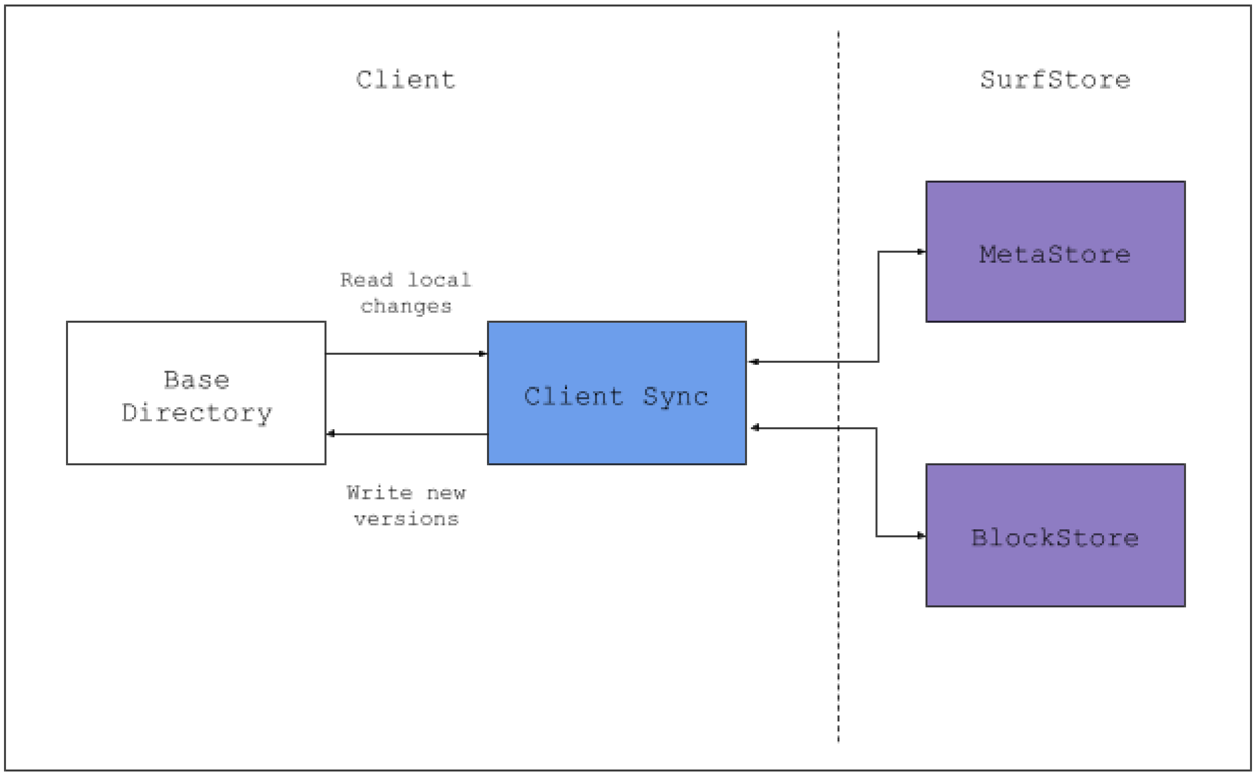
SurfStore

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 gRPC.

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:

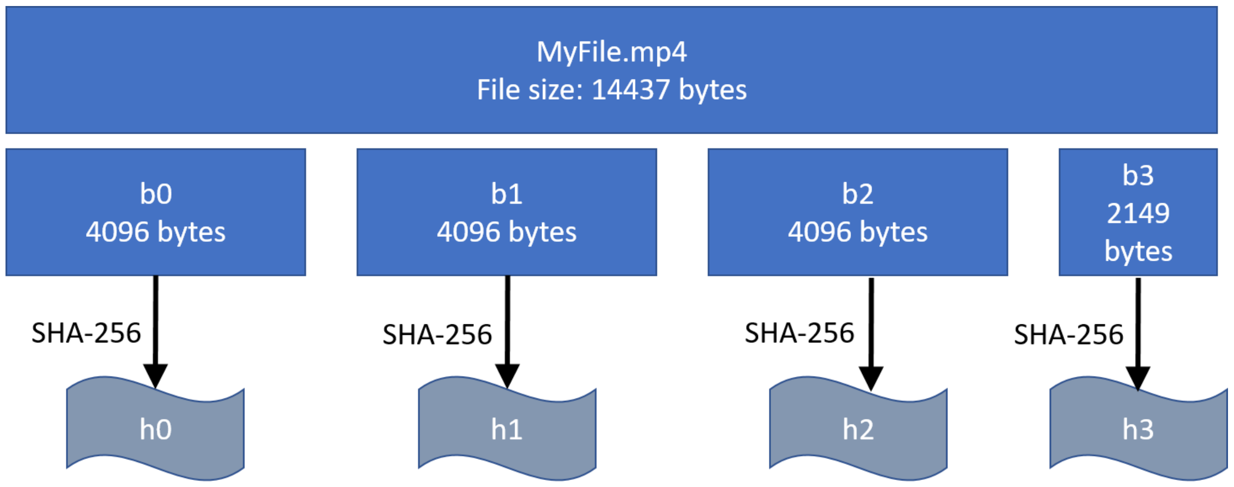
BlockStore	The content of each file in SurfStore is divided up into chunks, or blocks, each of which has a unique identifier. This service stores these blocks, and when given an identifier, retrieves and returns the appropriate block.
MetaStore	The MetaStore service manages the metadata of files and 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.db.

