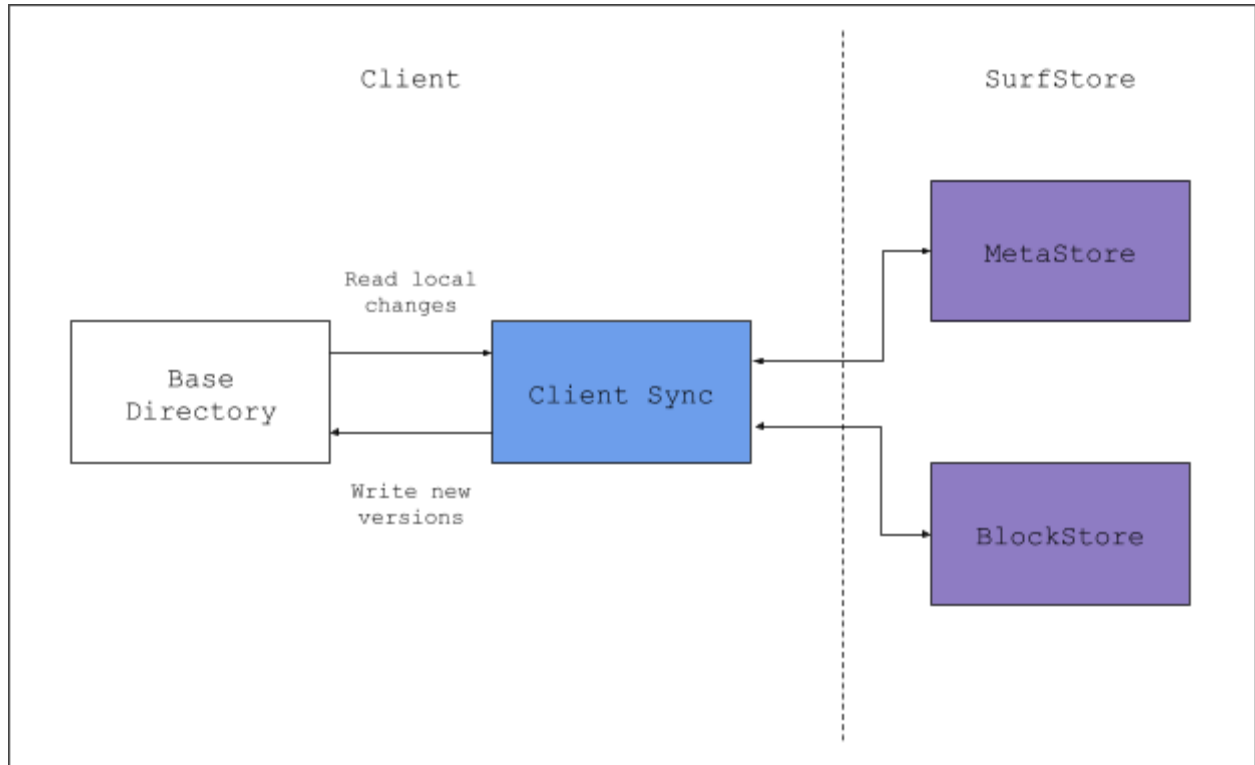# Project 3: SurfStore

Overview

In this project, you are going to create a cloud-based file storage service called SurfStore. SurfStore is a networked file storage application that is based on Dropbox, and lets you sync files to and from the "cloud". You will implement the cloud service, and a client which interacts with your service via RPC.

Multiple clients can concurrently connect to the SurfStore service to access a common, shared set of files. Clients accessing SurfStore "see" a consistent set of updates to files, but SurfStore does not offer any guarantees about operations across files, meaning that it does not support multi-file transactions (such as atomic move).

The SurfStore service is composed of the following two services:

BlockStore	The content of each file in SurfStore is divided up into chunks, or blocks, each of which has a unique identifier. This service stores these blocks, and when given an identifier, retrieves and returns the appropriate block.
MetaStore	The MetaStore service manages the metadata of files and the entire system. Most importantly, the MetaStore service holds the mapping of filenames to blocks. Furthermore, it should be aware of available BlockStores and map blocks to particular BlockStores. In a real deployment, a cloud file service like Dropbox or Google Drive will hold exabytes of data, and so will require 10s of thousands of BlockStores or more to hold all that data.

Fundamentals

In this section, we'll go over some of the fundamentals of SurfStore: blocks, files, and versioning.

