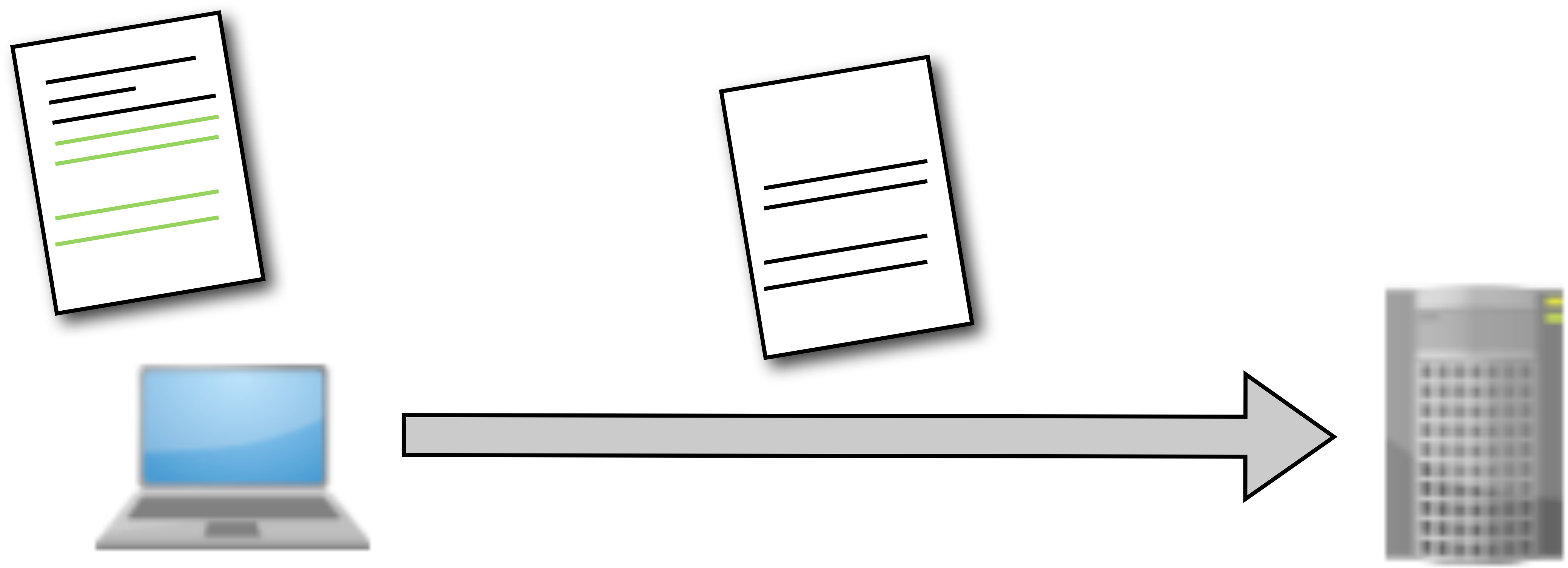


1. Copy

Remote access to a file system (Option 3)

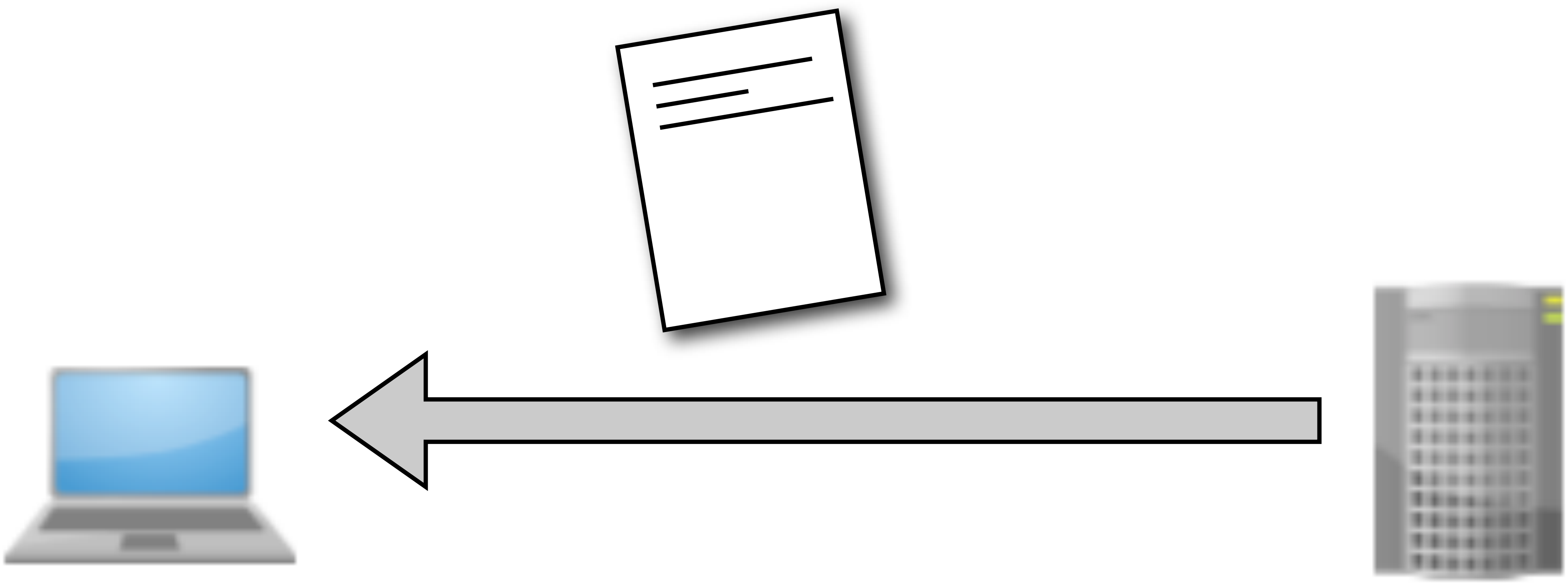
1. Copy
2. Modify remotely

Remote access to a file system (Option 3)



1. Copy
2. Modify remotely
3. Send changes

Low-bandwidth Network File System (LBFS)

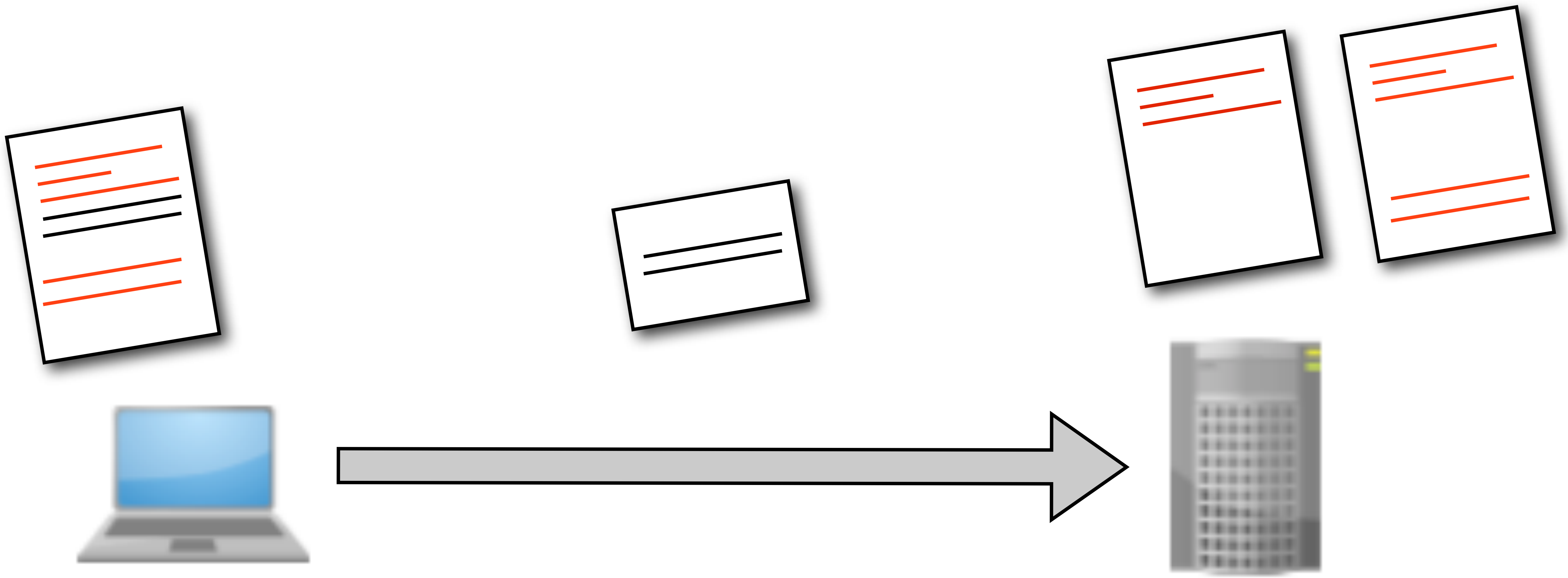


1. Copy

Low-bandwidth Network File System (LBFS)

1. Copy
2. Modify remotely

Low-bandwidth Network File System (LBFS)

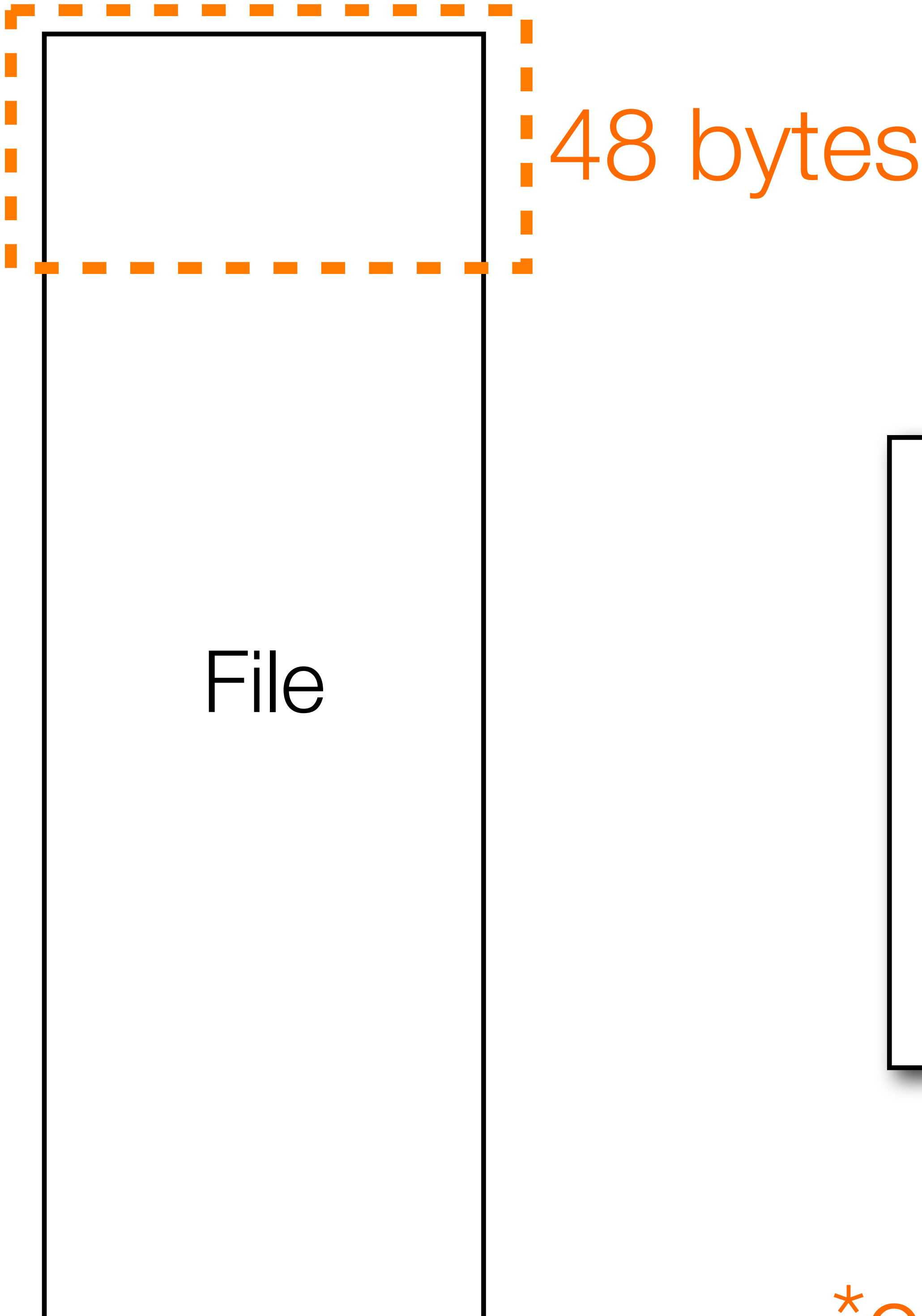


1. Copy
2. Modify remotely
3. Copy chunks that are not on the server

LBFS Design Principles

- Save bandwidth while providing traditional file systems semantics
- Close-to-open consistency
- Exploit similarities between files
- Hash indexing of data chunks
- Unobtrusive installation on an already running file system

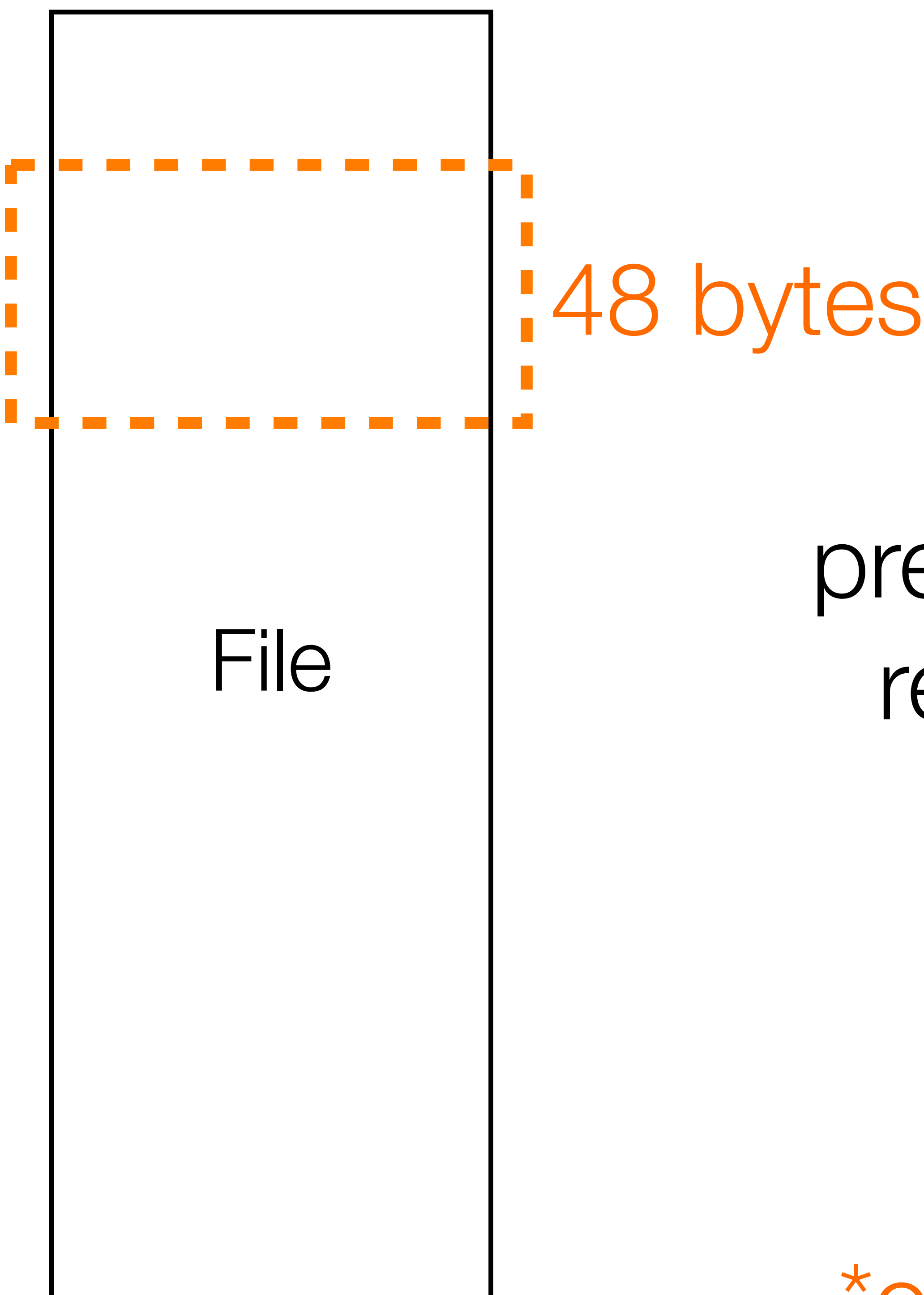
Dividing files into chunks...