The Base Directory

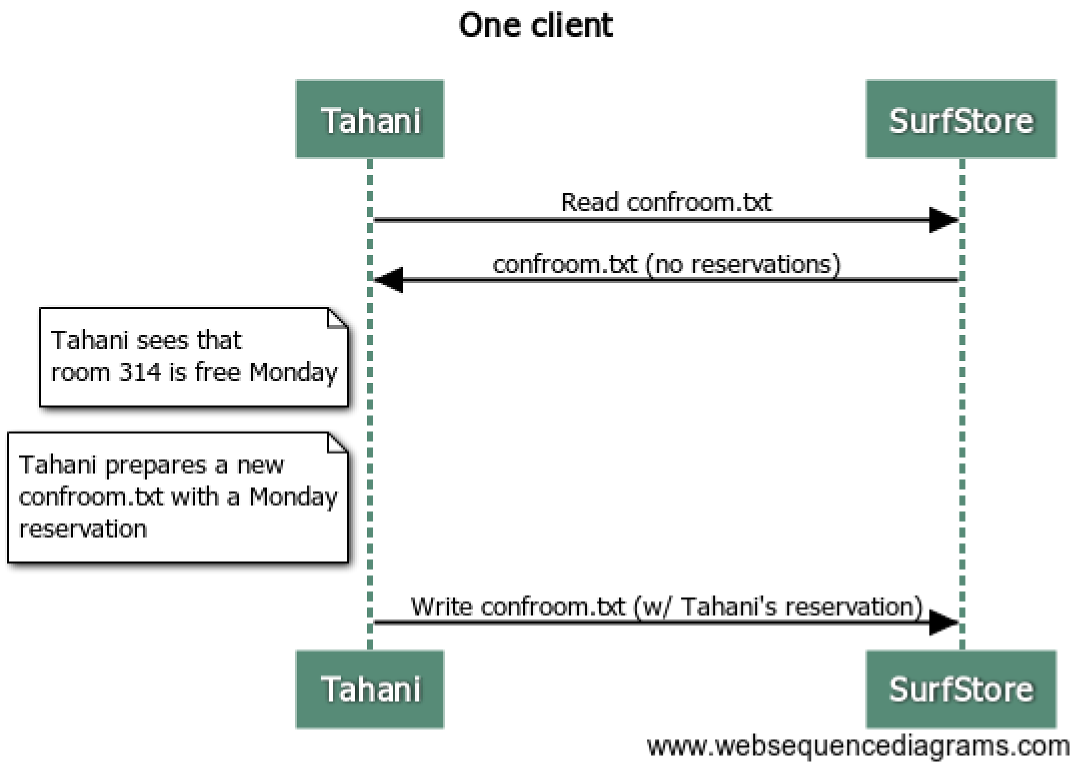
A command-line argument specifies a “base directory” for the client. This is the directory that is going to be synchronized with your cloud-based service. Your client will upload files from this base directory to the cloud, and download files (and changes to files) from the cloud into this base directory. 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