Blocks, hashes, and hashlists

A file in SurfStore is broken into an ordered sequence of one or more blocks. Each block is of uniform size (defined by the command line argument), except for the last block in the file, which may be smaller (but must be at least 1 byte large). As an example, assume the block size is 4096 bytes, and consider the following file:
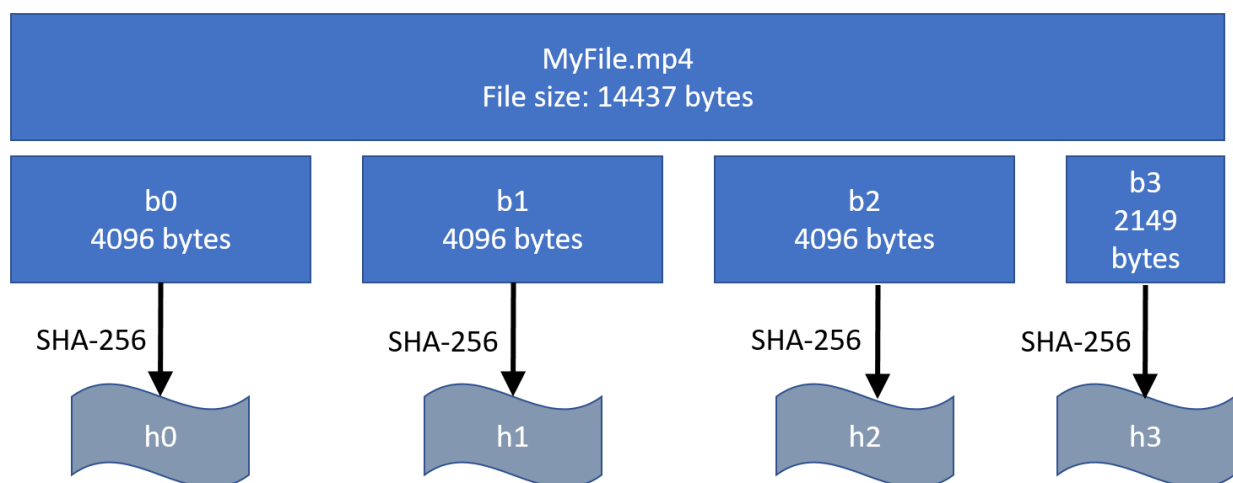
The file 'MyFile.mp4' is 14,437 bytes long, and the block size is 4KB. The file is broken into blocks b0, b1, b2, and b3 (which is only 2,149 bytes long). For each block, a hash value is generated using the **SHA-256** hash function. So for MyFile.mp4, those hashes will be denoted

as [h0, h1, h2, h3] in the same order as the blocks. This set of hash values, in order, represents the file, and is referred to as the *hashlist*. Note that if you are given a block, you can compute its hash by applying the **SHA-256** hash function to the block. This also means that if you change data in a block the hash value will change as a result. To update a file, you change a subset of the bytes in the file, and recompute the hashlist. Depending on the modification, at least one, but perhaps all, of the hash values in the hashlist will change

Files and Directories

Files and Filenames

Files in SurfStore are denoted by filenames, which are represented as strings. For example "MyDog.jpg", "WinterVacation.mp4", and "Expenses.txt" are all examples of filenames. SurfStore doesn't have a concept of a directory or directory hierarchy–filenames are just strings. For this reason, filenames can only be compared for equality or inequality, and there are no "cd" or "mkdir" commands. Filenames are case sensitive, meaning that "Myfile.jpg" is different than "myfile.jpg". Filenames can contain spaces, but as described below, cannot contain commas ',' or the forward slash '/', and cannot be named index.txt.

The Base Directory

A command-line argument specifies a "base directory" for the client. This is the directory that is going to be synchronized with your cloud-based service. Your client will upload files from this base directory to the cloud, and download files (and changes to files) from the cloud into this base directory. Your client should not modify any files outside of this base directory. Note in particular–your client should not download files into the "current" directory, only the base directory specified by that command line argument. This base directory will contain zero or more files, but won't have any subdirectories.

Versioning

File Versions

Each file/filename is associated with a *version*, which is a monotonically increasing positive integer. The version is incremented any time the file is created, modified, or deleted. The purpose of the version is so that clients can detect when they have an out-of-date view of the file hierarchy.