A file is split in data chunks by using Rabin fingerprints

*overlapping sliding window

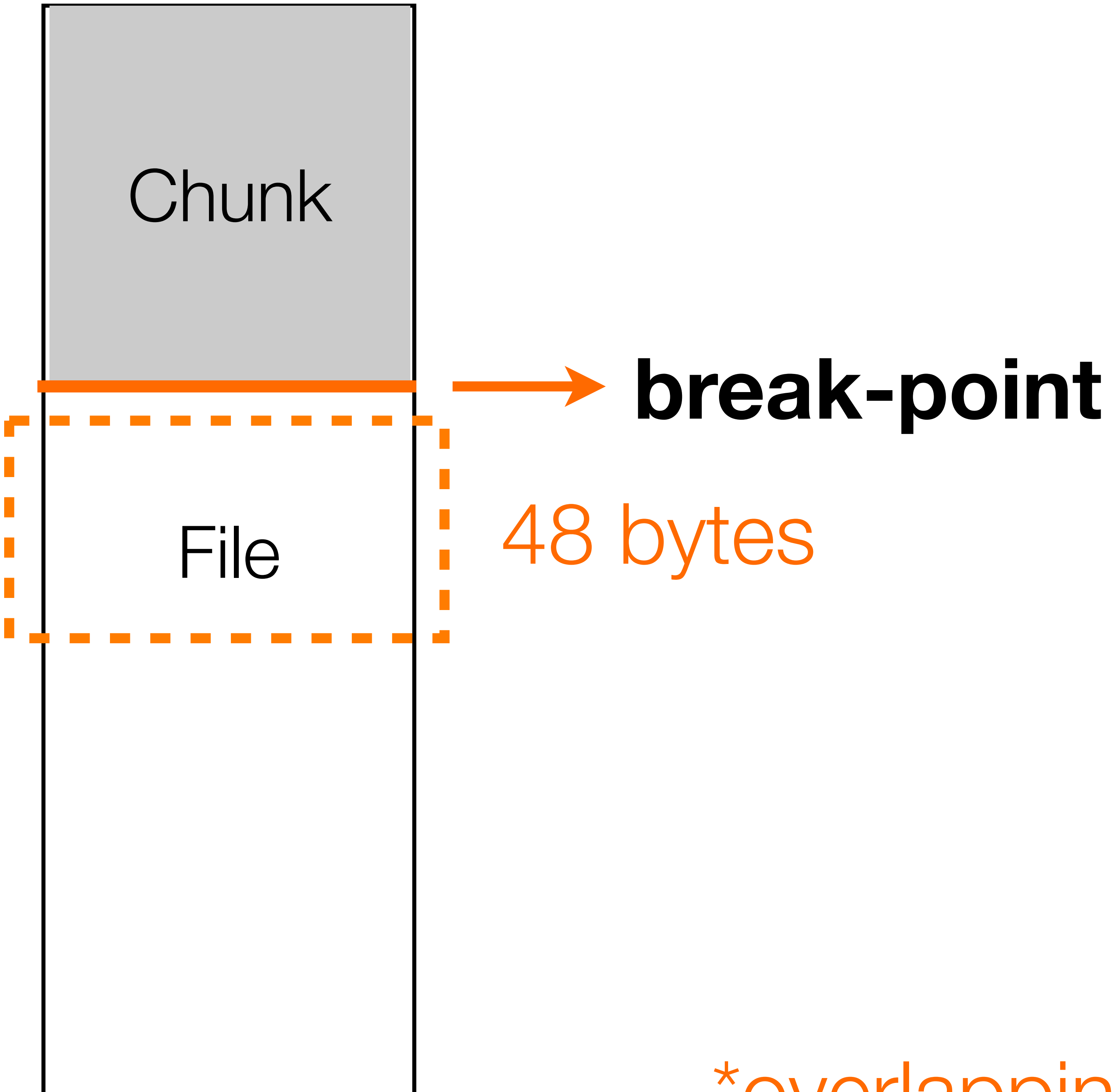
Dividing files into chunks...



If the fingerprint is equal to a predefined value, then a boundary region is defined (**break-point**)

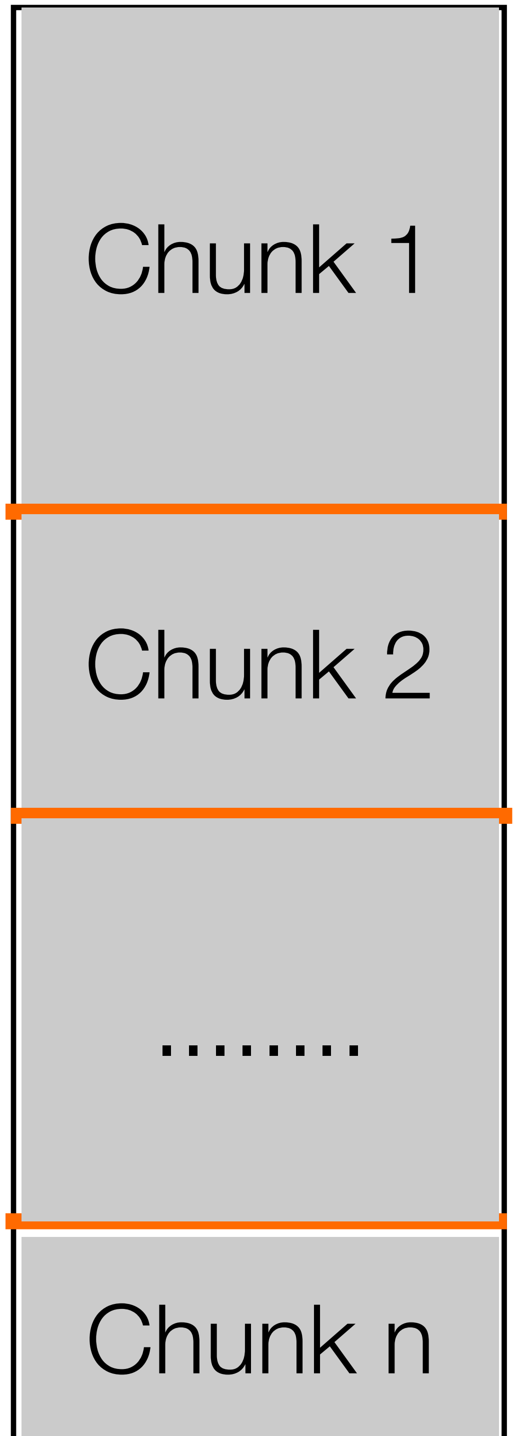
*overlapping sliding window

Dividing files into chunks...



*overlapping sliding window

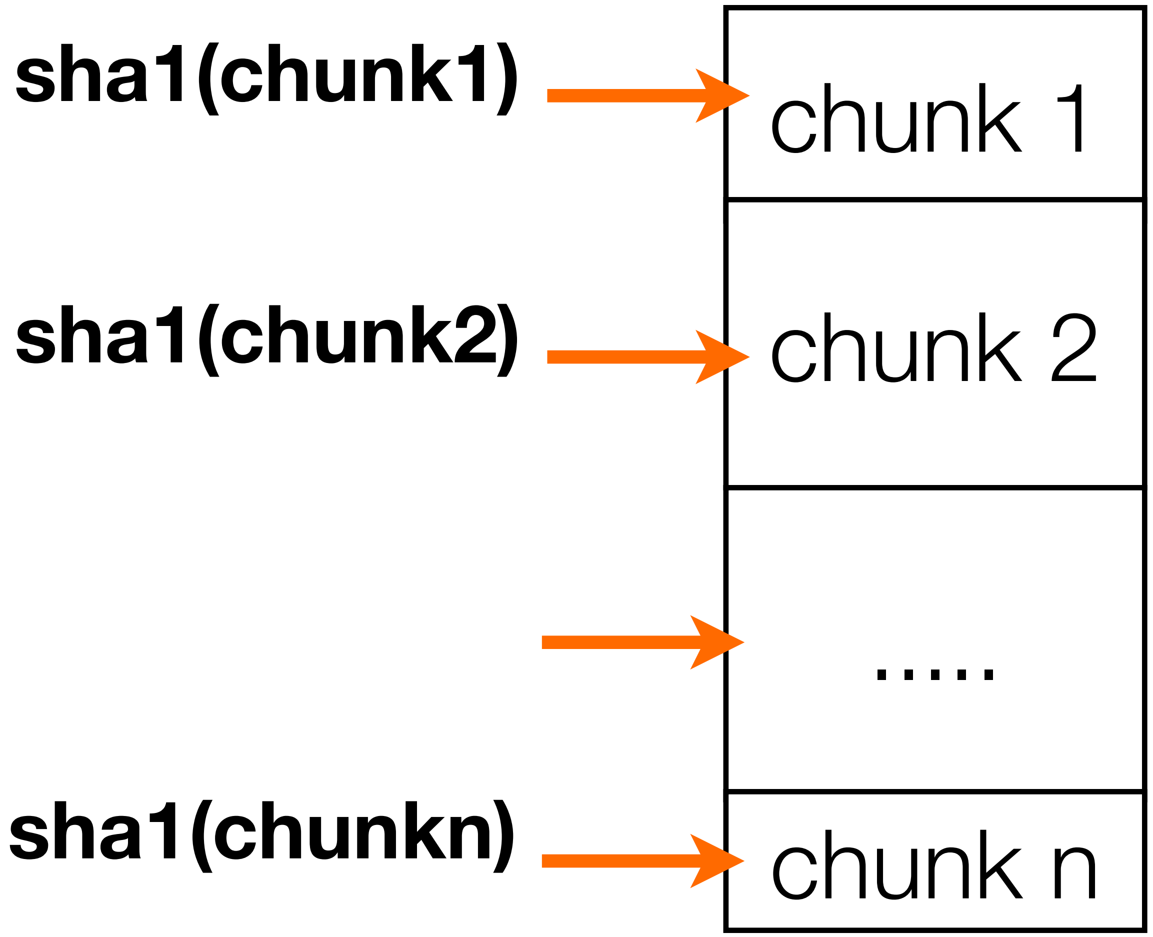
Dividing files into chunks...



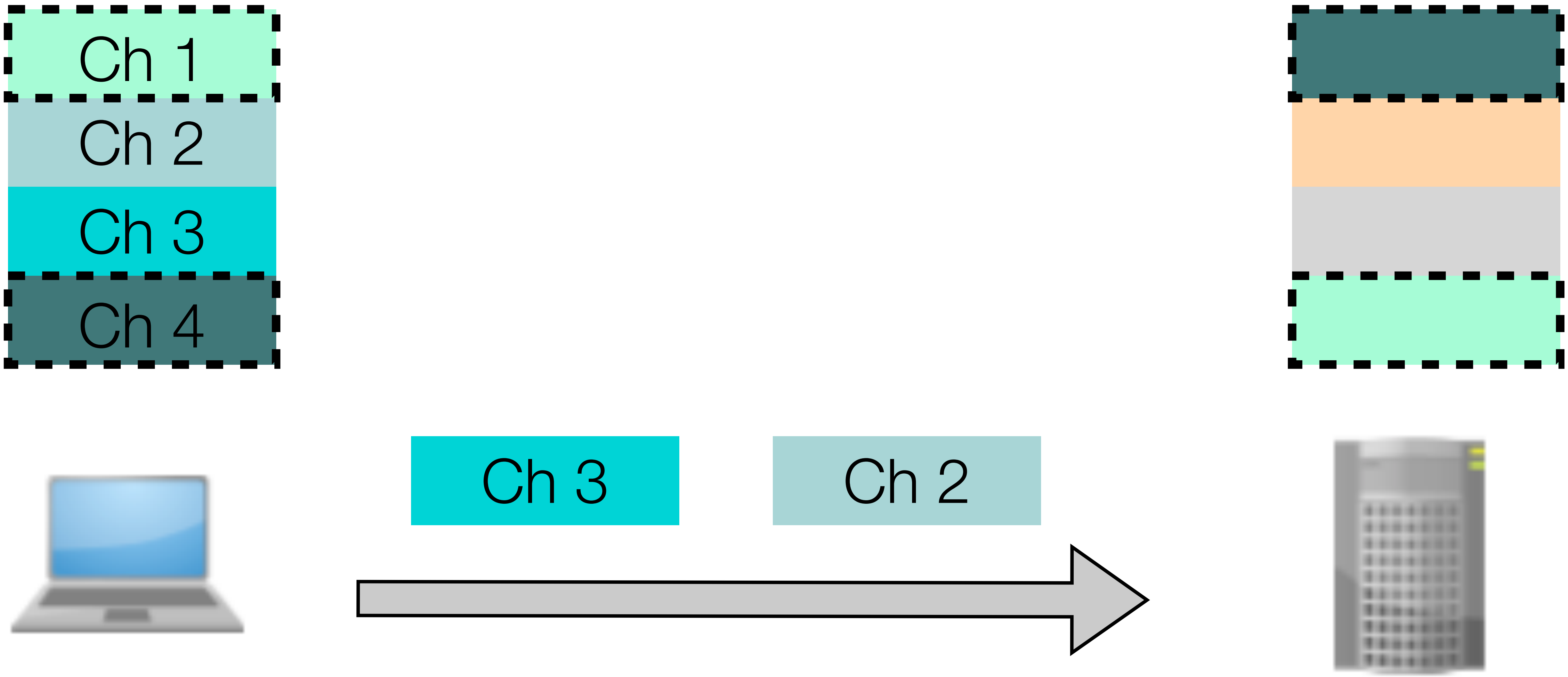
- Minimum chunk is 2K
- Maximum chunk is 64K

Indexing

- On both the client and server, LBFS indexes files
- Each chunk is indexed using SHA-1 (first 64 bits)
- Indexing is used to save chunk transfers



LBFS



Pathological cases (chunks size)

- The lower the chunk size, the greater the index (e.g., every 2K happened to be a breakpoint)
- The greater the chunk size, the lower the number of chunks (e.g., files containing enormous chunks)

Close-to-open consistency

- Client fetches a new version when the file is not in the local cache or the cached version is not up to date

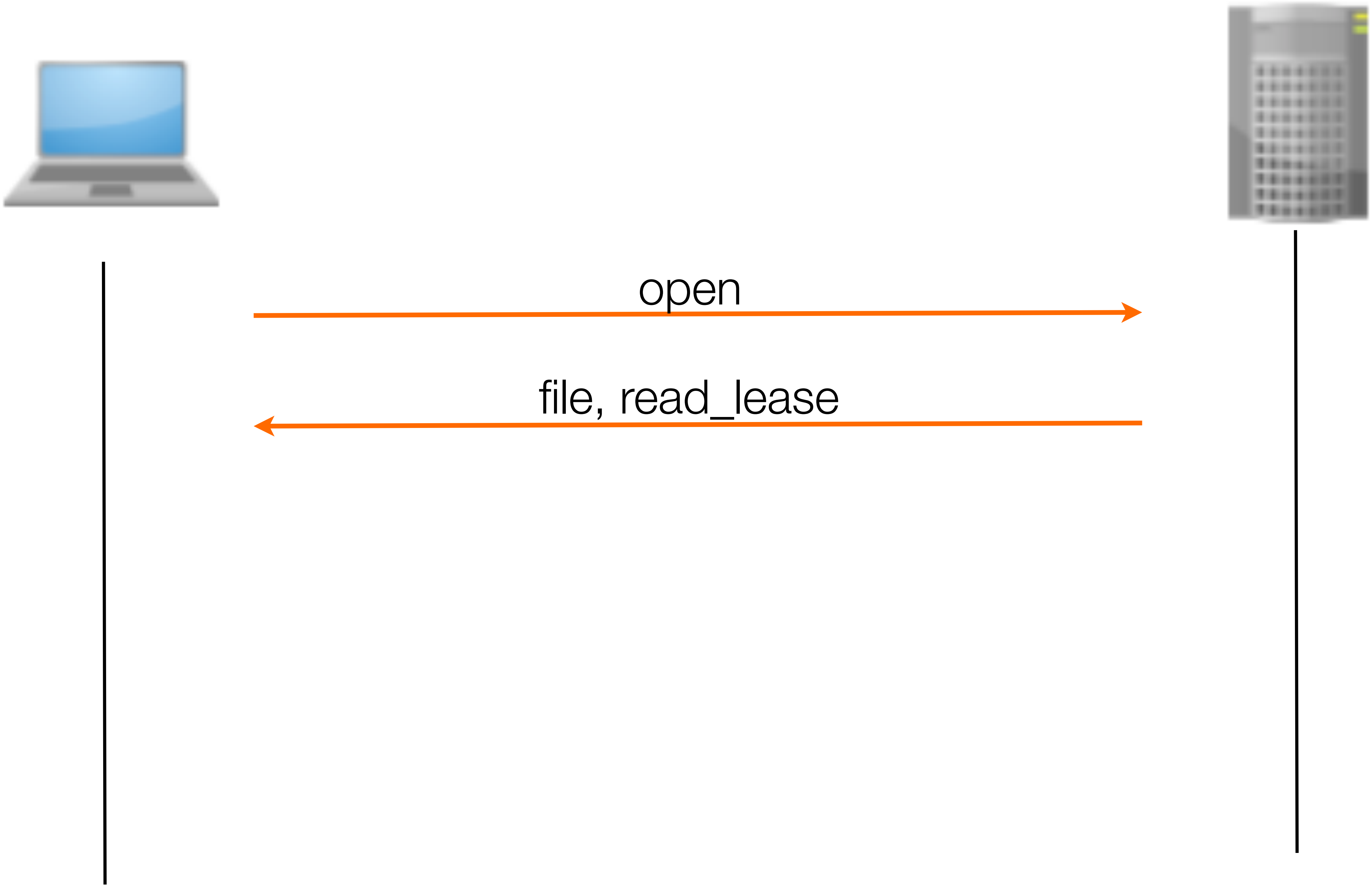
- When a process close a file, the client writes the data back to the server