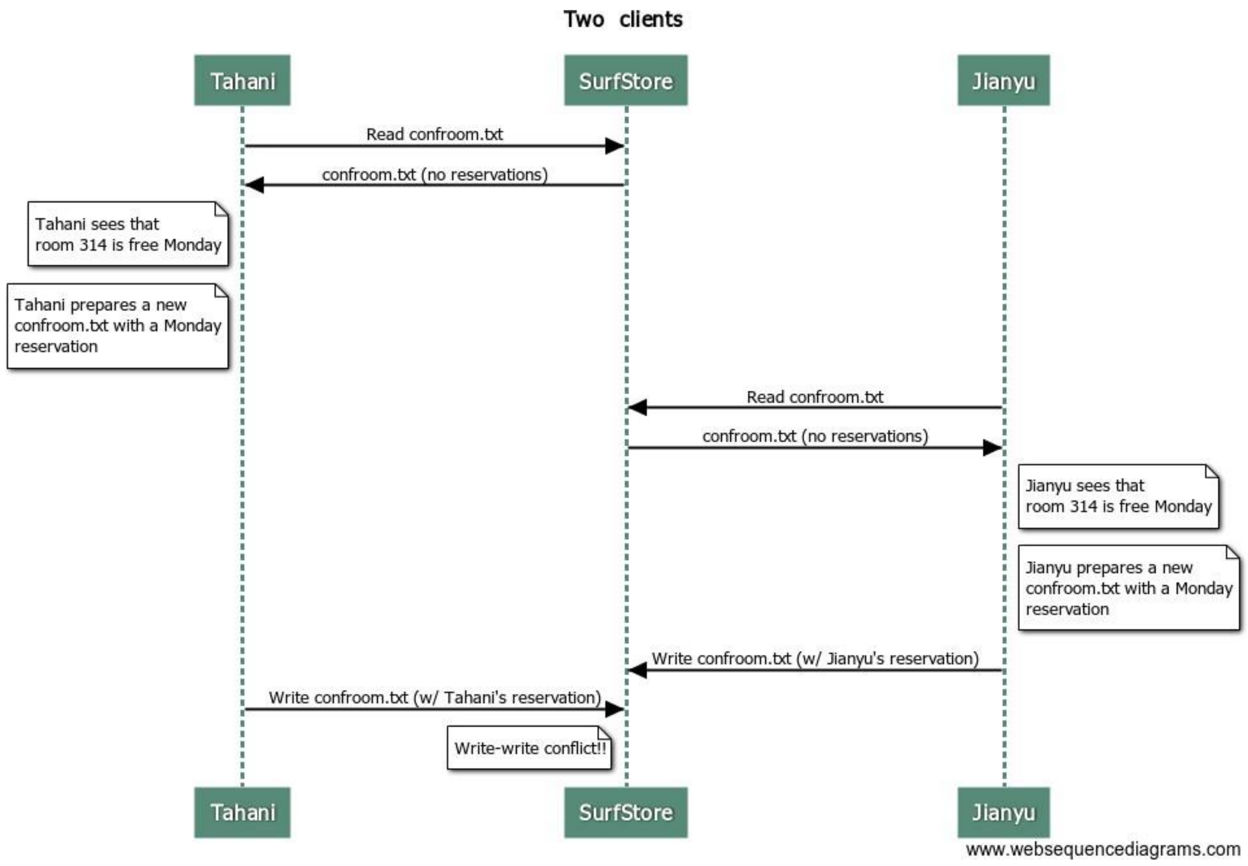
File Versions

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.