For example, imagine that Tahani wants to update a spreadsheet file that tracks conference room reservations. Ideally, they would perform the following actions:

One client

www.websequencediagrams.com
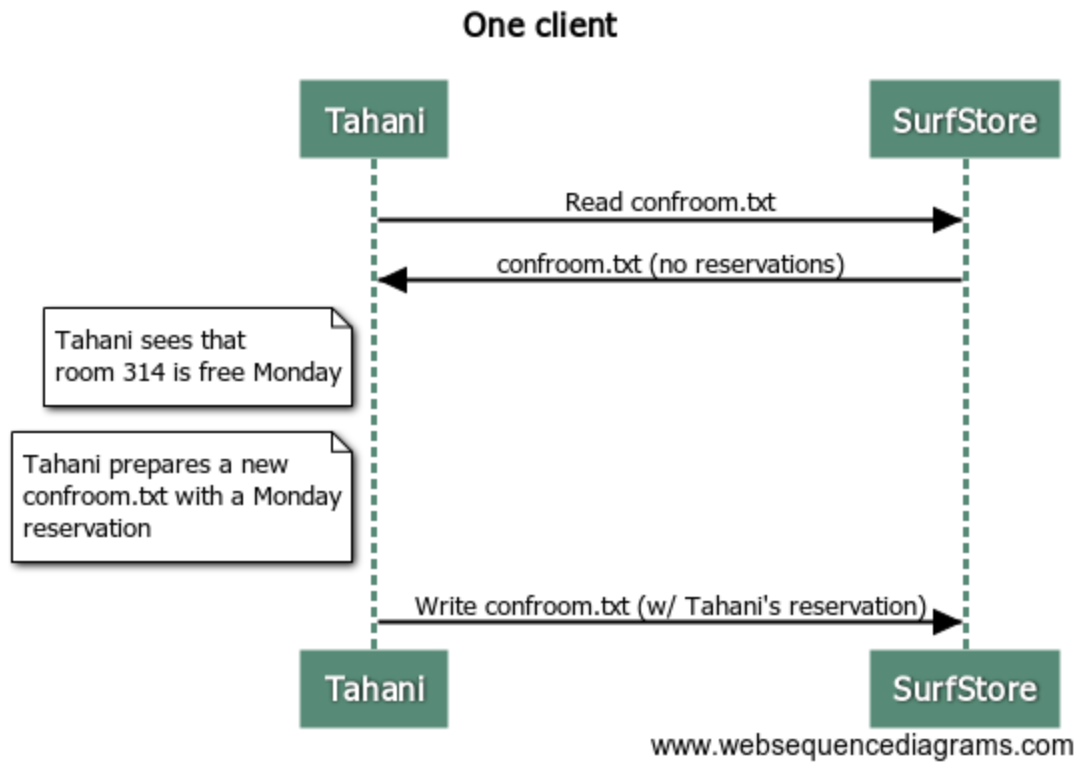
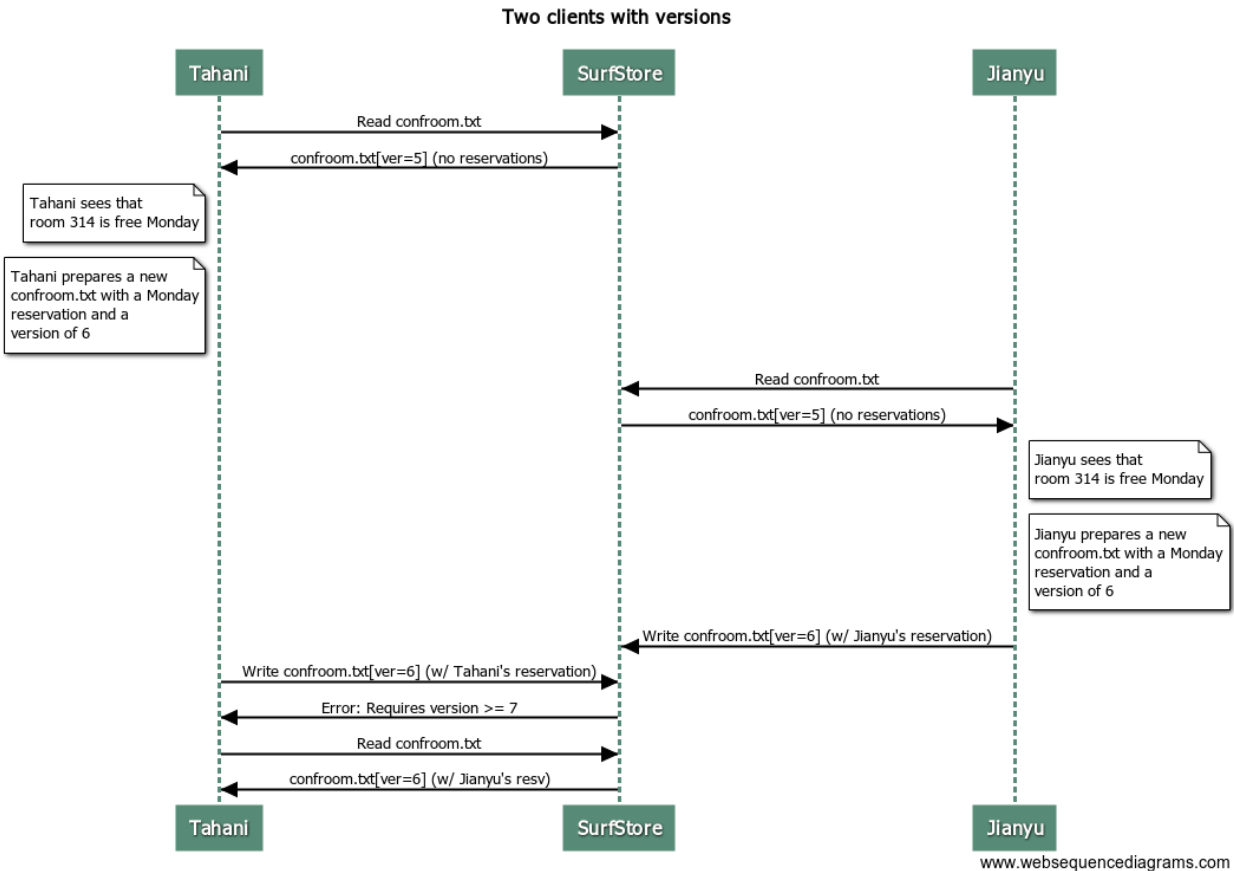However, another client might be concurrently modifying this file as well. In reality, the order of operations might be:

Two clients
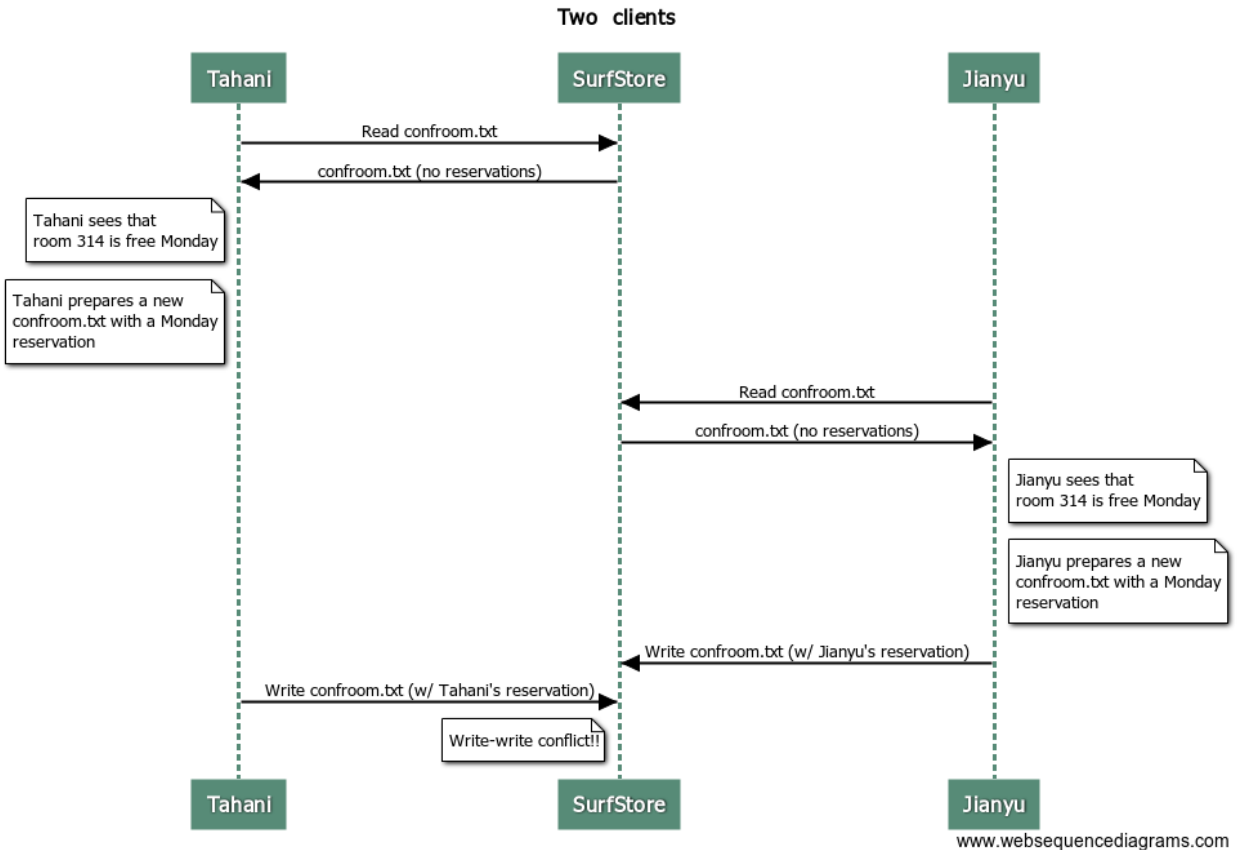
As you can see, Tahani overwrote the change that Jianyu made without realizing it. We can solve this problem with file *versions*. Every time a file is modified, its version number is incremented. SurfStore only records modifications to files if the version is **exactly** one larger than the currently recorded version. Let's see what would happen in the two-client case:

Two clients with versions

Tahani	SurfStore	Jianyu

Tahani → SurfStore: Read confroom.txt

SurfStore → Tahani: confroom.txt[ver=5] (no reservations)

> Tahani sees that room 314 is free Monday

> Tahani prepares a new confroom.txt with a Monday reservation and a version of 6

Jianyu → SurfStore: Read confroom.txt

SurfStore → Jianyu: confroom.txt[ver=5] (no reservations)

> Jianyu sees that room 314 is free Monday

> Jianyu prepares a new confroom.txt with a Monday reservation and a version of 6

Jianyu → SurfStore: Write confroom.txt[ver=6] (w/ Jianyu's reservation)

Tahani → SurfStore: Write confroom.txt[ver=6] (w/ Tahani's reservation)

SurfStore → Tahani: Error: Requires version >= 7

Tahani → SurfStore: Read confroom.txt

SurfStore → Tahani: confroom.txt[ver=6] (w/ Jianyu's resv)

www.websequencediagrams.com

.

In the above example, both Tahani and Jianyu downloaded identical copies of confroom.txt (at version 5). They then both started editing their local copies of the file. So there was a point where Tahani had "her own" version 6 of the file (with her local changes), and Jianyu had "his own" version 6 of the file (with his local changes). How do we know whose "version 6" is the real version 6?

The answer is that whoever syncs their changes to the cloud first wins. So in this example, Jianyu was first to sync his changes to the cloud, which caused his modifications to the file to become the official version 6 of the file. Later, when Tahani tries to upload her changes, she realizes that Jianyu beat her to it, and so Jianyu's changes to the file will overwrite her copy.

Deleting files

To delete a file, the MetadataStore service records a versioned "tombstone" update. This update simply indicates that the file has been deleted. In this way, deletion events also require version numbers, which prevents race conditions that can occur when one client deletes a file concurrently with another client deleting that file. Note that this means that a deleted file must be

recreated before it can be read by a client again. If a client tries to delete a file that has already been deleted, that is fine–just handle the version numbers of these tombstone updates appropriately.

To represent a "tombstone" record, we will set the file's hash list to a single hash value of "0" (a string with one character which is the 0 character).

Specification

Client-Side

Client Sync

Your client will "sync" a local base directory with your SurfStore cloud service. When you invoke your client, the sync operation will occur, and then the client will exit. As a result of syncing, new files added to your base directory will be uploaded to the cloud, files that were sync'd to the cloud from other clients will be downloaded to your base directory, and any files which have "edit conflicts'' will be resolved. Details on the general flow and resolving conflicts will be described below in [Algorithm](#).

A simple example, assuming that Tahani keeps her files in /tdata, and Jianyu keeps his files in /jdata:

```
tahani $ ls /tdata

tahani $
(Tahani's base directory starts empty)

tahani $ cp ~/kitten.jpg /tdata

tahani $ ./run-client myserver.ucsd.edu:5001 /tdata 4096
(syncs kitten.jpg to the server hosted on myserver.ucsd.edu port 5001,
using /tdata as the base directory, with a block size of 4096 bytes)

jianyu$ ls /jdata

jianyu $
(Jianu's base directory starts empty)

jianyu $ ./run-client myserver.ucsd.edu:5001 /jdata 4096
(kitten.jpg gets sync'd from the server hosted on myserver.ucsd.edu port
5001, using /jdata as the base directory, with a block size of 4096 bytes)
```

```
jianyu $ ls /jdata
kitten.jpg index.txt
```

index.txt

Your client program will create and maintain an index.txt file in the base directory which holds local, client-specific information that must be kept between invocations of the client. If that file doesn't exist, your client should create it. In particular, the index.txt contains a copy of the server's FileInfoMap accurate as of the last time that sync was called. The purpose of this index file is to detect files that have changed, or been added to the base directory since the last time that the client executed.

The format of the index.txt file should be one line per file, and each line should have the filename, the version, and then the hash list. The filename is separated from the version number with a comma, and the version number is separated from the hash list with a comma. Entries in the hash list should be separated by spaces. For example:

```
File1.dat,3,h0 h1 h2 h3
File2.jpg,8,h8 h9
```

Note that the name "index.txt" is special, and so our system does not allow you to sync regular files with that name. In reality, a client like Dropbox stores metadata hashes in a hidden file (e.g. .index) but we're just going to use a regular file.