Close-to-open consistency

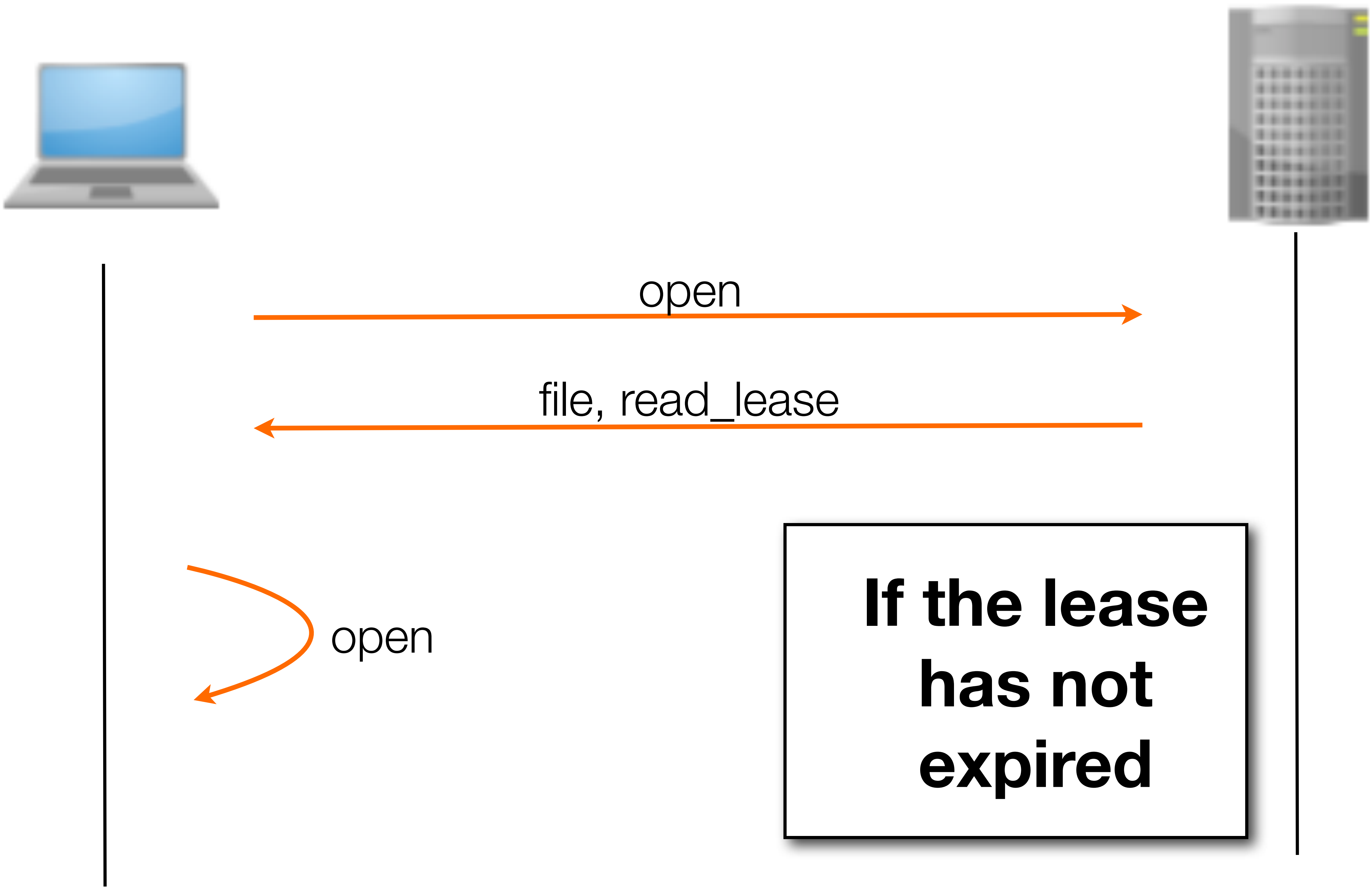
- Read leases are used to identify if a file is up to date
- Write leases are not used
- Similar semantic to AFS (the last process closing the file overwrites changes from others)

Close-to-open consistency (read lease)

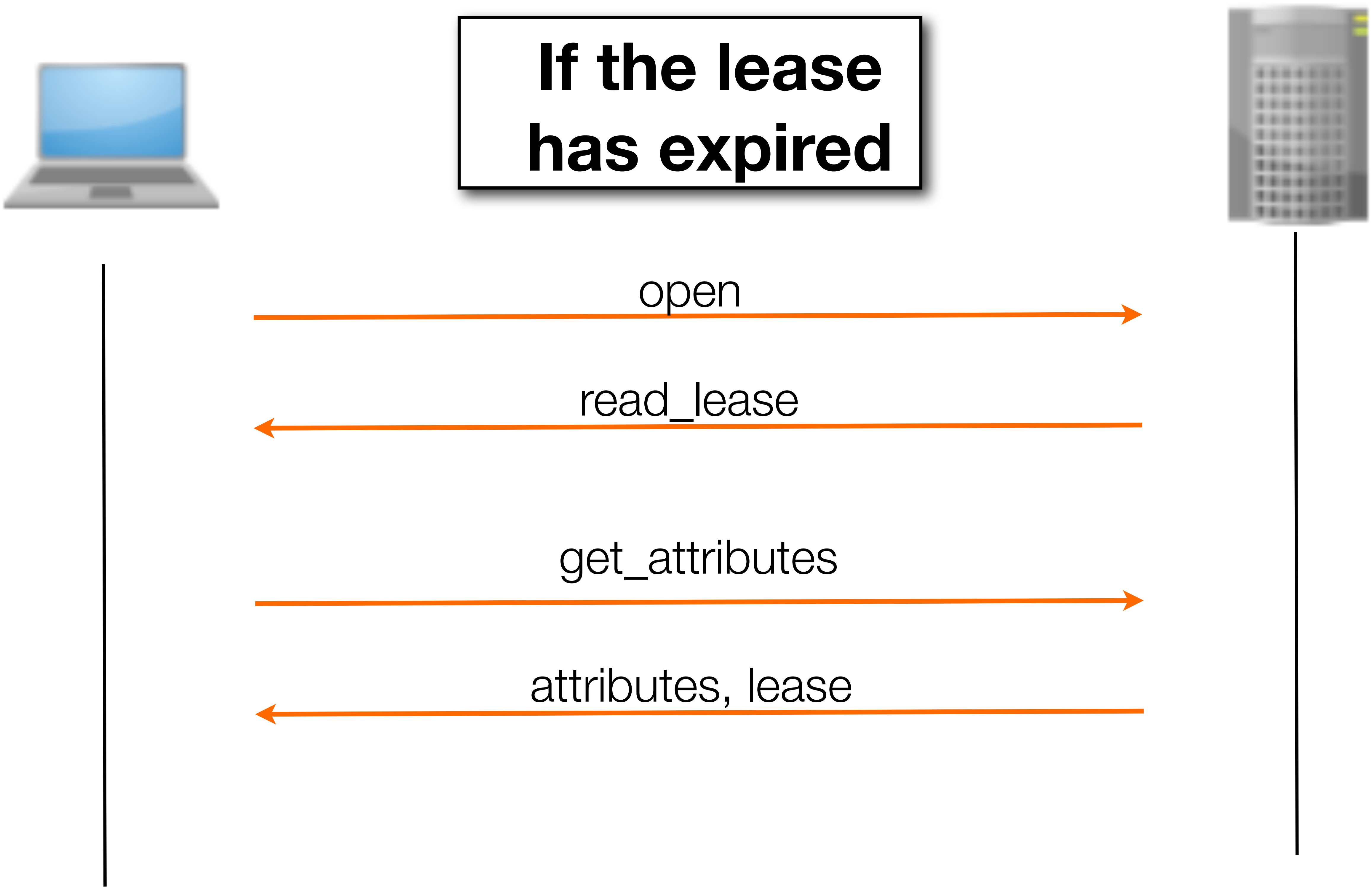
Close-to-open consistency (read lease)



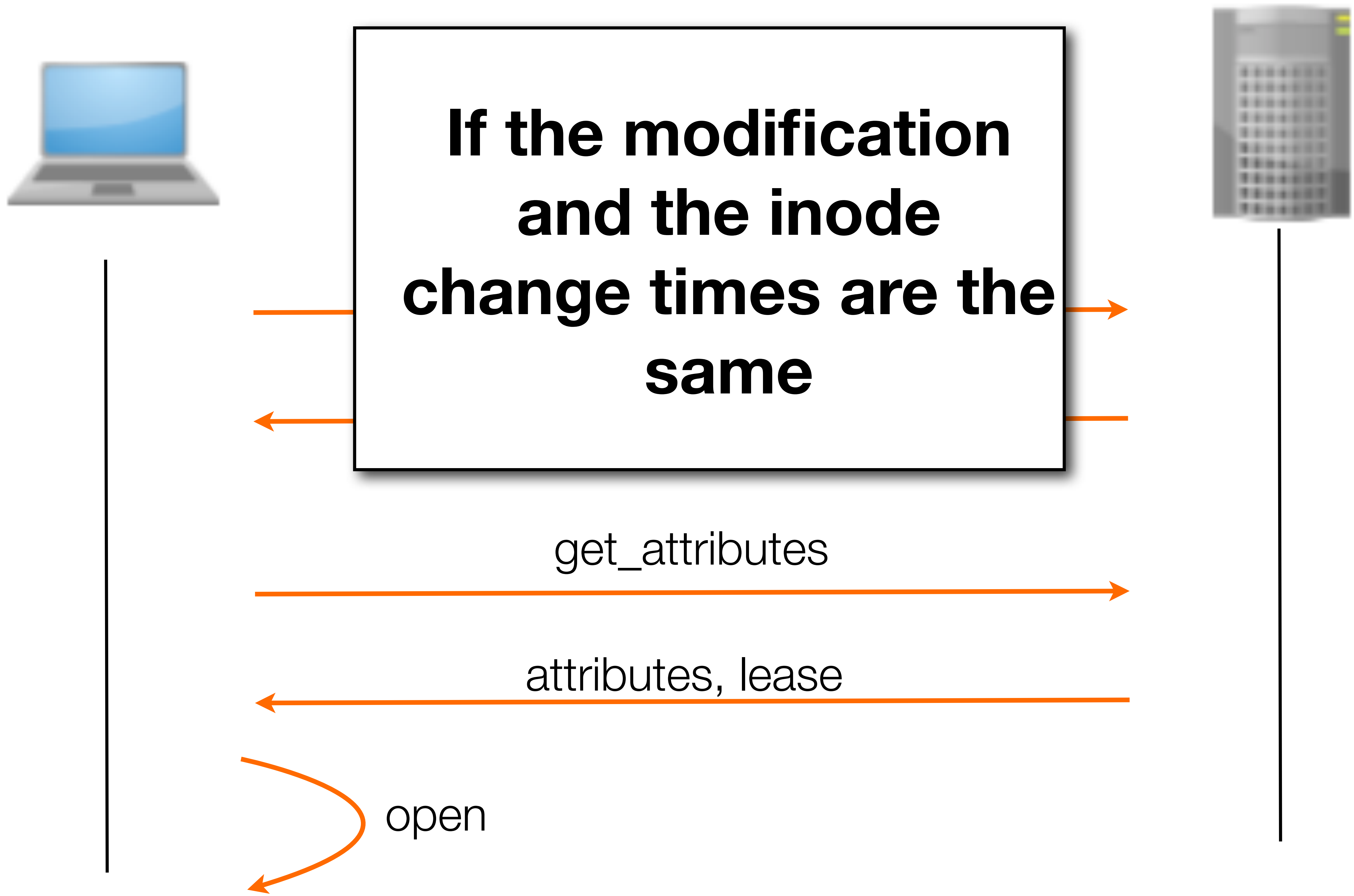
Close-to-open consistency (read lease)



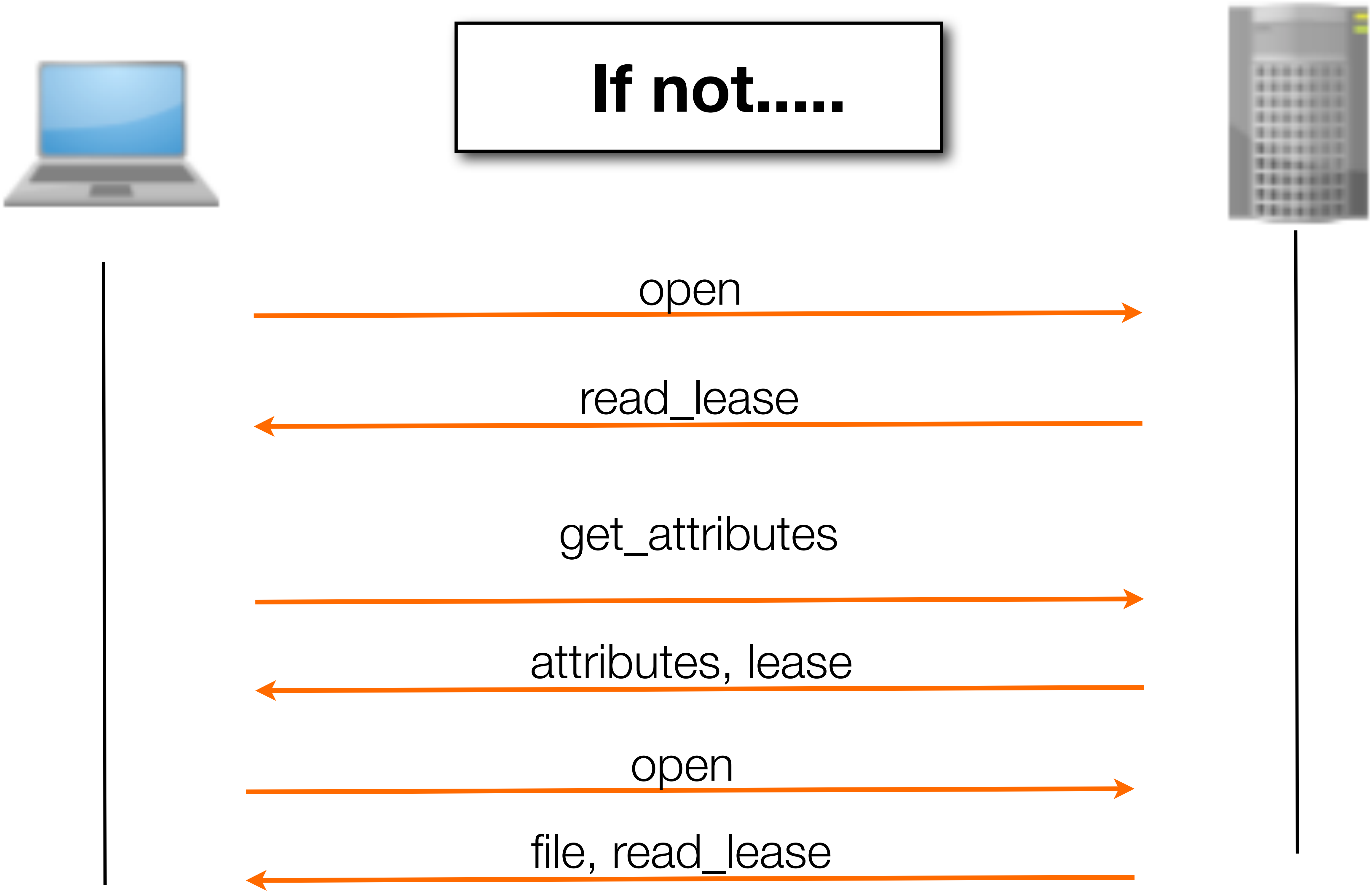
Close-to-open consistency (read lease)