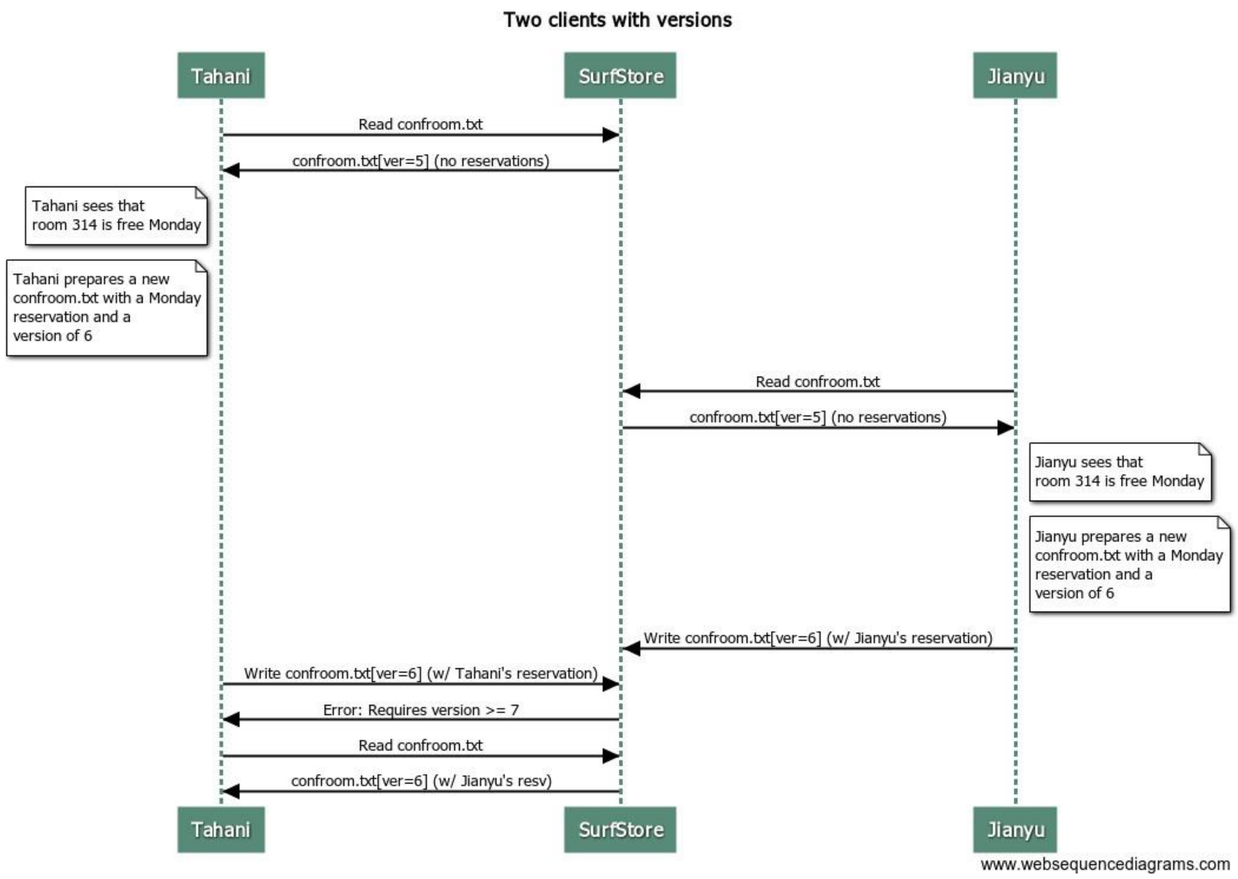
For example, imagine that Tahani wants to update a spreadsheet file that tracks conference room reservations. Ideally, they would perform the following actions:


One client
Tahani
SurfStore
Read confroom.txt
confroom.txt (no reservations)
Tahani sees that room 314 is free Monday
Tahani prepares a new confroom.txt with a Monday reservation
Write confroom.txt (w/ Tahani's reservation)
Tahani
SurfStore
www.websequencediagrams.com

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


Two clients
Tahani
SurfStore
Jianyu
Read confroom.txt
confroom.txt (no reservations)
Tahani sees that room 314 is free Monday
Tahani prepares a new confroom.txt with a Monday reservation
Read confroom.txt
confroom.txt (no reservations)
Jianyu sees that room 314 is free Monday
Jianyu prepares a new confroom.txt with a Monday reservation
Write confroom.txt (w/ Jianyu's reservation)
Write confroom.txt (w/ Tahani's reservation)
Write-write conflict!!
Tahani
SurfStore
Jianyu
www.websequencediagrams.com

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
Read confroom.txt
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
Read confroom.txt
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
Write confroom.txt[ver=6] (w/ Jianyu's reservation)
Write confroom.txt[ver=6] (w/ Tahani's reservation)
Error: Requires version >= 7
Read confroom.txt
confroom.txt[ver=6] (w/ Jianyu's resv)
Tahani
SurfStore
Jianyu
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:

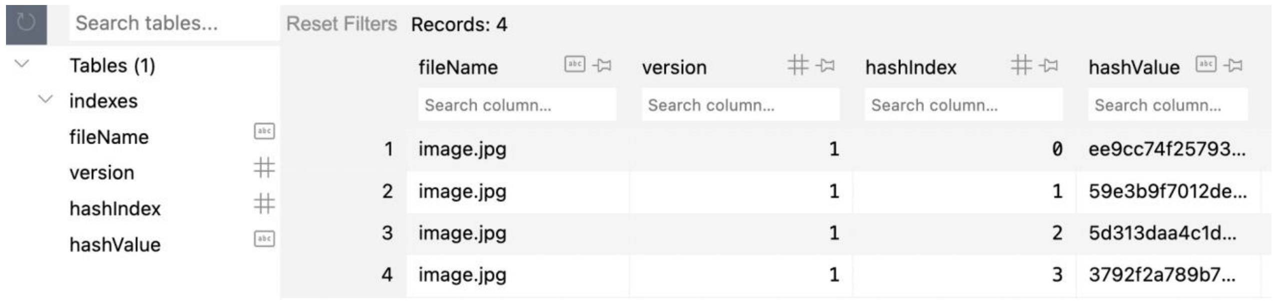
```
tahani $ ls /tdata

tahani $
(Tahani’s base directory starts empty)

tahani $ cp ~/kitten.jpg /tdata

tahani $ go run cmd/SurfstoreClientExec/main.go 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 $ go run cmd/SurfstoreClientExec/main.go 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.db
```

index.db

Your client program will create and maintain an index.db 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.db 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.

One .db file can contain multiple tables. For this project, you only need to maintain a table named as "indexes". The table has 4 columns which are fileName, version, hashIndex, hashValue. Their types are TEXT, INT, INT, and TEXT respectively. You can think of TEXT in SQL like a string in Golang. The following table is one example of a index.db that contains a table "indexes" with four records.



	fileName	version	hashIndex	hashValue
1	image.jpg	1	0	ee9cc74f25793...
2	image.jpg	1	1	59e3b9f7012de...
3	image.jpg	1	2	5d313daa4c1d...
4	image.jpg	1	3	3792f2a789b7...

There are records for image.jpg. All versions of the same file should be consistent. You can utilize the four hash values to get blocks from BlockStore. HashIndex is important in that a file is an ordered sequence of blocks and it's ordered by hashIndex. Make sure your hashIndex always starts from 0 and increment by 1 for the next hashValue.

The tool to view tables in .db file as shown above is sqlite browser. You can download it with link https://sqlitebrowser.org/dl/.

Note that the name "index.db" 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

a

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, you can send version=-1 to the client telling them that the version they are trying to store is not right (likely too old).
GetBlockStoreAddr()	Returns the BlockStore address.

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:

```
go run cmd/SurfstoreClientExec/main.go -d <meta_addr:port> <base_dir>
<block_size>


 Usage:

                   -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

Starting a server by:

```
go run cmd/SurfstoreServerExec/main.go -s <service_type> -p <port> -l -d
(blockstoreAddr*)


 Usage:

   -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*): BlockStore address (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. This can be done with service_type = both. Internally, this should register both the MetaStore interface and BlockStore interface to the grpcServer.

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 8081.
  (& 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:8081)
*/