BlockStore Service

The BlockStore server stores the contents of each block of data, indexed by its hash value. It supports basic get and put operations. It does not need to support deleting blocks of data–we just let unused blocks remain in the store. The BlockStore service only knows about blocks–it doesn't know anything about how blocks relate to files.

The service implements the following API:

PutBlock(b)	Stores block b in the key-value store, indexed by hash value h
b = **GetBlock**(h)	Retrieves a block indexed by hash value h
hashlist_out = **HasBlocks**(*hashlist_in*)	Given an input hashlist, returns an output hashlist containing the subset of *hashlist_in* that are stored in the key-value store

MetaStore service

The MetaStore server maintains the mapping of filenames to hashlists (and version numbers) called a FileInfoMap. It is also aware of the available BlockStores and is in charge of assigning blocks to BlockStores. When we start to test your project, we will always start from a "clean slate" in which there are no files in the system.

The service implements the following API:

GetFileInfoMap()	Returns a mapping of the files stored in the SurfStore cloud service, including the version, filename, and hashlist.
UpdateFile()	Updates the FileInfo values associated with a file stored in the cloud. This method replaces the hash list for the file with the provided hash list only if the new version number is exactly one greater than the current version number. Otherwise, an error is sent to the client telling them that the version they are trying to store is not right (likely too old) as well as the current value of the file's version on the server.
GetBlockStoreMap (*hashlist_in*)	Given an input hashlist, returns a mapping of BlockStore addresses to hashlists. Semantically, each entry in this map is a pairing of a BlockStores with the set of hashes from the input hashlist that it manages. Each hashlist in this mapping should be a subset of the input hashlist and the union of all hashlists should be equal to the input hashlist.
	For Project 3, you can assume that the MetaStore will be configured with **exactly 1 BlockStore**, so the implementation of this method can be trivial. In Project 4, you'll be extending your Surfstore implementation to support multiple BlockStores.

To create a file that has never existed, use the UpdateFile() API call with a version number set to 1. To create a file that was previously deleted, update the version number that is one larger than the "tombstone" record.

Algorithm

Basic Operating Theory

When a client syncs its local base directory with the cloud, a number of things must be done to properly complete the sync operation.

The client should first scan the base directory, and for each file, compute that file's hash list. The client should then consult the local index file and compare the results, to see whether (1) there are now new files in the base directory that aren't in the index file, or (2) files that are in the

index file, but have changed since the last time the client was executed (i.e., the hash list is different).

Next, the client should connect to the server and download an updated FileInfoMap. For the purposes of this discussion, let's call this the "remote index."

The client should now compare the local index (and any changes to local files not reflected in the local index) with the remote index. A few things might result.

First, it is possible that the remote index refers to a file not present in the local index or in the base directory. In this case, the client should download the blocks associated with that file, reconstitute that file in the base directory, and then add the updated FileInfo information to the local index.

Next, it is possible that there are new files in the local base directory that aren't in the local index or in the remote index. The client should upload the blocks corresponding to this file to the server, then update the server with the new FileInfo. If that update is successful, then the client should update its local index. Note it is possible that while this operation is in progress, some other client makes it to the server first, and creates the file first. In that case, the UpdateFile() operation will fail with a version error, and the client should handle this conflict as described in the next section.

Handling Conflicts

The above discussion assumes that a file existed in the server that wasn't locally present, or a new file was created locally that wasn't on the server. Both of these cases are pretty straightforward (simply upload or download the file as appropriate). But what happens when there is some kind of version mismatch, as described in the motivation at the top of this specification? We describe what to do in this subsection.

Imagine that for a file like cat.jpg, the local index shows that file at version 3, and we compare the hash list in the local index with the file contents, and confirm that there are no local modifications to the file. We then look at the remote index, and see that the version on the server is larger, for example 4. In this case, the client should download any needed blocks from the server to bring cat.jpg up to version 4, then reconstitute cat.jpg to become version 4 of that file, and finally the client should update its local index, bringing that entry to version 4. At this point, the changes from the cloud have been merged into the local file.

Consider the opposite case: the client sees that its local index references cat.jpg with version 3. The client compares the hash list in the local index to the file contents, and sees that there are uncommitted local changes (the hashes differ). The client compares the local index to the remote index, and sees that both indexes are at the same version (in this case, version 3). This means that we need to sync our local changes to the cloud. The client can now update the mapping on the server, and if that RPC call completes successfully, the client can update the

entry in the local index and is done (there is no need to modify the file's contents in the base directory in this case).