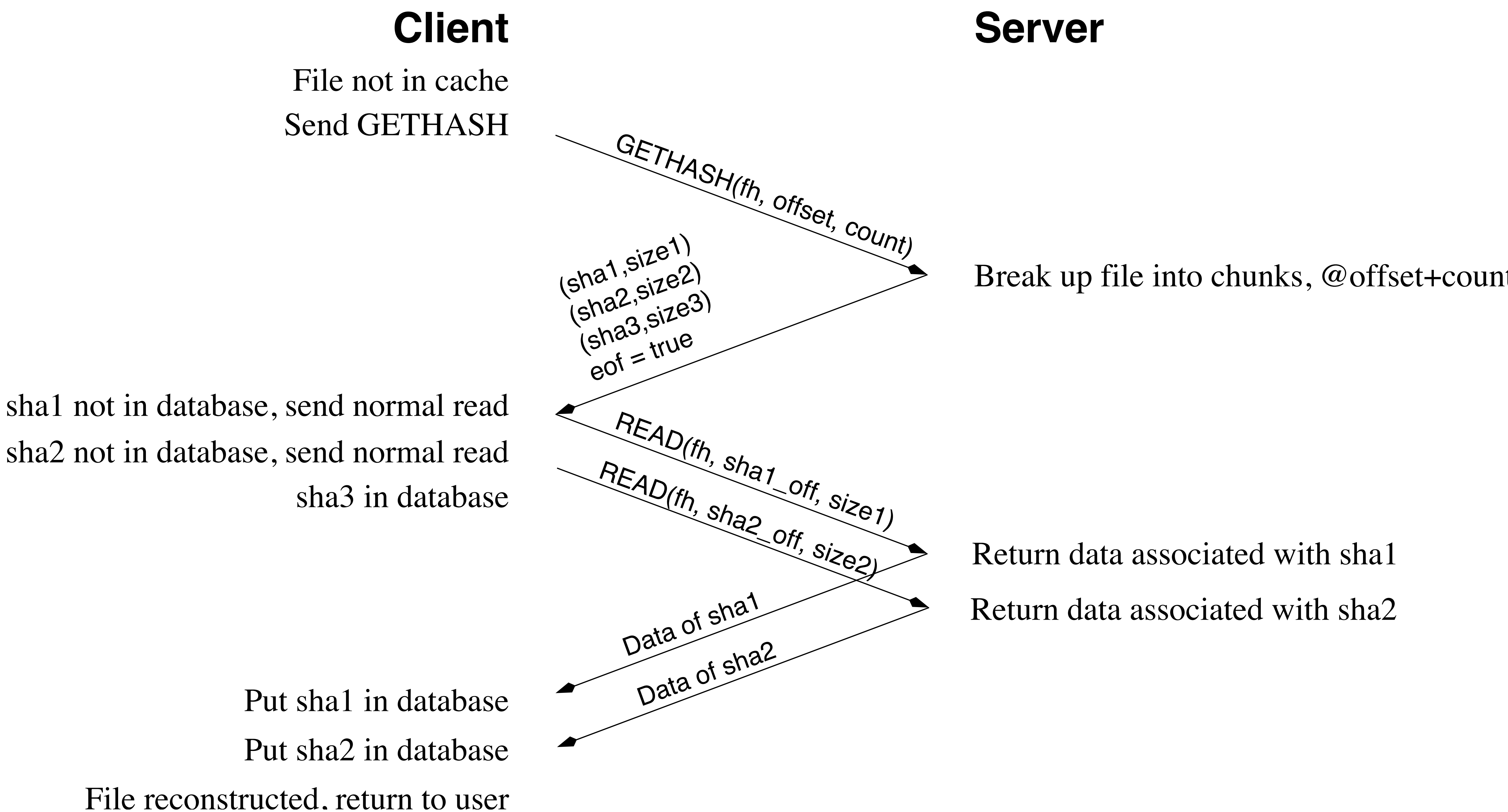
Close-to-open consistency (read lease)



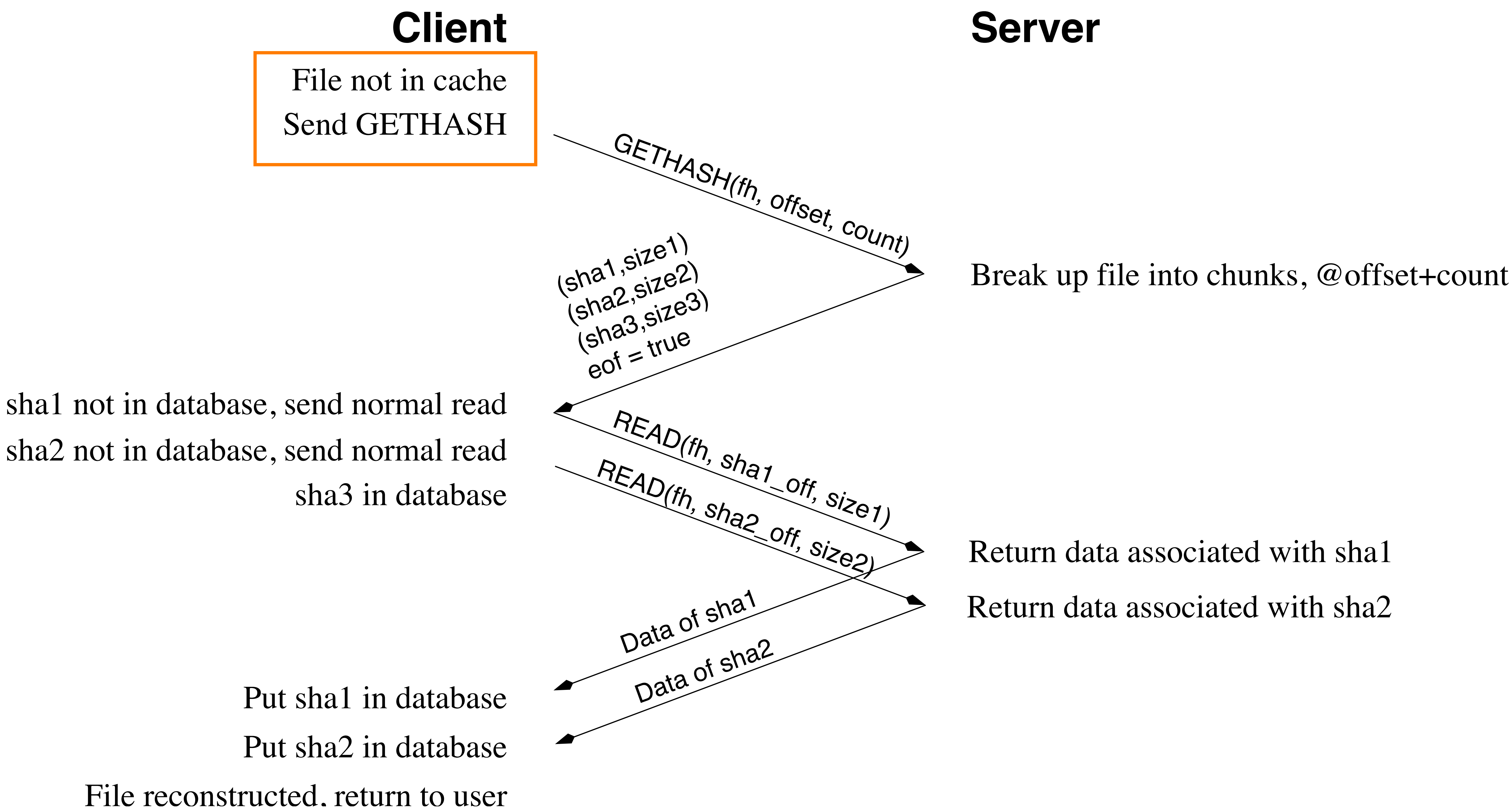
Close-to-open consistency (read lease)



File reads

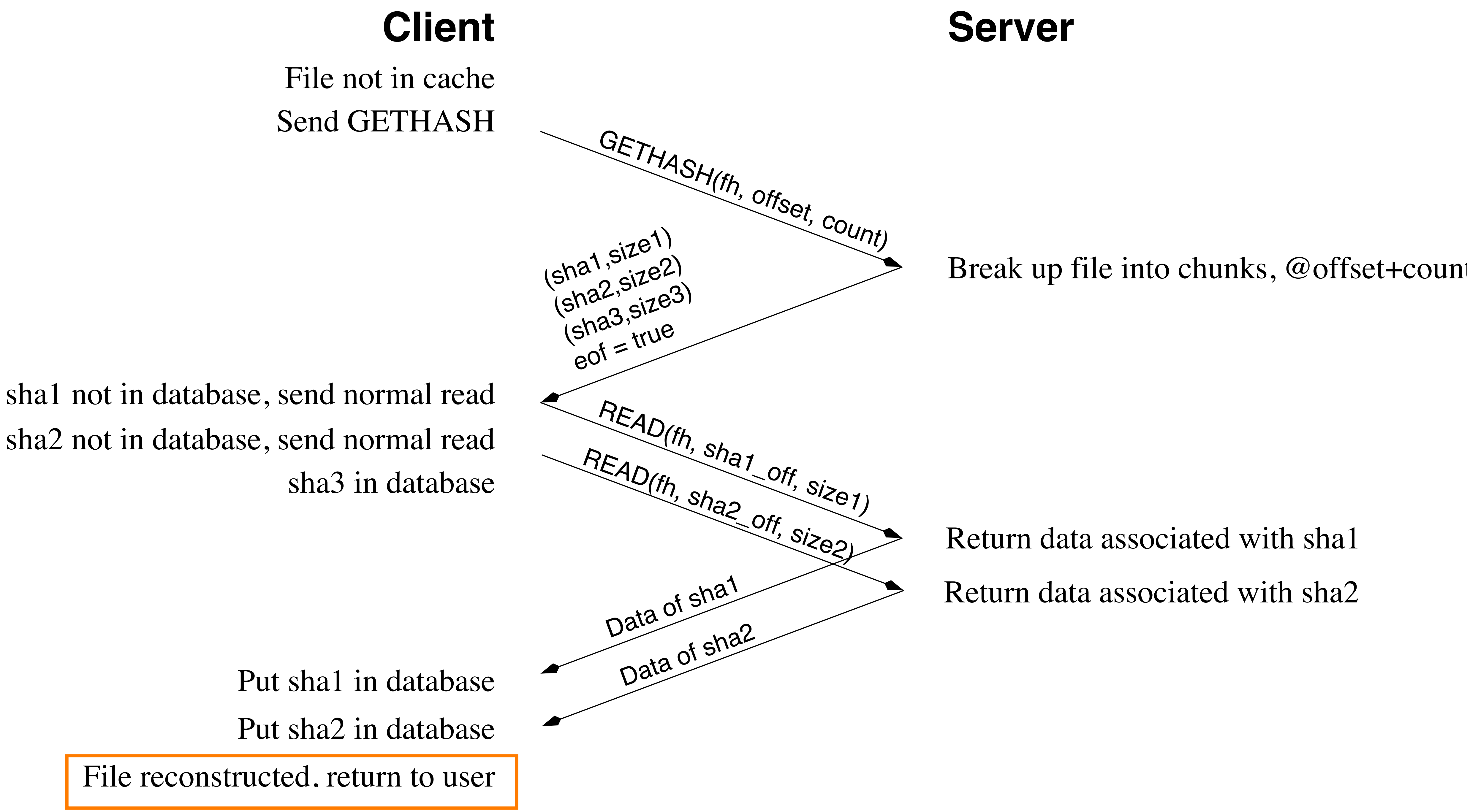

Client
Server
File not in cache
Send GETHASH
GETHASH(fh, offset, count)
Break up file into chunks, @offset+count
(sha1,size1)
(sha2,size2)
(sha3,size3)
eof = true
sha1 not in database, send normal read
sha2 not in database, send normal read
sha3 in database
READ(fh, sha1_off, size1)
READ(fh, sha2_off, size2)
Return data associated with sha1
Return data associated with sha2
Data of sha1
Data of sha2
Put sha1 in database
Put sha2 in database
File reconstructed. return to user

File reads

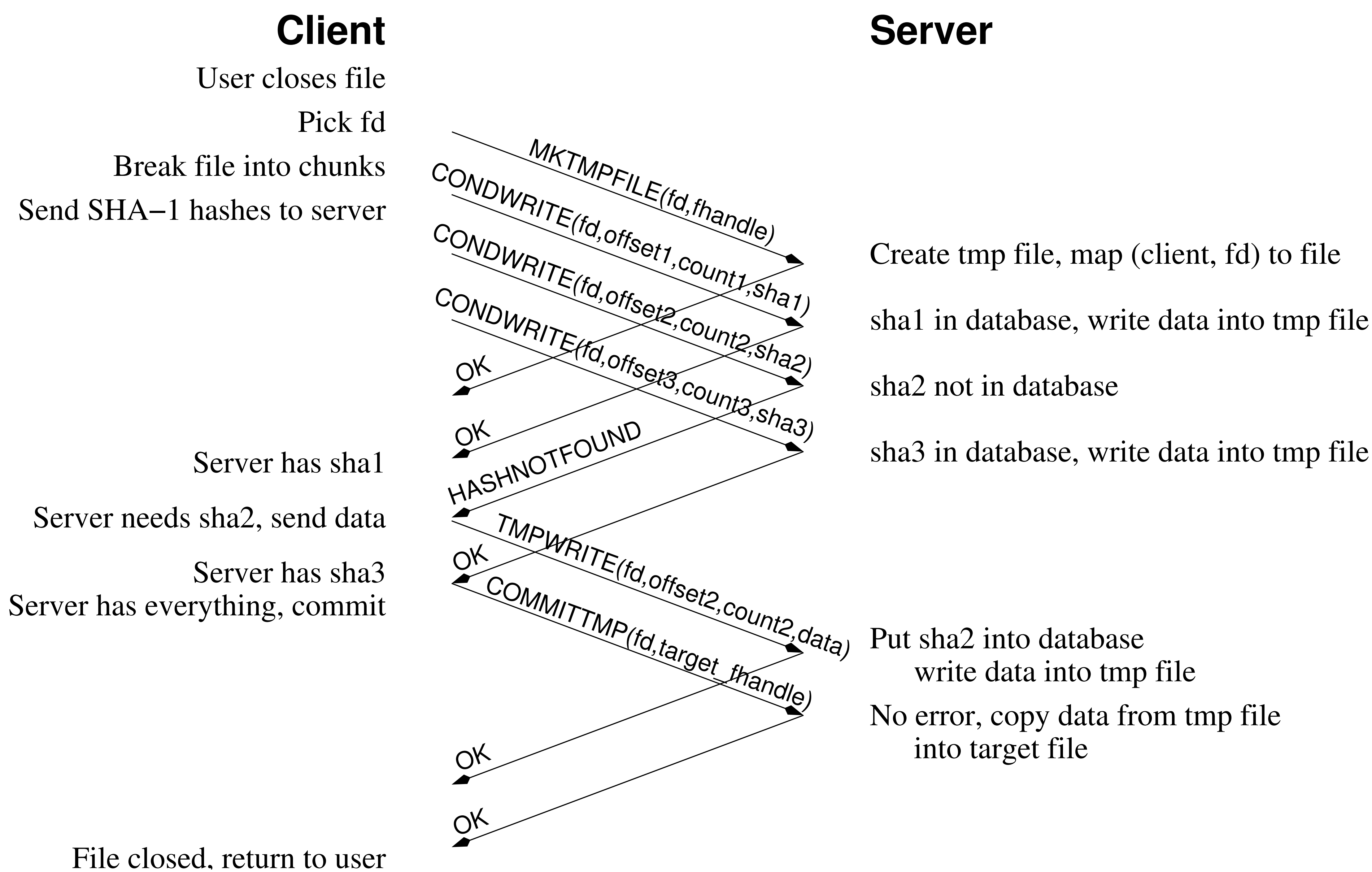


Client
Server
File not in cache
Send GETHASH
GETHASH(fh, offset, count)
Break up file into chunks, @offset+count
(sha1,size1)
(sha2,size2)
(sha3,size3)
eof = true
sha1 not in database, send normal read
sha2 not in database, send normal read
sha3 in database
READ(fh, sha1_off, size1)
READ(fh, sha2_off, size2)
Return data associated with sha1
Return data associated with sha2
Data of sha1
Data of sha2
Put sha1 in database
Put sha2 in database
File reconstructed. return to user