> go run cmd/SurfstoreServerExec/main.go -s block -p 8081 -l &
> go run cmd/SurfstoreServerExec/main.go -s meta -l localhost:8081
```

Pre-Submission Checklist

- ☐ Make sure that your program generates/reads/uses a local index.db 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)
- ☐

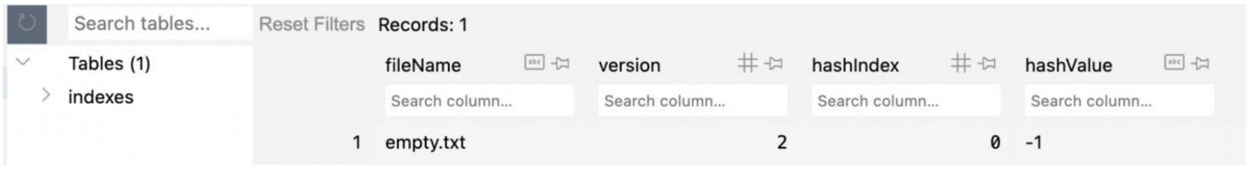
FAQ / Updates

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.

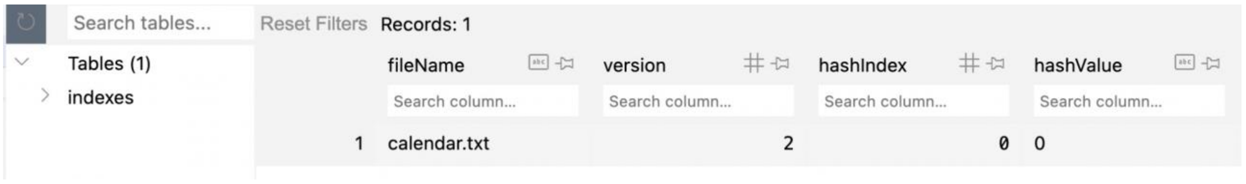
Q: As we execute a sync operation, if we notice that a file has changed and we update it, do we start over and re-check every file again? Or just process the files we haven't processed yet?
A: Each time your client runs, it should "sync" the local files and any remote files. But it should only process 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. You don't need to go back and re-check A again.

Q: How should we represent an empty file?
A: In index.db, an empty file has a row that a single hash value of "-1" (a string with two characters). It's like the following in the table.



	fileName	version	hashIndex	hashValue
1	empty.txt	2	0	-1

Q: How should we represent a deleted file?
A: In index.db,, a deleted file has a row that a single hash value of "0" (a string with one character). It's like the following in the table.



	fileName	version	hashIndex	hashValue
1	calendar.txt	2	0	0

Q: Can we assume there will be no hash collision for the blocks? A:
Yes, you can assume there will be no hash collision.

Code

Q: Can we change the starter code (e.g. protocol buffer, gRPC interface)? A:
Yes, so long as the various requirements in the specification are still met.

Q: Why does UpdateFile() return a version? Do we need to use it?
A: When the update fails because of verison, we can return version=-1 and err=nil to the client

telling them that the version they are trying to store is not right (likely too old) instead of returning error as mentioned in the previous answer. And only return error for network error related to gRPC call.

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 `grpcServer.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 outany unnecessary `fmt` statements.)

Submission / Testing

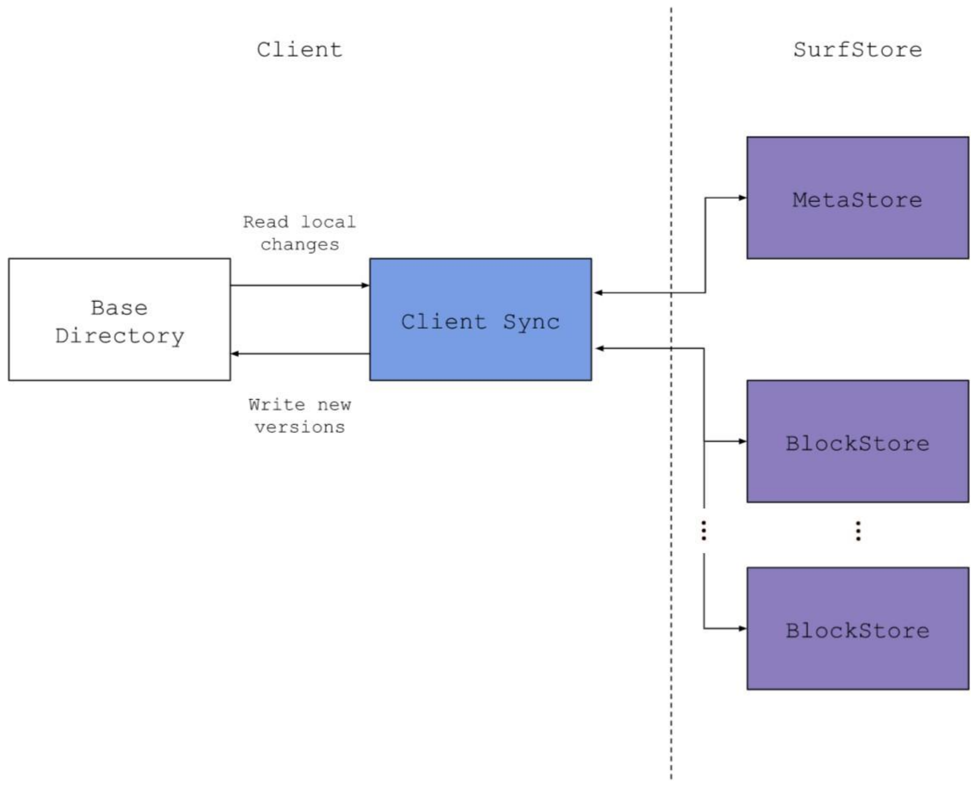
Q: Will you use our client with our server?
A: Yes, we will use your client with your server.

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.

Scalable SurfStore

Overview

In a real deployment, you would need thousands, perhaps 10s of thousands, of block store servers to handle the contents of a largeservice like Dropbox or Google Drive. In this project, we are going to simulate having many block store servers, and implement a mapping algorithm that maps blocks to servers.



Scaling your solution with consistent hashing

Consistent hashing