Finally, we must consider the case where there are local modifications to a file (so, for example, our local index shows the file at version 3, but the file's contents do not match the local index). Further, we see that the version in the remote index is larger than our local index. What should we do in this case? Well, we follow the rule that whoever syncs to the cloud first wins. Thus, we must go with the version of the file already synced to the server. So we download any required blocks and bring our local version of the file up to date with the cloud version.

Usage Details

Client

For this project, clients will sync the contents of a "base directory" by explicitly executing the provided `./run-client.sh` script:

```
./run-client.sh -d <meta_addr:port> <base_dir> <block_size>

 Usage of ./run-client.sh:

                  -d:  Output log statements

    <meta_addr:port>:  (required) IP address and port of the MetaStore the
                       client is syncing to

          <base_dir>:  (required) Base directory of the client

        <block_size>:  (required) Size of the blocks used to fragment files
```

The block size is specified as a command line option and you should use that instead of a hard-coded number. Note that while your system must support different-sized blocks (specified on the command line), **the size of the block will remain constant during any particular series of tests.** So we might run a set of tests with a block size of 4096, then clear everything and run a totally different set of tests in a new environment with a block size of 1 megabyte (for example).

Server

As mentioned earlier, Surfstore is composed of two services: MetaStore and BlockStore. In this project, the location of the MetaStore and BlockStore shouldn't matter. In other words, the

MetaStore and BlockStore could be serviced by a single server process, separate server processes on the same host, or separate server processes on different hosts. Regardless of where these services reside, the functionality should be the same.

Starting a Server

To start a server, we've provided a `./run-server.sh` script that runs the SurfstoreServerExec program which can be configured with command-line arguments:

```
./run-server.sh -s <service_type> -p <port> -l -d (blockstoreAddr*)


 Usage of ./run-server.sh:

   -s <service_type>:  (required) This defines the service provided by this
                       server. It can be "meta", "block", or "both" (you
                       don't need to include the quotation marks).

          -p <port>:  (default=8080) Port to accept connections

                -l:  Only listen on localhost if included

                -d:  Output log statements

  (blockStoreAddr*):  Space-separated BlockStore addresses (ip:port) the
                      MetaStore should be initialized with. (Note: if
                      service_type = both, then you should also include the
                      address of the server that you're starting)
```

Single Server Process

When iteratively developing your implementation of Surfstore, it'll be beneficial to have an easy way to start both a MetaStore and BlockStore. Using the run-server.sh script, this can be done with service_type = both. Internally, this should register both the MetaStore interface and BlockStore interface to the rpcServer.

```
/*
  The command below starts a server with service_type = both (MetaStore and
  BlockStore) that listens only to localhost at port 8071. Also, since
  `both` includes the MetaStore interface, you'll also have to include the
  ip:port of the server that you're starting with this command. In this
  case, this will be `localhost:8071`.
*/
```

```
./run-server.sh -s both -p 8071 -l localhost:8071
```

Separate Server Processes

For other scenarios where you might want to test having separate MetaStores and BlockStores, you can use the other two service types. Below, I've given examples for starting separate MetaStores and BlockStores locally, but you can just as easily do the same with different hosts e.g. maybe having the MetaStore on local and the BlockStore on an AWS instance.

```
/*
  The first command below creates a server process that registers
  only the BlockStore interface and listens only to localhost on port 8071.
  (& just means to run the process in the background i.e. the shell
  (parent) doesn't wait for the started server (child) to finish. However,
  you'll need to remember to kill the process when you're done. Another way
  to do this is just to run the commands below on separate terminals
  without the '&').

  Next, the second command starts a server process that registers only the
  MetaStore interface and listens only to localhost on port 8080 (default).
  Furthermore, it's configured to be initialized with the address of the
  BlockStore that we create above (localhost:8071)
*/

> ./run-server.sh -s block -p 8071 -l &
> ./run-server.sh -s meta -l localhost:8071
```

Grading Rubric

Sanity Check	- Calling sync with an empty base directory and empty server doesn't result in an error, etc. - New local files sync to the cloud - Files on the server that aren't on the client sync to the client - Mixture of new and missing files sync properly — the new files get uploaded to the cloud, and the missing files get downloaded properly
Updates	- If the cloud has changes to a local file, those changes get downloaded properly. If the local client has changes that aren't on the cloud, those changes get uploaded properly
Conflicts	- Your client and server handle the case where multiple clients have modified a file concurrently, and the above described rules are followed.

Pre-Submission Checklist

☐ Make sure that your program generates/reads/uses a local index.txt file. Make sure the format of that file matches the above spec because our testing program might need to read e.g. the version number of a particular file.

☐ Verify that your program works with binary files (images, video, etc).