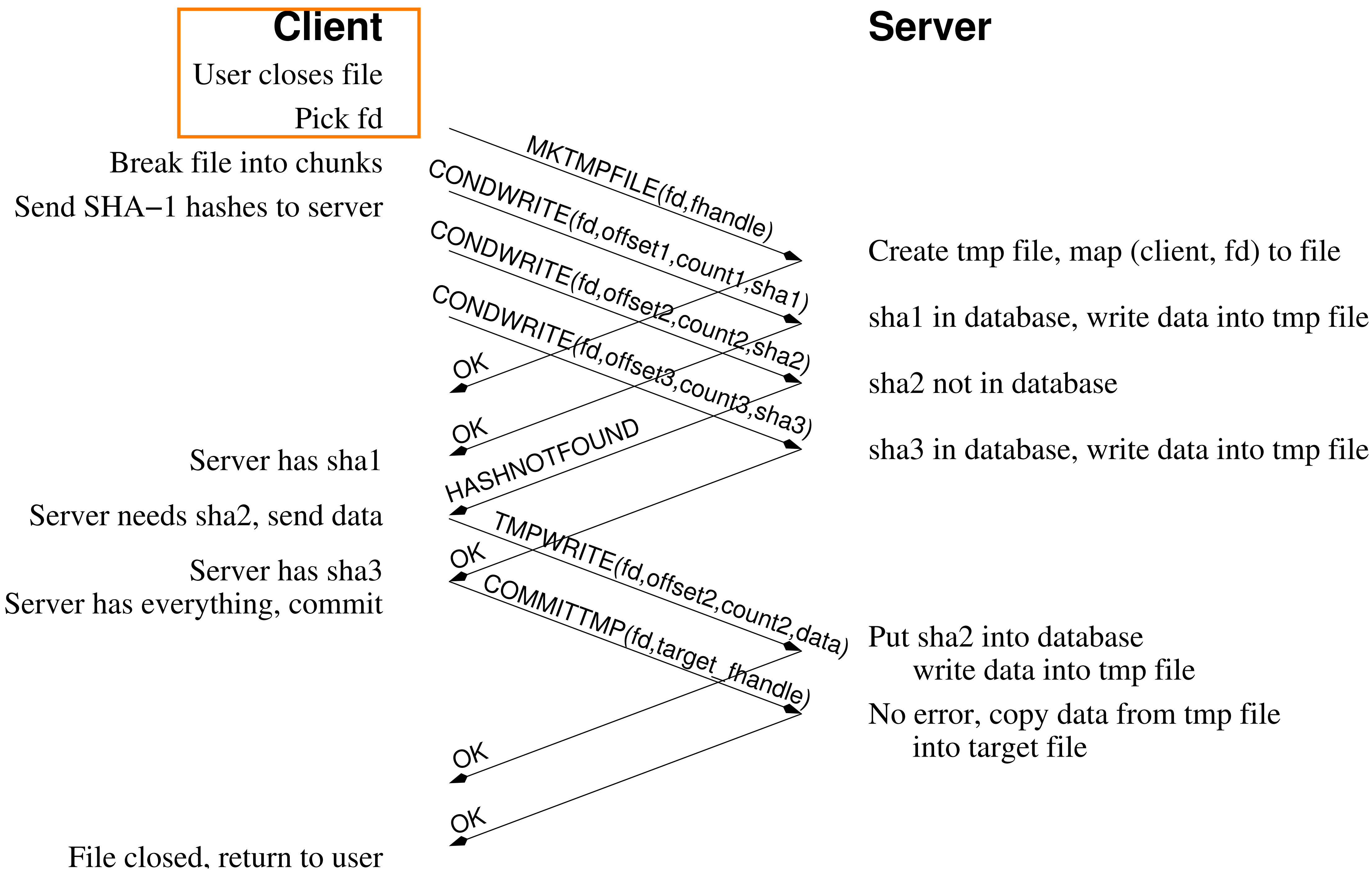
File reads


Client
Server
File not in cache
Send GETHASH
GETHASH(fh, offset, count)
Break up file into chunks, @offset+count
(sha1,size1)
(sha2,size2)
(sha3,size3)
eof = true
sha1 not in database, send normal read
sha2 not in database, send normal read
sha3 in database
READ(fh, sha1_off, size1)
READ(fh, sha2_off, size2)
Return data associated with sha1
Return data associated with sha2
Data of sha1
Data of sha2
Put sha1 in database
Put sha2 in database
File reconstructed. return to user

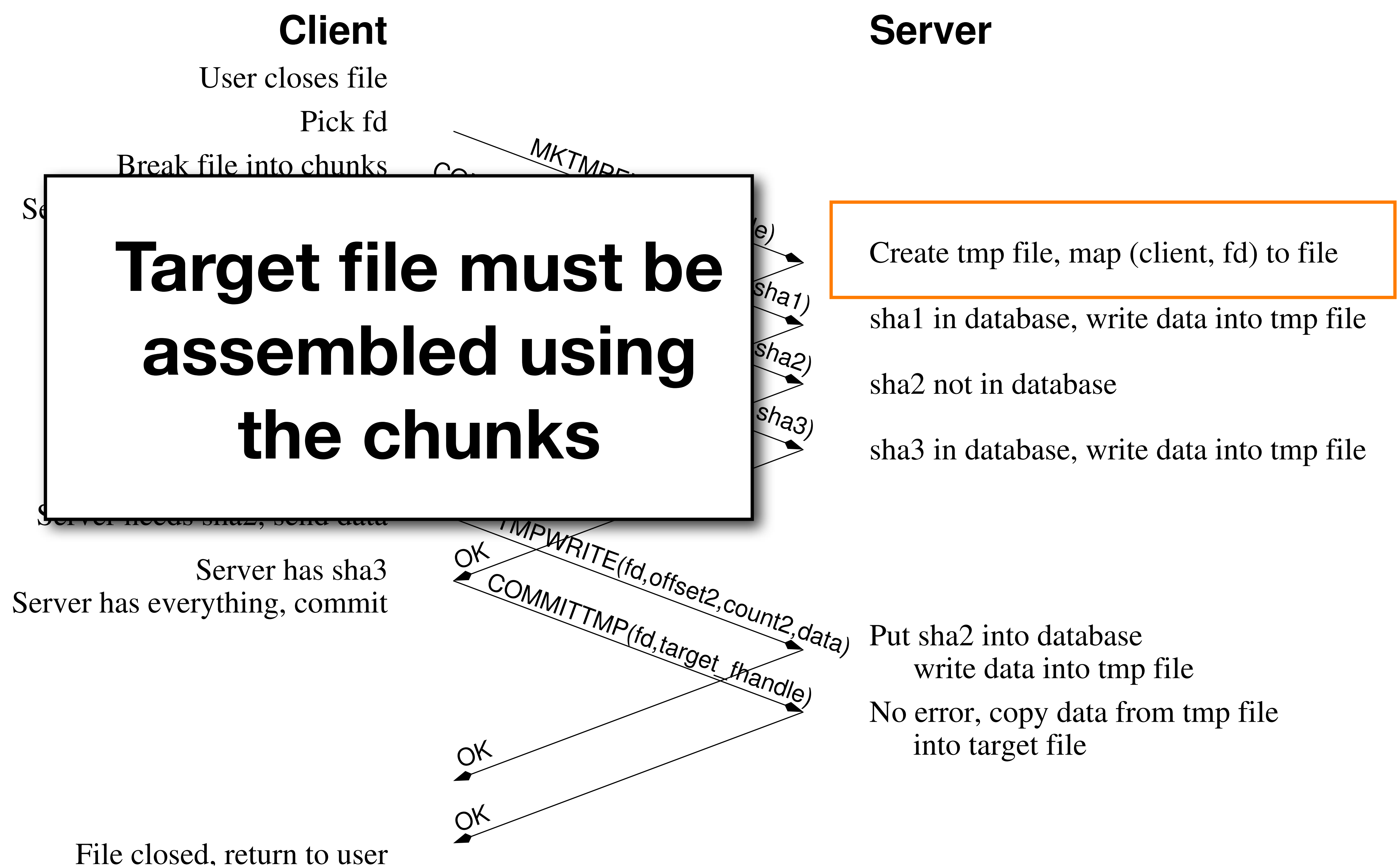
File writes


Client
Server
User closes file
Pick fd
Break file into chunks
Send SHA−1 hashes to server
MKTMPFILE(fd,fhandle)
CONDWRITE(fd,offset1,count1,sha1)
CONDWRITE(fd,offset2,count2,sha2)
CONDWRITE(fd,offset3,count3,sha3)
Create tmp file, map (client, fd) to file
sha1 in database, write data into tmp file
sha2 not in database
sha3 in database, write data into tmp file
OK
OK
HASHNOTFOUND
OK
Server has sha1
Server needs sha2, send data
Server has sha3
Server has everything, commit
TMPWRITE(fd,offset2,count2,data)
COMMITTMP(fd,target_fhandle)
Put sha2 into database
write data into tmp file
No error, copy data from tmp file
into target file
OK
OK
File closed, return to user

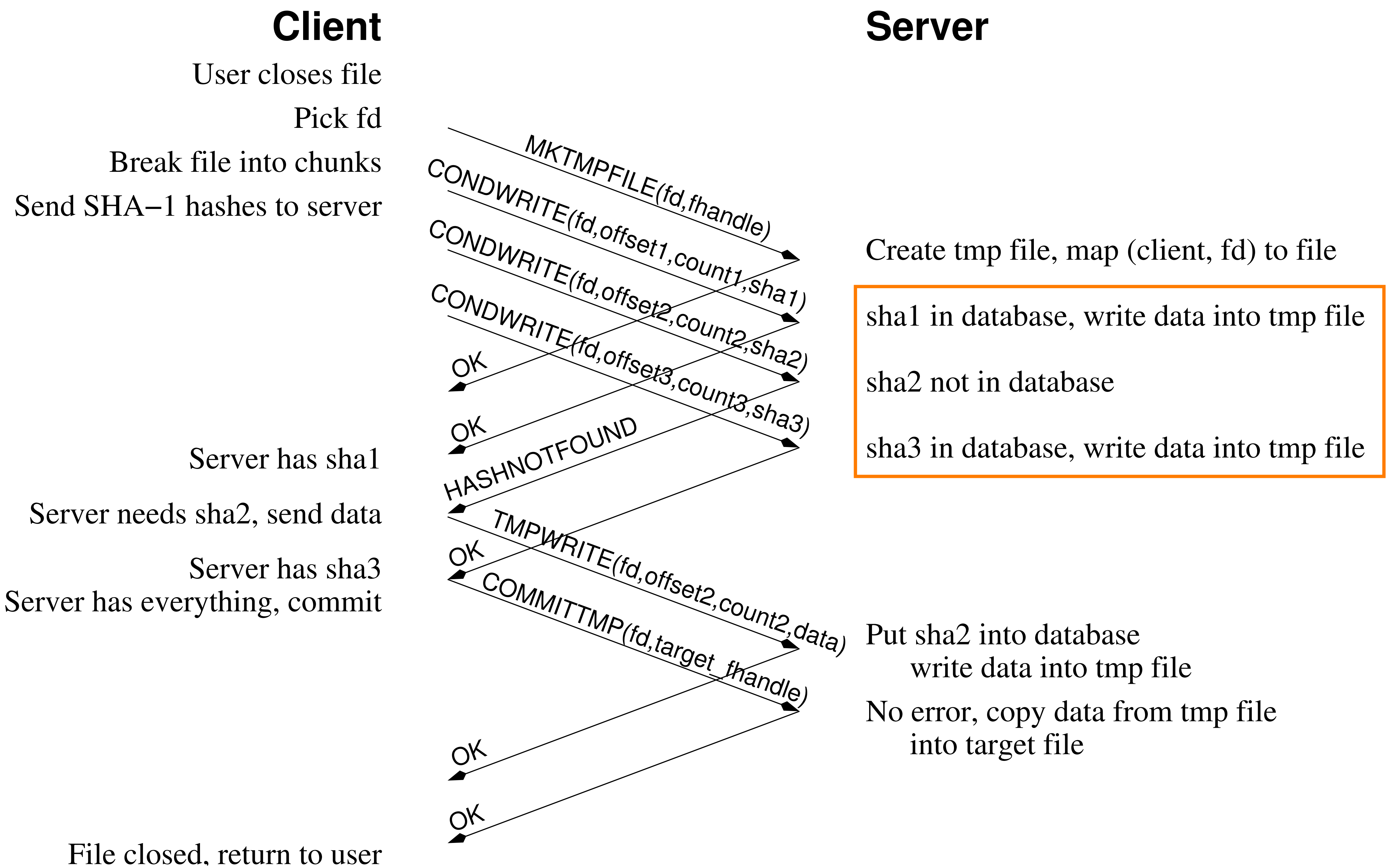
File writes


Client
User closes file
Pick fd
Break file into chunks
Send SHA−1 hashes to server
Server has sha1
Server needs sha2, send data
Server has sha3
Server has everything, commit
File closed, return to user
Server
MKTMPFILE(fd,fhandle)
CONDWRITE(fd,offset1,count1,sha1)
CONDWRITE(fd,offset2,count2,sha2)
CONDWRITE(fd,offset3,count3,sha3)
OK
OK
HASHNOTFOUND
OK
TMPWRITE(fd,offset2,count2,data)
COMMITTMP(fd,target_fhandle)
OK
OK
Create tmp file, map (client, fd) to file
sha1 in database, write data into tmp file
sha2 not in database
sha3 in database, write data into tmp file
Put sha2 into database
write data into tmp file
No error, copy data from tmp file
into target file

File writes


Client
User closes file
Pick fd
Break file into chunks
Server
Create tmp file, map (client, fd) to file
sha1 in database, write data into tmp file
sha2 not in database
sha3 in database, write data into tmp file
Target file must be assembled using the chunks
TMPWRITE(fd,offset2,count2,data)
OK
Server has sha3
Server has everything, commit
COMMITTMP(fd,target_fhandle)
Put sha2 into database
write data into tmp file
No error, copy data from tmp file
into target file
OK
OK
File closed, return to user

File writes


Client
User closes file
Pick fd
Break file into chunks
Send SHA−1 hashes to server
Server has sha1
Server needs sha2, send data
Server has sha3
Server has everything, commit
File closed, return to user
MKTMPFILE(fd,fhandle)
CONDWRITE(fd,offset1,count1,sha1)
CONDWRITE(fd,offset2,count2,sha2)
CONDWRITE(fd,offset3,count3,sha3)
OK
OK
HASHNOTFOUND
OK
TMPWRITE(fd,offset2,count2,data)
COMMITTMP(fd,target_fhandle)
OK
OK
Server
Create tmp file, map (client, fd) to file
sha1 in database, write data into tmp file
sha2 not in database
sha3 in database, write data into tmp file
Put sha2 into database
write data into tmp file
No error, copy data from tmp file
into target file