☐ Make sure your program uses the block size specified in the command line argument. Don't hard-code a 4096 byte block size.

☐ When updating or creating a file, upload the necessary blocks to the block store first, before updating the fileinfo map on the server. This way, there isn't a race condition where a client downloads the fileinfo map, then tries to download the blocks but they haven't been uploaded yet.

☐ Make sure Surfstore works with all possible configurations of single MetaStore and single BlockStore. (Don't hard-code the BlockStore address in the MetaStore)

Starter code

● https://classroom.github.com/a/ZPtXhk2R

Submitting your work

Access GradeScope via Canvas so we can ensure that your grades sync properly.

```
Submission Files:

|— build.sh
|— run-client.sh
|— run-server.sh
|— src
    |—surfstore
        |— ...

Notes:
 - If submitting with a zip file, make sure the files listed above are at
   the top-most level i.e. the zip file doesn't contain a folder that
   contains the files above
 - You don't have to submit: `bin`, `pkg`, and any base directory created
   during local testing
```

Due date/time

The due date/time are listed on the course calendar/schedule.

FAQ / Questions / Updates

Behavior

Q: Can we assume that the files on the client wont change for the duration of sync?
A: If Client 1 is executing a sync operation, you can assume that none of the files on Client 1's computer are changing. However, it is possible that a different Client, say Client 2, is interacting with the server. So two clients could by sync'ing at the same time.

Q: As we execute a sync operation, if we notice that a file has changed and we update it, do we start over and re-check every file again? Or just process the files we haven't processed yet?
A: Each time your client runs, it should "sync" the local files and any remote files. But it should only process the files that haven't been processed yet. For example, if you have a base directory with files A, B, C, and D in it, and you read and process A, then read and process B, if you notice that B needs to be updated (and you update it), then go on to C and D. Don't go back and re-check A again.

Q: How should we represent an empty file?
A: In memory, use an empty hashlist. In index.txt, write like the following:

```
EmptyFile.txt,5,
```

Code

Q: Can we change the starter code?
A: Yes, so long as the various requirements in the specification are still met, but don't change the RPC interface itself or the data types passed through RPC

Q: Should we copy the RPC demo code covered in lecture and expect it to work for this project?
A: You should use the demo code to study and understand how RPC works in Go. But for this project, in order for the autograder to work properly, please don't use `fmt.Scanln()` and go routines for `http.Serve`. More specifically, for the server, instead of:

```
go http.Serve(l, nil)
fmt.Print("Press enter key to end server")
```

```
fmt.Scanln()
```
Please just do:
```
http.Serve(l, nil)
```

Q: Register() vs. RegisterName()
A: `Register()` registers the service with the name of the concrete type. `RegisterName()` is similar to `Register()` but allows you to specify a custom name for the service. You are free to name the services however you want, but you might need to change some provided code as we assume that the BlockStore and MetaStore are registered with `BlockStore` and `MetaStore` respectively.

Q: rpc can't find service …
A: For this error, check if you're correctly registering services to rpc. Specifically, you might have to use `Register(&<blockStore/metaStore>)` instead of `Register(<blockStore/metaStore>)`.

Q: GOPATH or go mod?
A: For this project, please follow the setup process described in README.md. Because the version of the Go textbook that we're using doesn't cover go modules, we've decided to use GOPATH for this project.

Q: How to compute the SHA-256 hash for some bytes?
A: First, you could use the [crypto/sha256](#) package to compute the SHA-256 hash in bytes. Then to express the hash bytes in string, the convention is to use the hexadecimal encoding, which is available from the [encoding/hex](#) package. An example is given below:

```
var blockBytes []byte
hashBytes := sha256.Sum256(blockBytes)
hashString := hex.EncodeToString(hashBytes[:])
```

To avoid interfering with the grading script, please avoid any of the following:
➔ `fmt.Scanln()` and go routines for `http.Serve`
➔ `os.Chdir()` in your `ClientSync()` implementation
➔ Excessive `fmt.Println()` (Since I/O is slow, this might cause the autograder to timeout even if your implementation is correct. To address this issue, please use `log` instead of `fmt` since we silence them during testing with the debug flag or comment out any unnecessary `fmt` statements.)

Submission / Testing

Q: Will you use our client with our server?
A: Yes, we will use your client with your server

Q: Will the server crash at any point during the tests?
A: No. Assuming that your implementation is bug-free, you can assume that the **server will not crash** at any point during the tests.

Q: What version of go is used in the grading script?
A: Originally, we mentioned in the README.md that the autograder uses 1.14.6. We've recently bumped the version to 1.17.3. Since go is backwards compatible, you should be able to use any methods and packages introduced before 1.17.3.

###