File writes

Client

User closes file

Pick fd

Break file into chunks

Send SHA−1 hashes to server

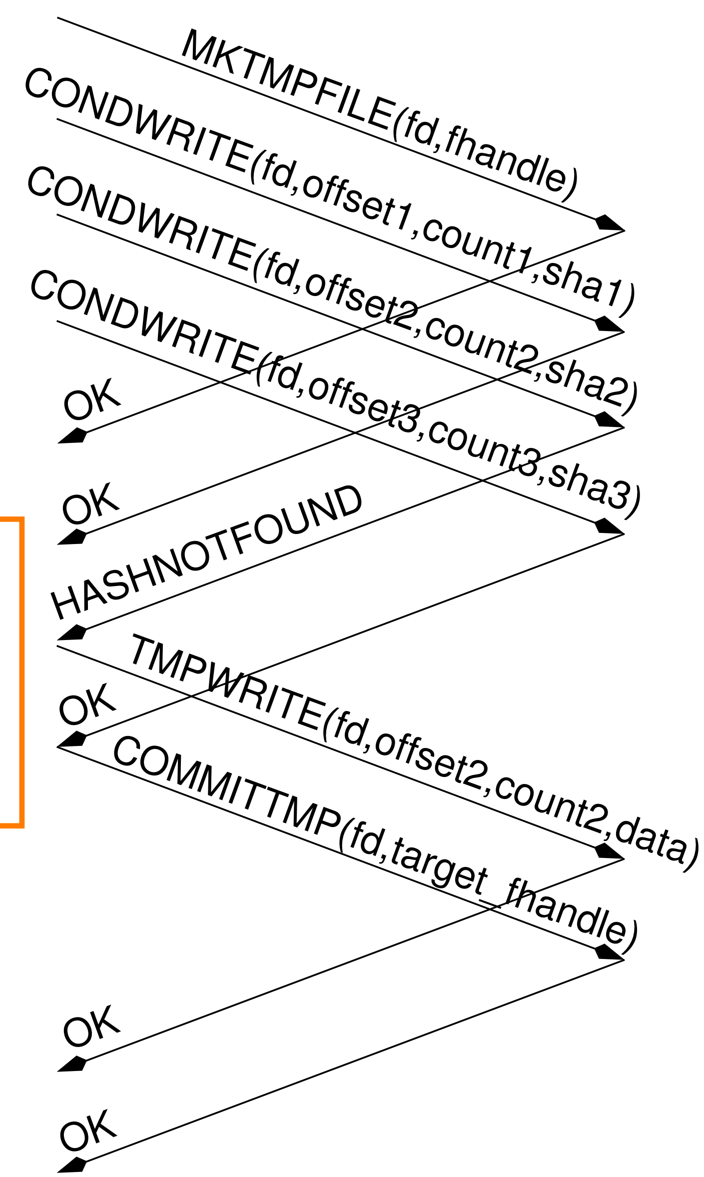
Server has sha1

Server needs sha2, send data

Server has sha3

Server has everything, commit

File closed, return to user



Server

Create tmp file, map (client, fd) to file

sha1 in database, write data into tmp file

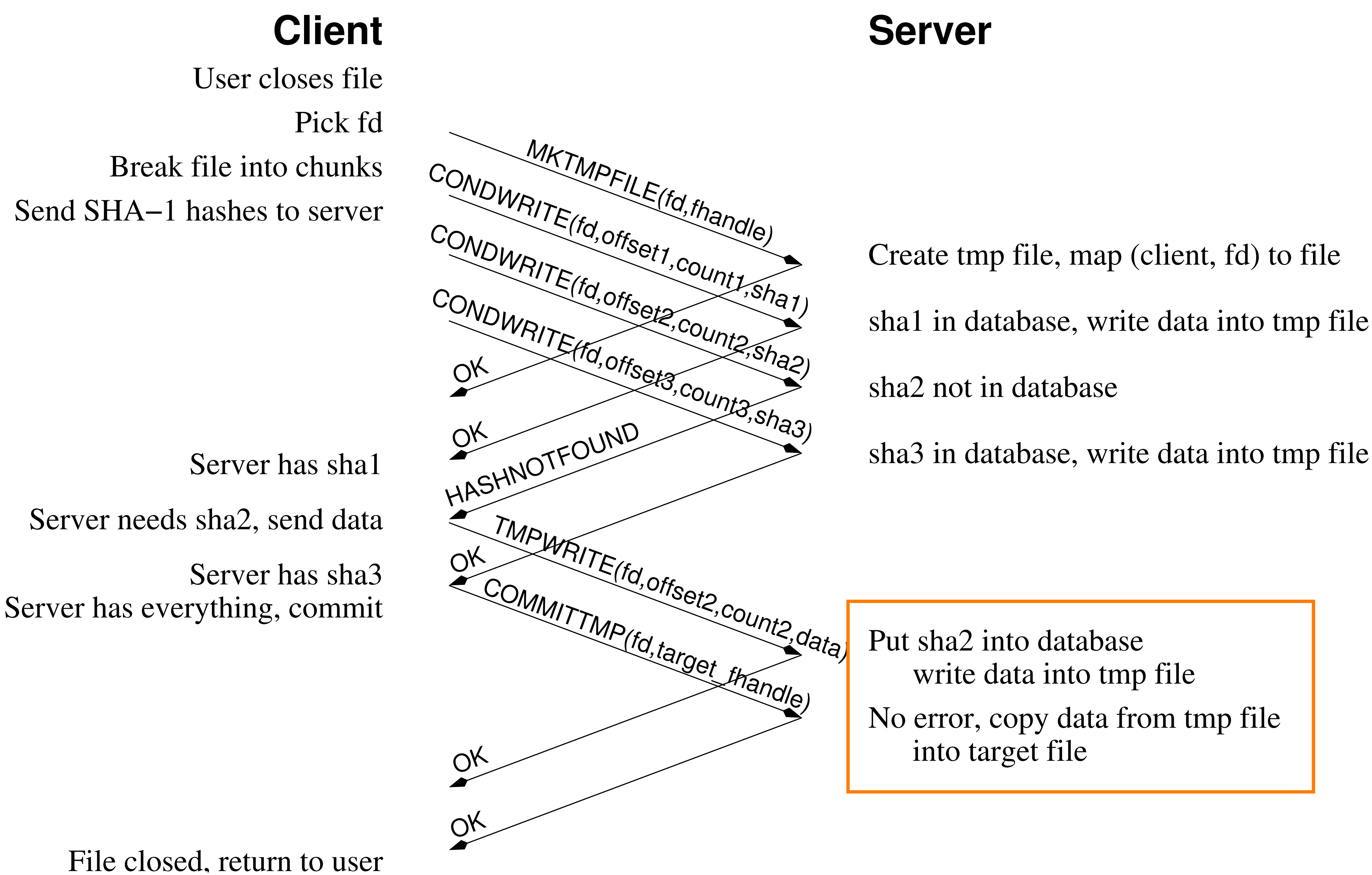
sha2 not in database

sha3 in database, write data into tmp file

Put sha2 into database
write data into tmp file

No error, copy data from tmp file
into target file

File writes


Client
User closes file
Pick fd
Break file into chunks
Send SHA−1 hashes to server
Server has sha1
Server needs sha2, send data
Server has sha3
Server has everything, commit
File closed, return to user
Server
Create tmp file, map (client, fd) to file
sha1 in database, write data into tmp file
sha2 not in database
sha3 in database, write data into tmp file
Put sha2 into database
write data into tmp file
No error, copy data from tmp file
into target file
MKTMPFILE(fd,fhandle)
CONDWRITE(fd,offset1,count1,sha1)
CONDWRITE(fd,offset2,count2,sha2)
CONDWRITE(fd,offset3,count3,sha3)
OK
OK
HASHNOTFOUND
OK
TMPWRITE(fd,offset2,count2,data)
COMMITTMP(fd,target_fhandle)
OK
OK

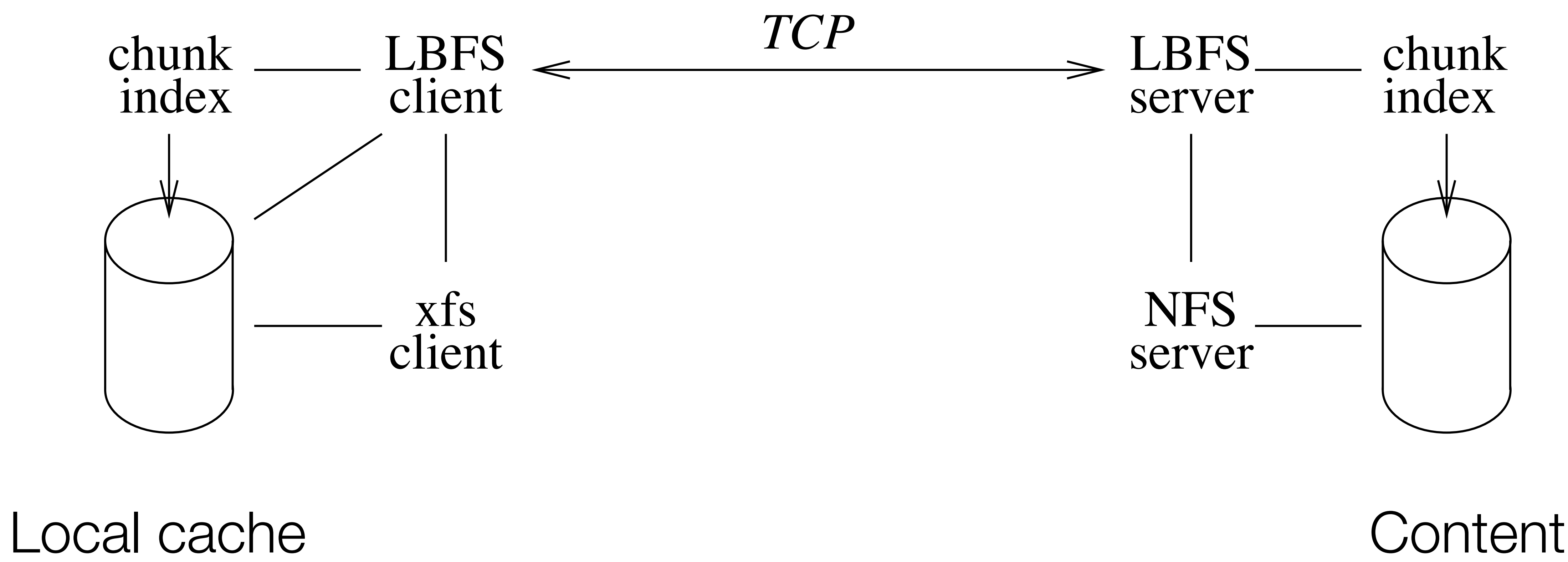
Security

- Every server has a public key
- Messages (protocol) are compressed, tagged with an authentication code, and then encrypted
- At mount time, the client and server negotiate a session key

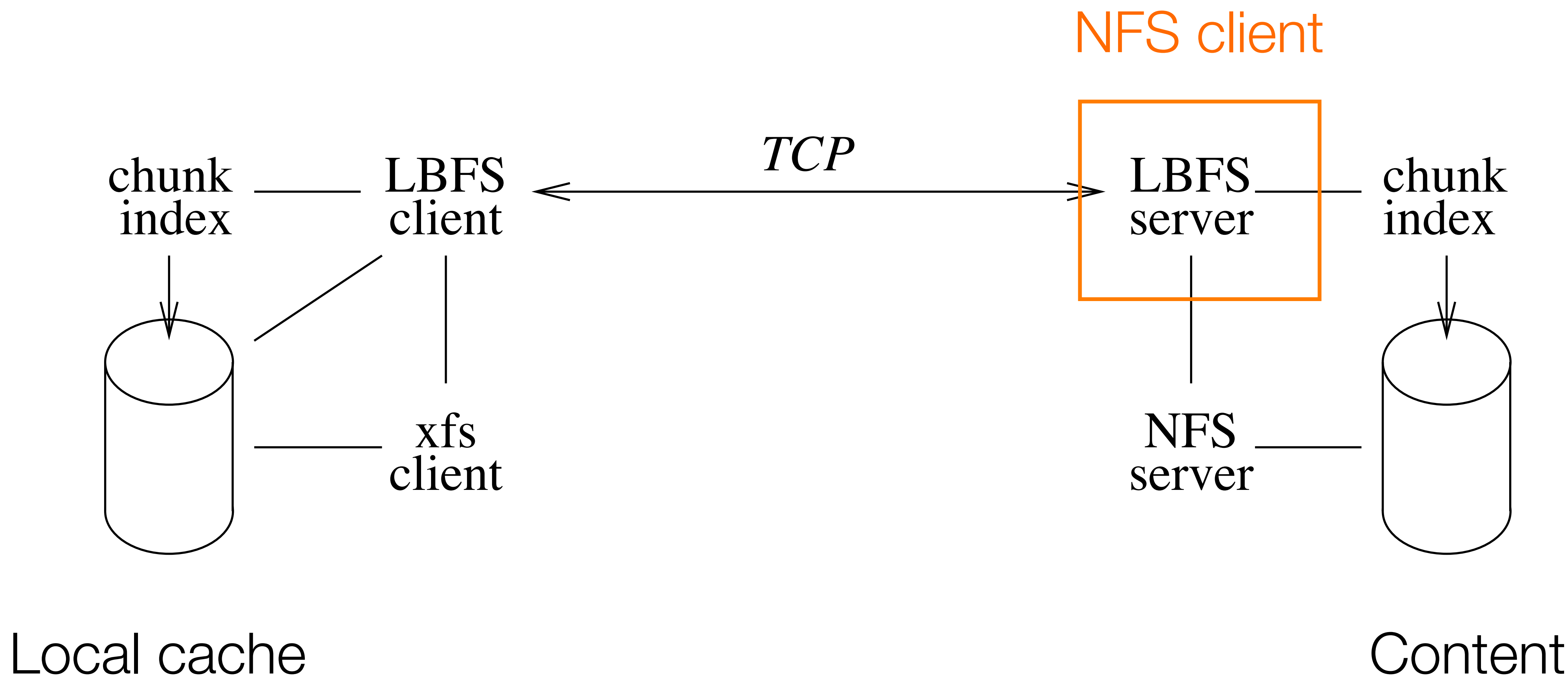
Implementation

- Server and client run at user-level
- Server accesses files through NFS
- Client and server communicates over TCP
- Client implements the file system using XFS

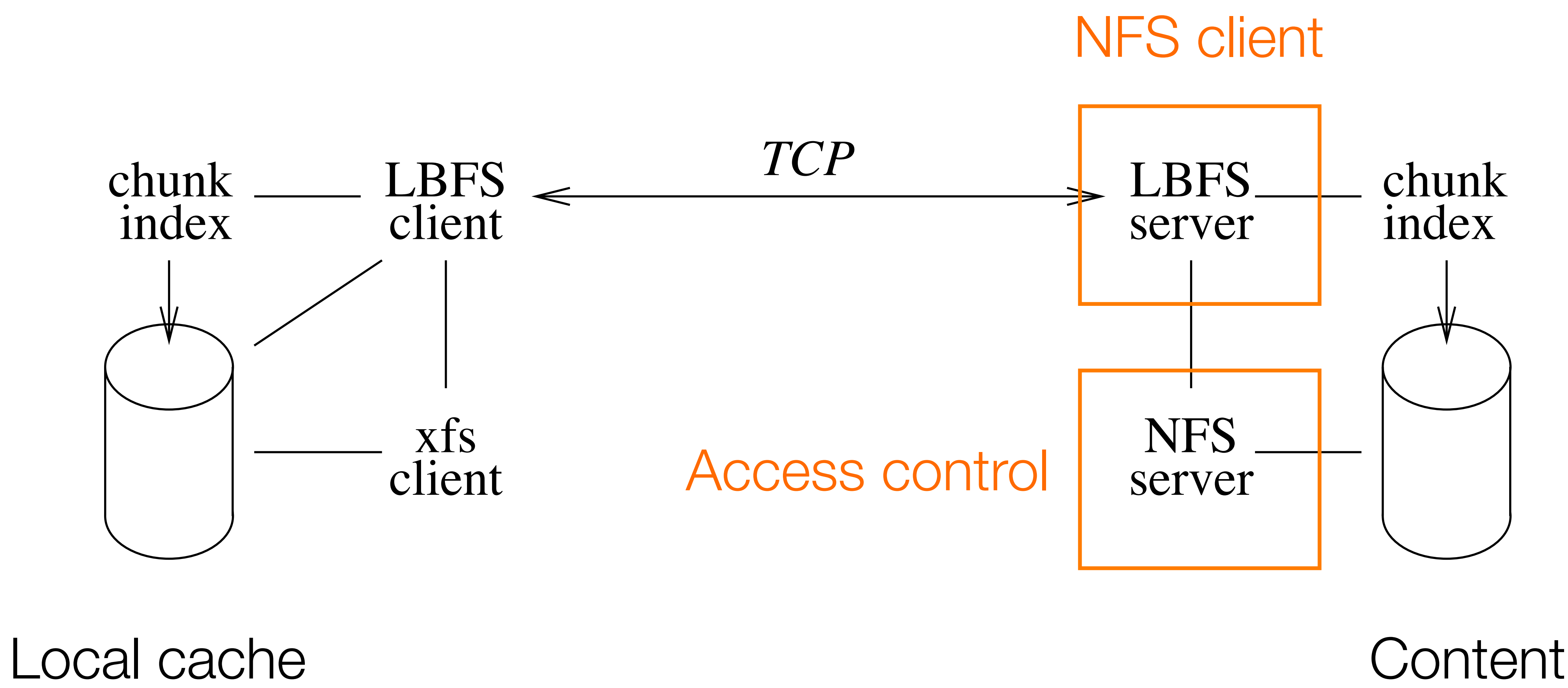
Implementation



Implementation



Implementation



Evaluation

- Bandwidth consumption and network utilization of LBFS under several common workloads
- Comparison to CIFS, NFS version 3, and AFS

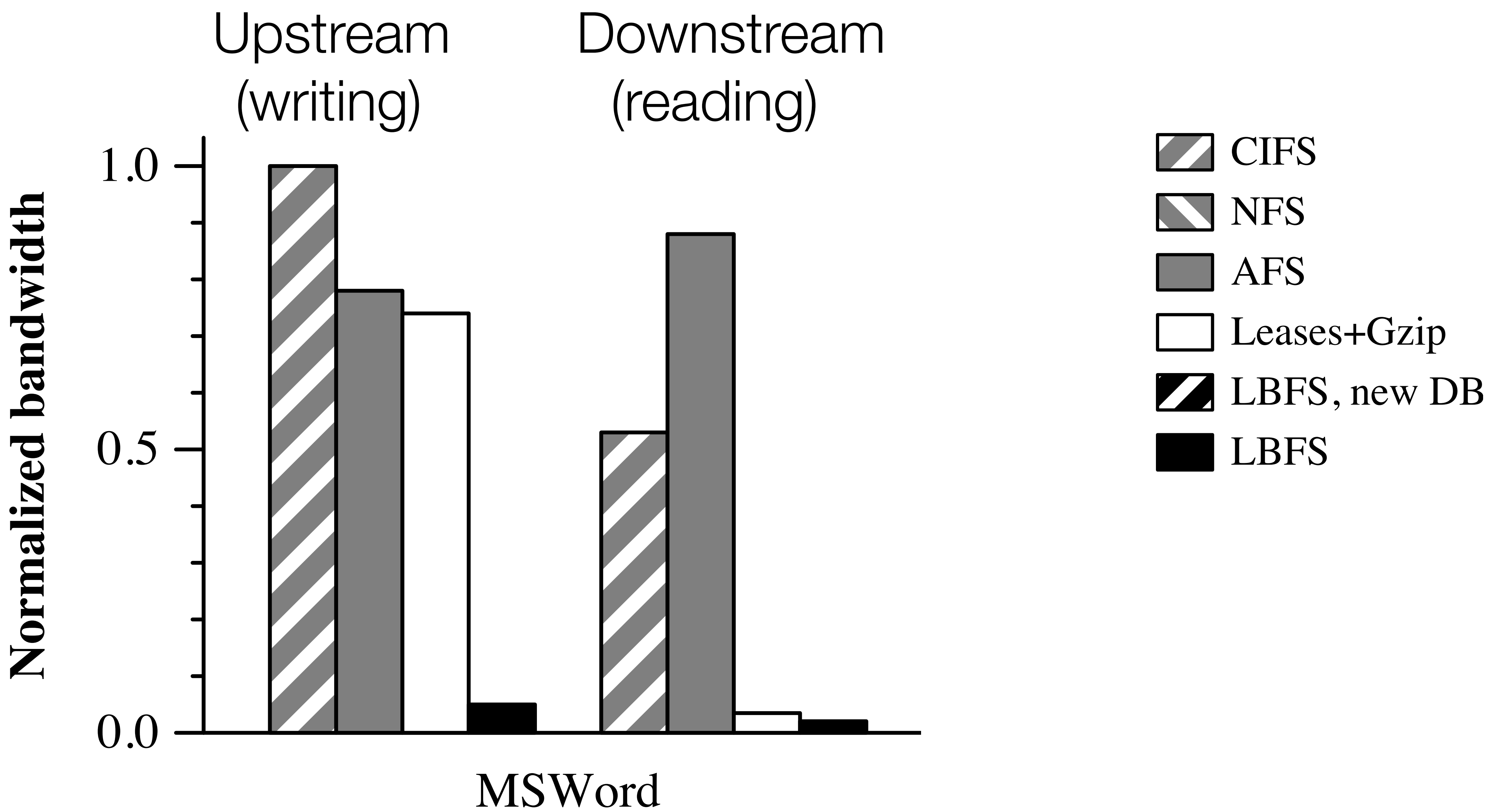
Evaluation - workloads

- **MSWord**: open a MSWord document (1.4MByte) about Windows 2000 and edit its references

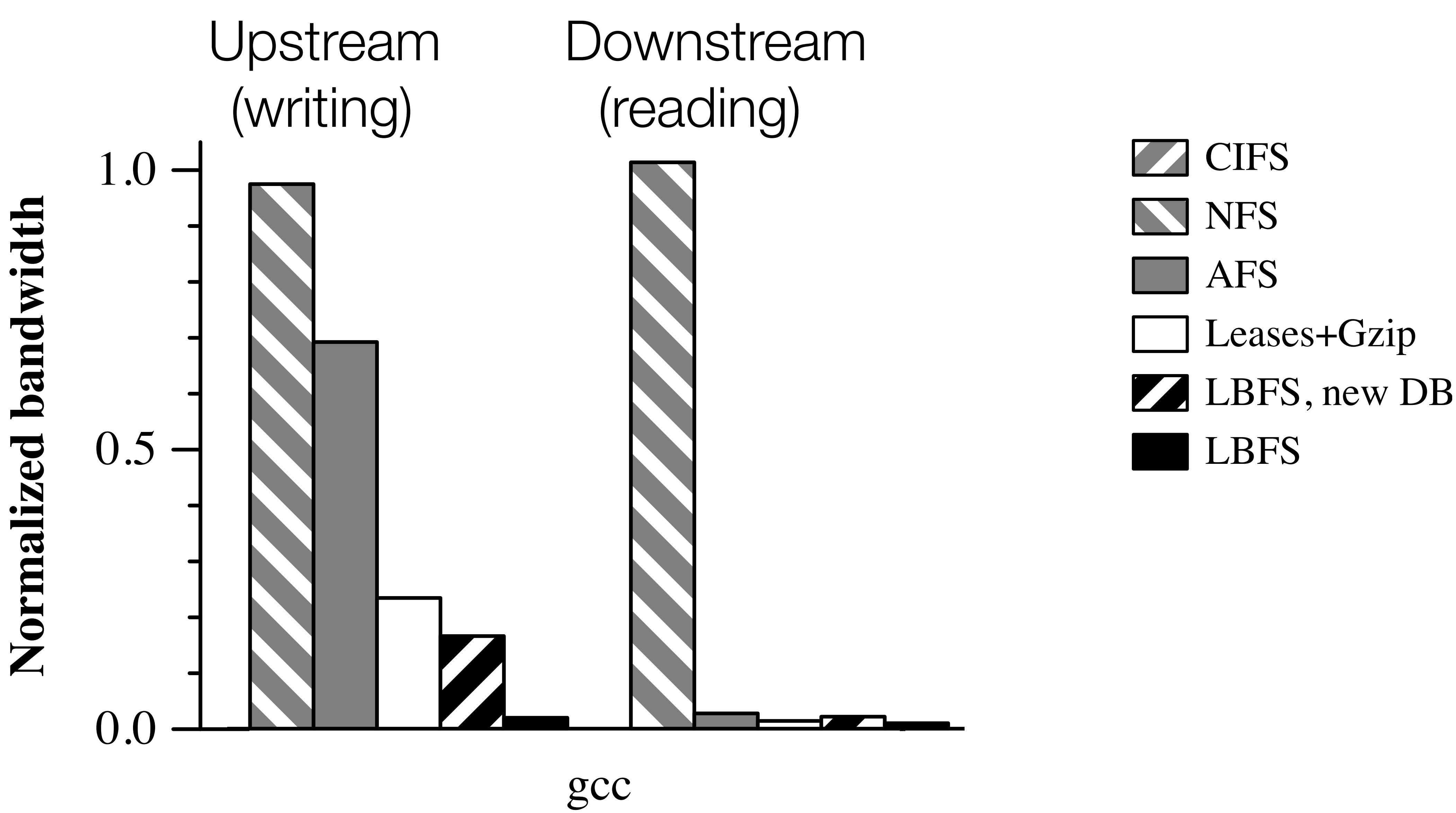
- **gcc**: recompile emacs 20.7 from source, after modifying a header file

- **ed**: transform the perl 5.6.0 source tree into perl 5.6.1

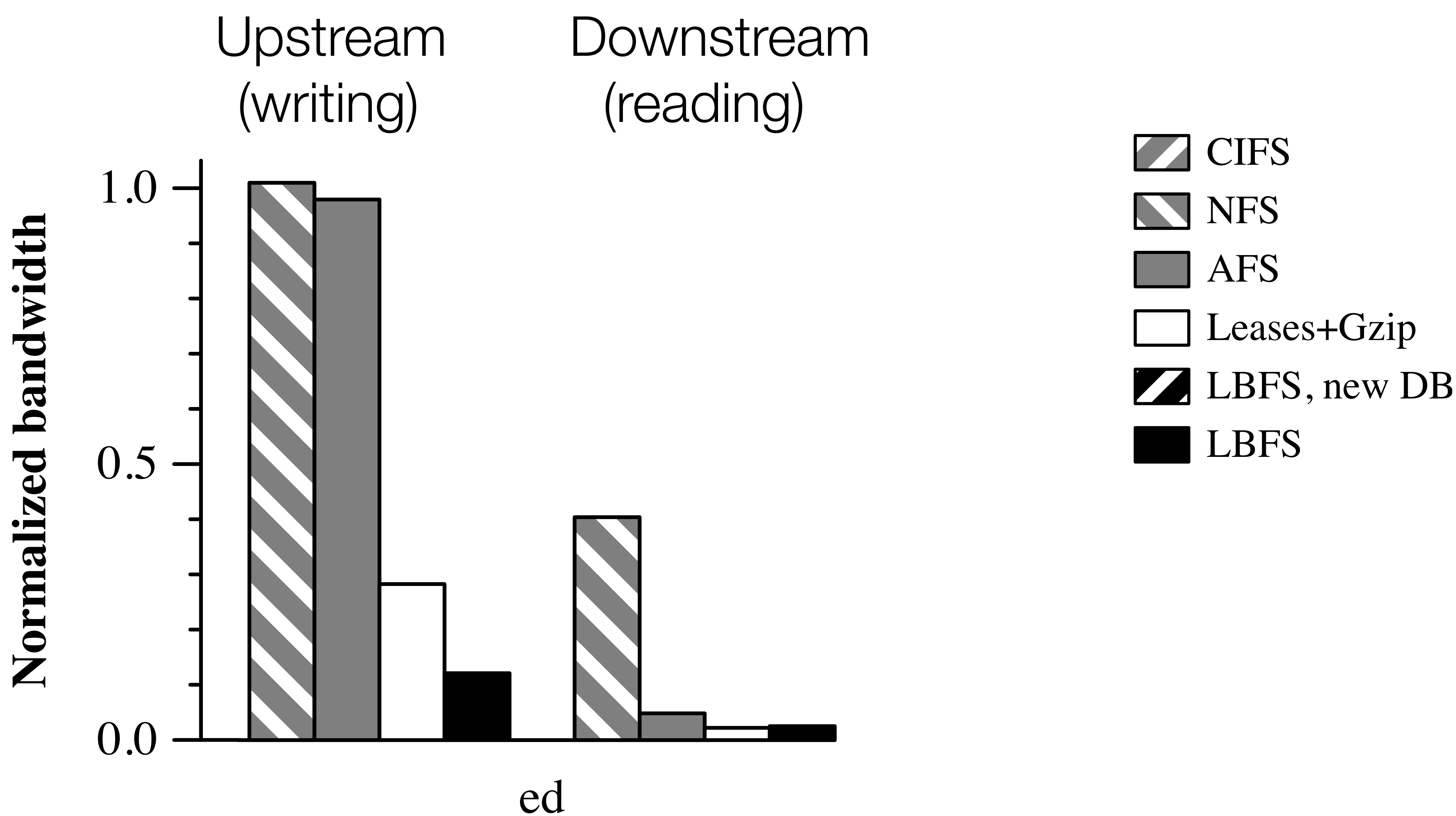
Bandwidth utilization



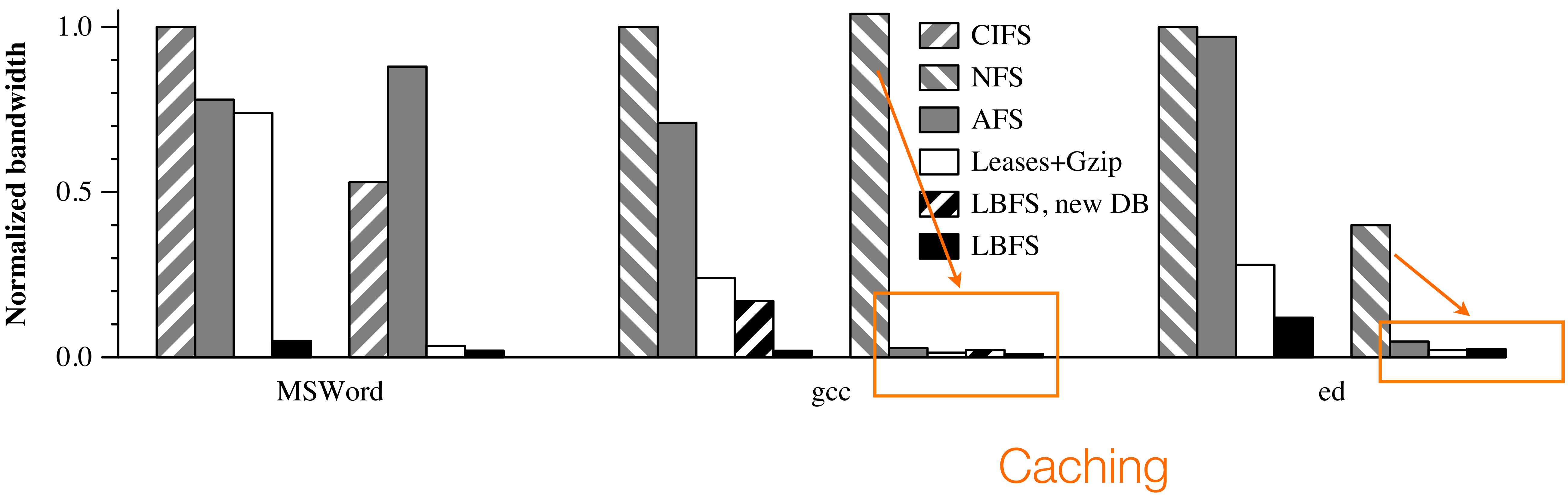
Bandwidth utilization



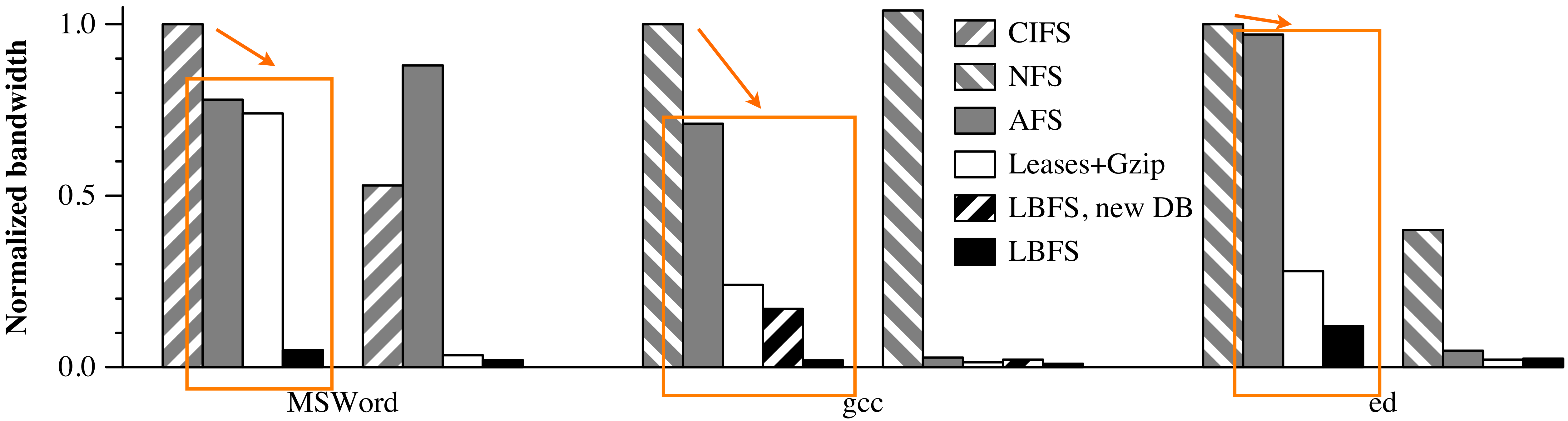
Bandwidth utilization



Bandwidth utilization (downstream)

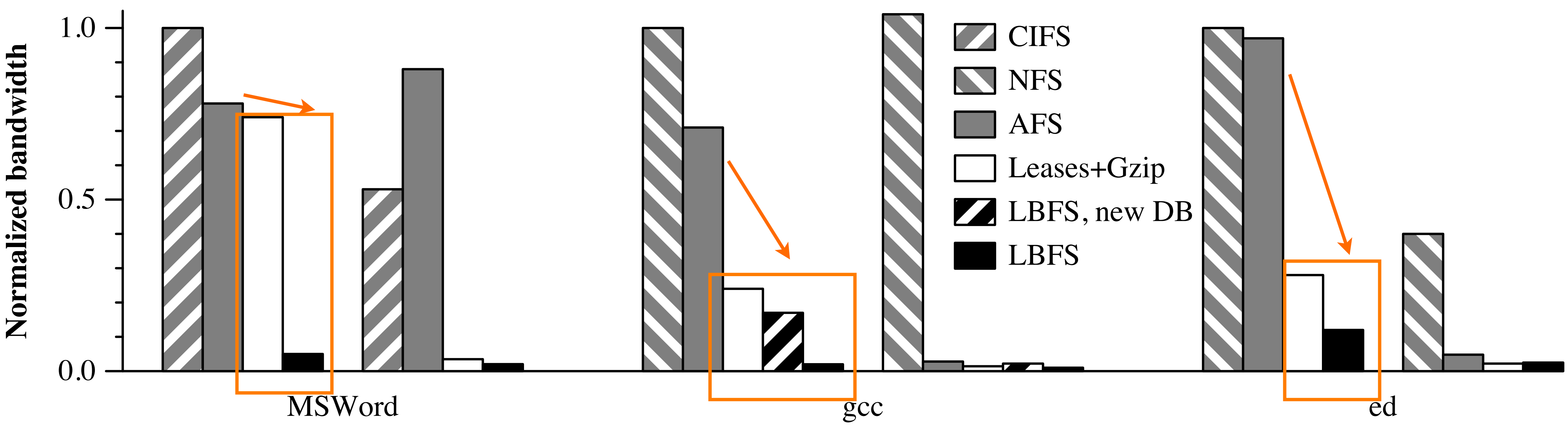


Bandwidth utilization (upstream)



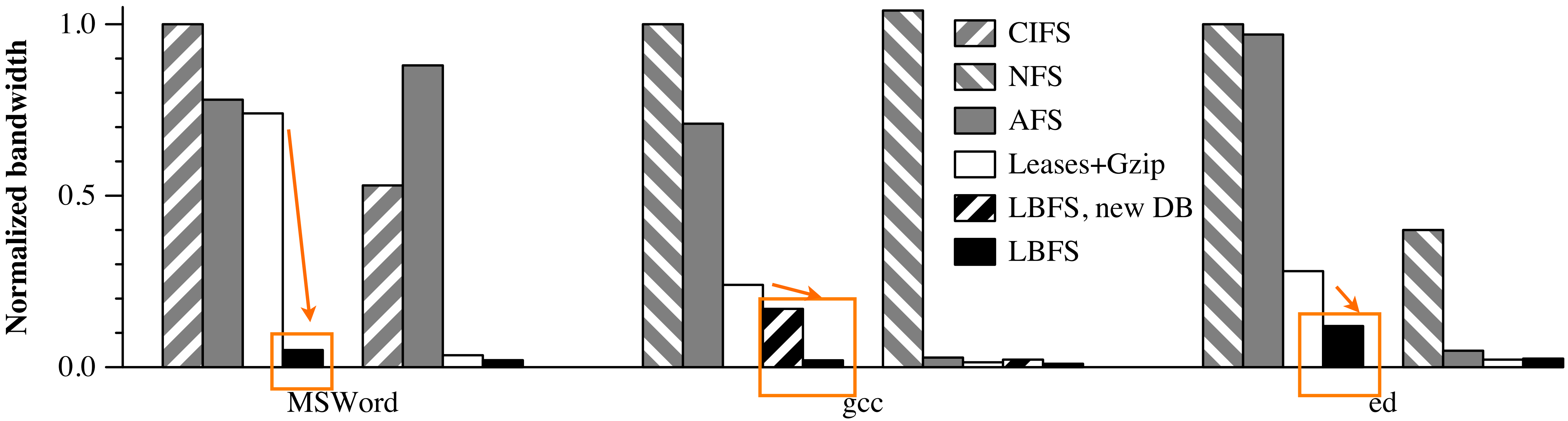
Writes are deferred to close time

Bandwidth utilization (upstream)



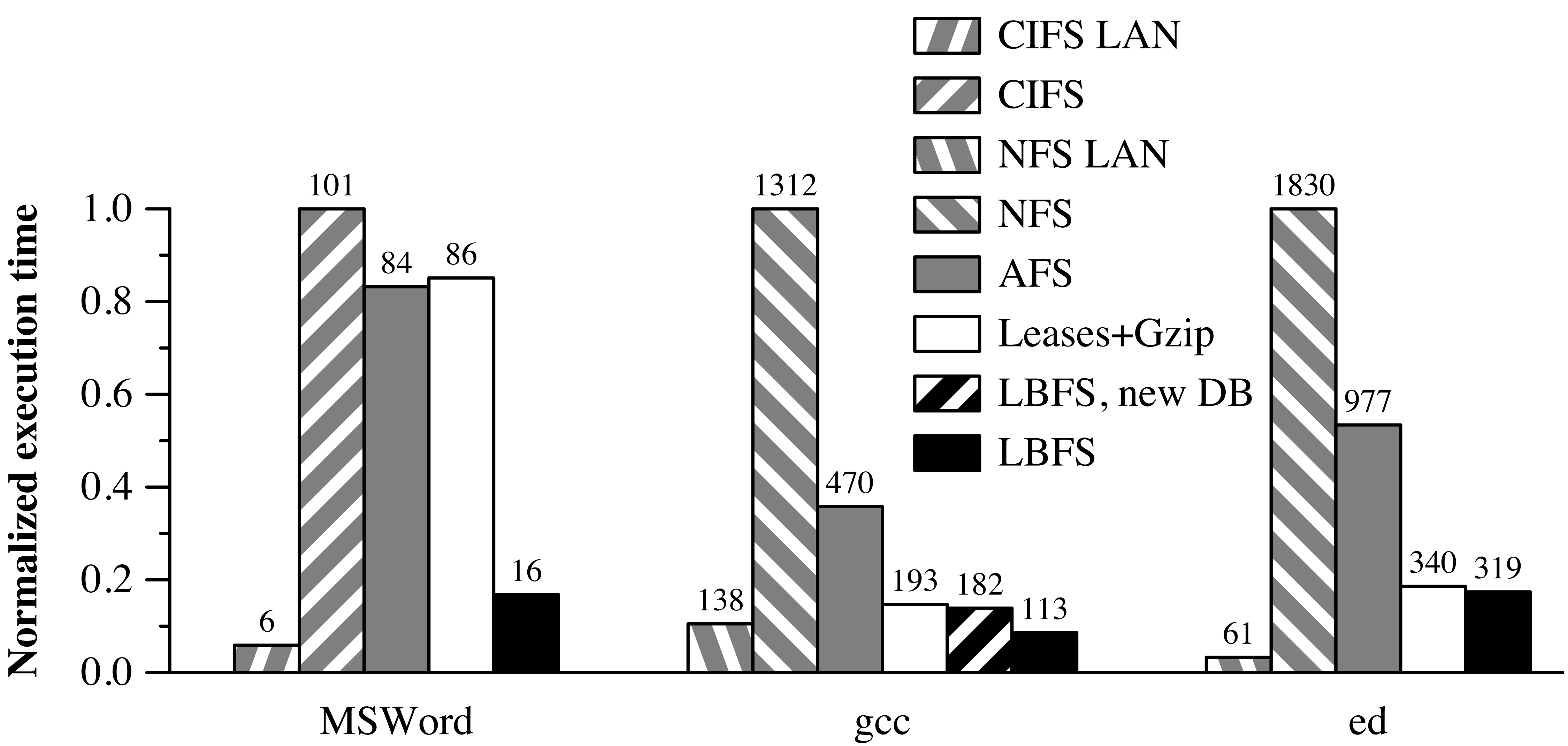
Compression

Bandwidth utilization (upstream)

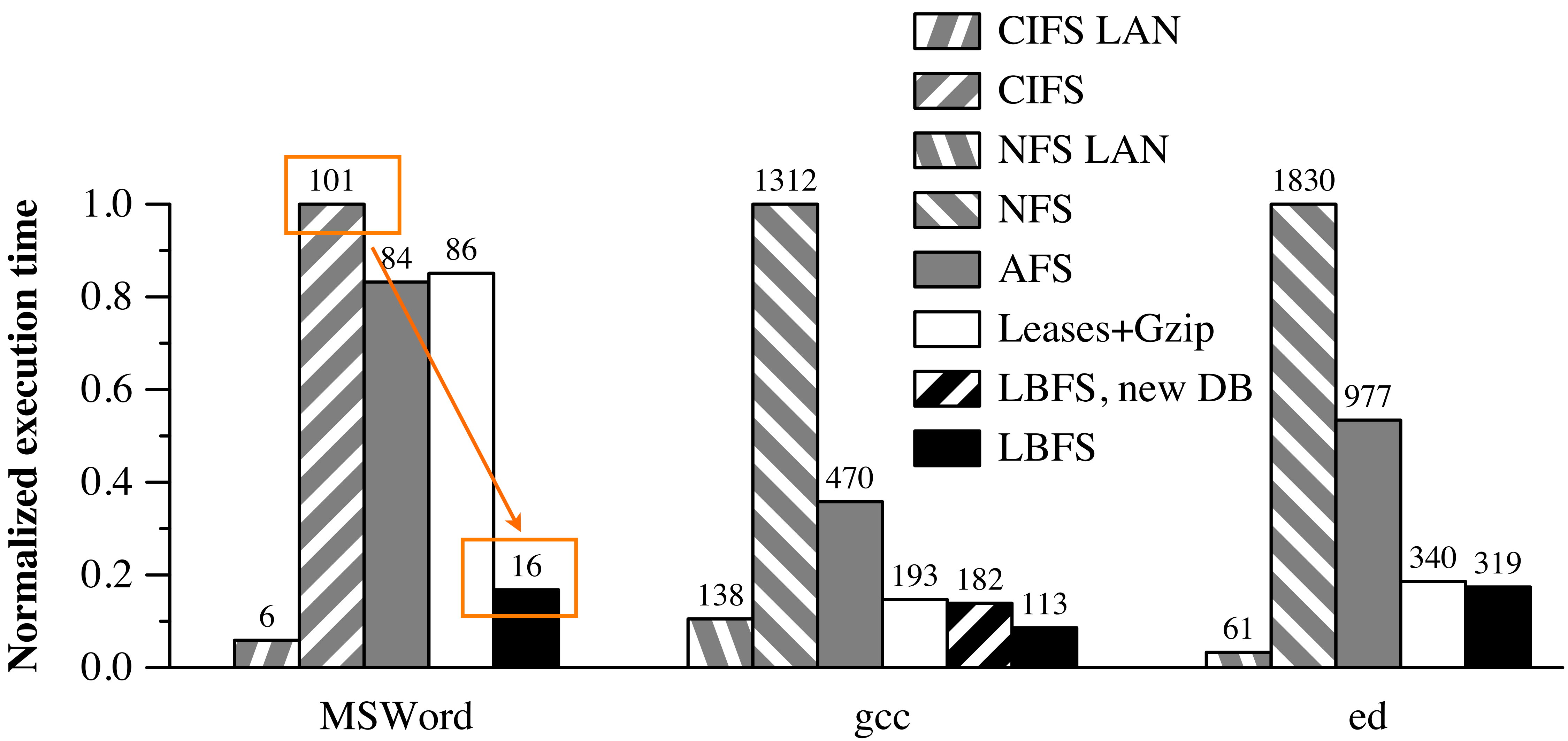


Chunking

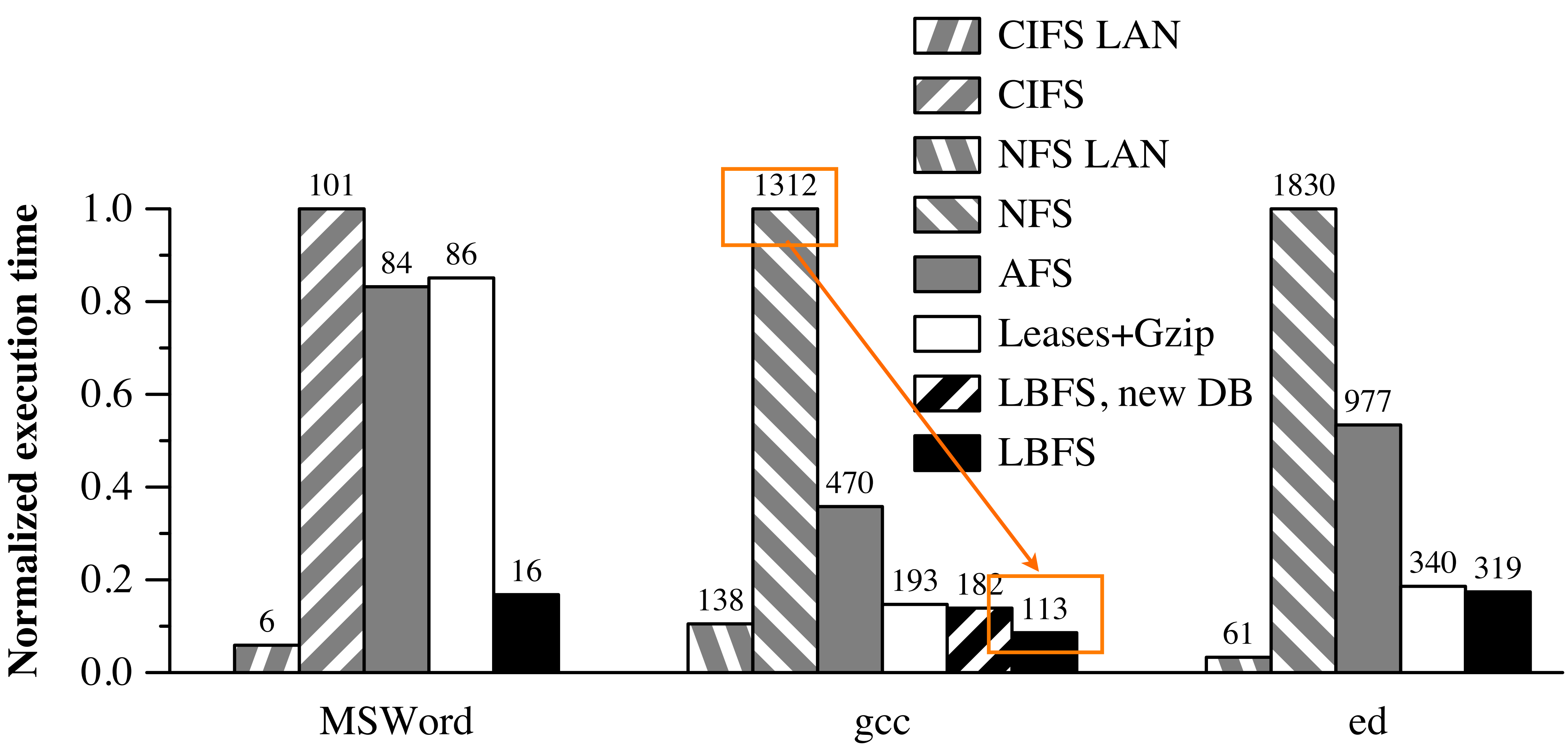
Application performance



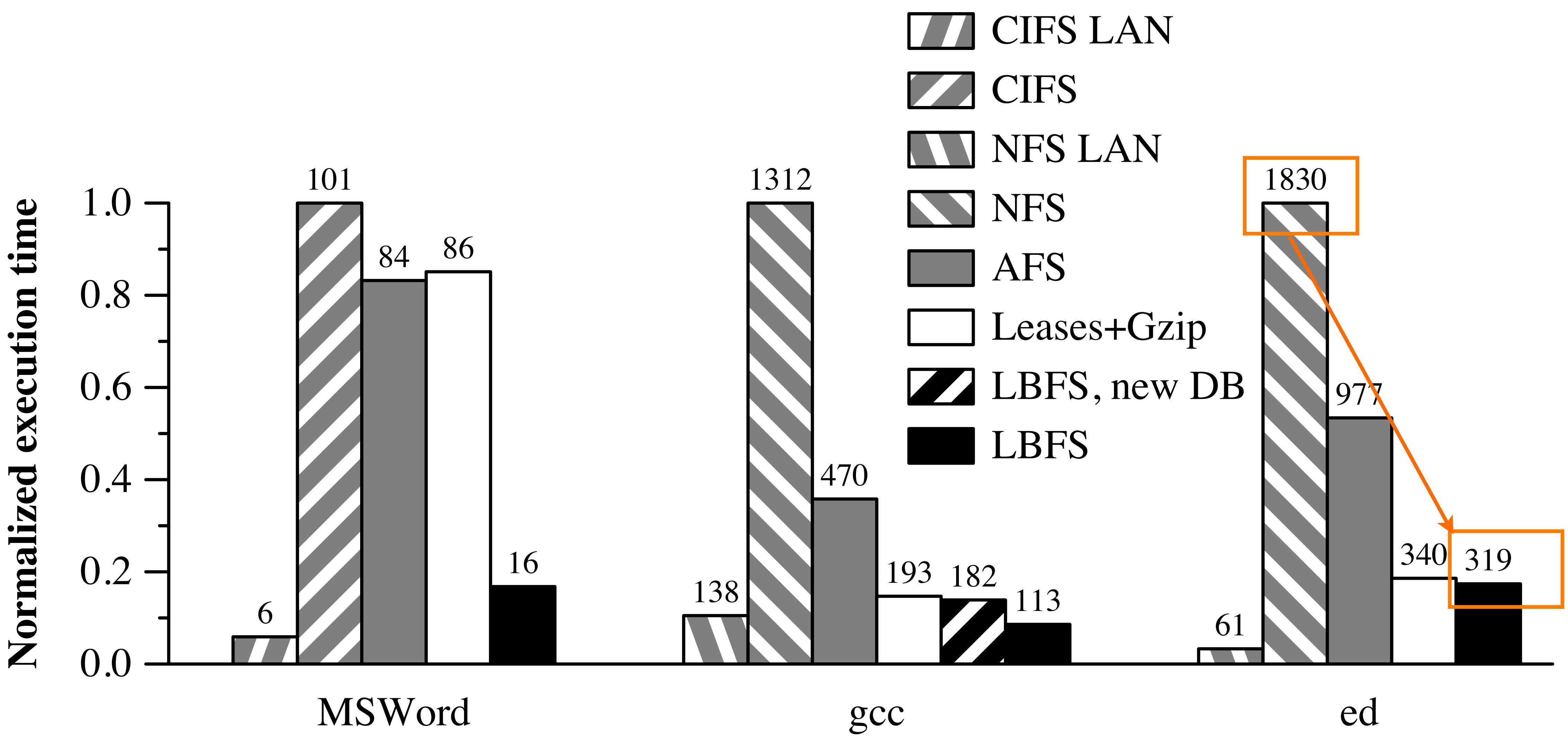
Application performance



Application performance



Application performance



Summary

- **LBFS** breaks files into chunks based on contents
- **LBFS** indexes file chunks by their hash values
- **LBFS** saves bandwidth by taking advantage of commonality between files

Summary

- **LBFS** can consume over an order of magnitude less bandwidth than traditional file systems
- **LBFS** makes transparent remote file access a viable and less frustrating alternative