Consider a deployment of SurfStore with 1000 block store servers. As described above, to upload a file, you'll break it into blocks, and upload those blocks to the block store service (consisting of 1000 servers). Consider block B_0 with hash value H_0. On which of the 1000 block stores should B_0 be stored? You could store it on a random block store server, but then how would you find it? (You'd have to connect to all 1000 servers looking for it…). On the other hand, you could have a single "index server" that kept the mapping of block hash H_0 to which block store is used to store B_0. But this single index server becomes a bottleneck. As described in lecture, you could use a simple hash function to map the hash value H_0 to one of the block servers, but if you ever changed the size of the set of servers, then you'd have to reload all the data, which is quite inefficient.

Instead, we're going to implement a mapping approach based on consistent hashing. When the MetaStore server is started, your program will create a consistent hash ring in MetaStore. Since you're providing a command line argument including each block server's address, each block server will have a name in the format of "blockstore" + address (e.g. `blockstorelocalhost:8081`, `blockstorelocalhost:8082`, etc). You'll hash these strings representing the servers using the same hash function as the block hashes – SHA-256.

Each time when you update a file, your program will break the file into blocks and compute hash values for each block (you've already implemented this in P3). Then instead of calling `GetBlockStoreAddr`, this time we will call `GetBlockStoreMap` which returns a map indicating which servers the blocks belong to based on the consistent hashing algorithm covered in the lecture. Based on this map, you can upload your blocks to corresponding block servers.

The updates on grpc calls are described as follows.

MetaStore Service

The service implements the following API. Compared to the previous project, `GetBlockStoreAddr()` is replaced by `GetBlockStoreMap()` and `GetBlockStoreAddrs()`.

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, you can send version=-1 to the client telling them that the version they are trying

	to store is not right (likely too old).
GetBlockStoreMap()	Given a list of block hashes, find out which block server they belong to. Returns a mapping from block server address to block hashes.
GetBlockStoreAddrs()	Returns all the BlockStore addresses.

BlockStore Service

The service implements the following API. The function `GetBlockHashes()` is added in this project.

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
GetBlockHashes()	Returns a list containing all block hashes on this block server

Getting started

1. Generate new gRPC client and serverinterfaces from our .proto service definition.
2. Implement consistent hash ring in pkg/surfstore/ConsistentHashRing.go to calculate the mapping from blocks to servers. Modify necessary code to adapt this change in surfstore. Test sync file and consistent hashing.
3. Experiment and observe: Compare the mapping of blocks to servers with and without failed servers. For example, first run block servers on port 8081, 8082, 8083, 8084, and then run again on port 8081, 8083, 8084. Approximately what percent of the blocks are located on a different server? Does the output match your expectations based on our understanding of consistent hashing algorithms?

Usage Details

Client

Clients will sync the contents of a “base directory” by:

```
go run cmd/SurfstoreClientExec/main.go -d <meta_addr:port> <base_dir>
<block_size>


 Usage:

                  -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
```

Server

For a scalable surfstore, you may start one MetaStore server with multiple BlockStore servers. Block server addresses are defined in tail command-line arguments, separated by spaces.

Starting a server by:

```
go run cmd/SurfstoreServerExec/main.go -s <service_type> -p <port> -l -d
(blockstoreAddrs*)


 Usage:

   -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

  (blockStoreAddrs*):  BlockStore addresses (ip:port) the MetaStore should
                       be initialized with. Separated with spaces.
```

Example: Run surfstore on 2 block servers

```
> go run cmd/SurfstoreServerExec/main.go -s block -p 8081 -l
> go run cmd/SurfstoreServerExec/main.go -s block -p 8082 -l
> go run cmd/SurfstoreServerExec/main.go -s meta -l localhost:8081
localhost:8082
```

Testing

Before testing consistent hashing, you should make sure your surfstore still works as expected, that is, can successfully sync files. Our test will include all the cases for the previous surfstore project.

We will use SurfstorePrintBlockMapping to test which blocks each block server has after a sync operation. You can run it by:

```
> go run cmd/SurfstorePrintBlockMapping/main.go -d <meta_addr:port>
<base_dir> <block_size>
```

Please do not change this file!

Pre-Submission Checklist

- ☐ Make sure that your program supports all the features in project 3.
- ☐ Make sure Surfstore works with all possible configurations of single MetaStore and multiple BlockStore. (Don’t hard-code the BlockStore address in the MetaStore)
- ☐ Make sure you implement a consistent hash ring, and upload data to corresponding block servers.

FAQ / Updates

Behavior

Q: Can we assume there will be no hash collision for the blocks? A:
Yes, you can assume there will be no hash collision.

Code

Q: Can we change the starter code (e.g. protocol buffer, gRPC interface)?
A: Yes, so long as the various requirements in the specification are still met, and do not modify the functions we used for tests.

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 `grpcServer.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 outany 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: Will you add / delete some of the block servers during the tests? A:
No, we will not add or delete servers while testing.

Fault-tolerant SurfStore

FAQ/Updates

- What should happen if the client fails to sync because a majority of the cluster is crashed?

The client should throw an exception/exit with status 1

- What happens if the client cannot find a leader?

You can assume there will always be leader

Overview

You built a DropBox clone called “SurfStore”. Because data blocks are immutable and cannot be updated (since doing so would change their hash values, and thus they’d become entirely new blocks), replicating blocks is quite easy. On the other hand, replicating the MetaStore service is quite challenging, because multiple clients can update the Metadata of a file in a concurrent manner. To ensure that the Metadata store is fault tolerant and stays consistent regardless of failures, we can implement it as a replicated state machine design, which is the purpose of this project.

In this project, you are going to modify your metadata server to make it fault tolerant based on the RAFT protocol. To make the project reasonable to complete in 2 weeks you will only implement the log replication part of the protocol. The exact differences are explained in this document.

You **must** implement the interfaces in RaftInterfaces.go and your server **must** be defined as RaftSurfstoreServer. In this project, we're only testing the metadata functions (updatefile and getfileinfo), we will not test the putblock, getblock, etc. functions. These should still be functional however so that your client can successfully upload and download files. To help with testing we have released a few test cases, and you should implement your own as well.

Review

- The RAFT paper
  - Section 1, 2, 4, 5, 8, and 11 are required reading
  - Sections 3, 6, 7, 9, 10, and 12 are optional and not necessary for your project
  - You will **not** be implementing log compaction or membership changes in this project!
- The RAFT website
- Very helpful visualization of the protocol
- The RAFT simulator
  - If you have questions about what "should" happen during certain circumstances, this simulator is your best resource for investigating that situation.

Design

You will implement a RaftSurfstoreServer which functions as a fault tolerant MetaStore from project 4. Each RaftSurfstoreServer will communicate with other RaftSurfstoreServers via GRPC. Each server is aware of all other possible servers (from the configuration file), and new servers do not dynamically join the cluster (although existing servers can "crash" via the Crash api). Leaders will be set through the SetLeader API call, so there are no elections.

Using the protocol, if the leader can query a majority quorum of the nodes, it will reply back to the client with the correct answer. As long as a majority of the nodes are up and not in a crashed state, the clients should be able to interact with the system successfully. When a majority of nodes are in a crashed state, clients should block and not receive a response until a majority are restored. Any clients that interact with a non-leader should get an error message and retry to find the leader.

ChaosMonkey

To test your implementation you will need to use the RaftTestingInterface which defines 3 functions for ‘chaos’ testing and one function to access the internal state. This simulates the server crashing and failing. Note that we won’t really crash your program (e.g. by typing “Control-C” or sending it the kill command). When the Crash() call is given to a server, the server should enter a crashed state, and if it gets any AppendEntries or other calls, it should reply back with an error and not update its internal state. If a client tries to contact a crashed node, it should just return an error indicating it is crashed (letting the client search for the actual leader).

Our autograding code will call the Crash/Restore/GetInternalState methods as part of testing your codebase. To ensure your code works correctly, you should implement your own tests that invoke these methods to ensure that the properties of RAFT hold up in your solution.

API summary

As a quick summary of the calls you need in P5:

RPC call	Description	Who calls this?	Response during “crashed” state	GRPC Output
AppendEntries()	Replicates log entries; serves as a heartbeat mechanism	Your server code only	Should return ERR_SERVER_CRASHED error; procedure has no effect if server is crashed	AppendEntryOutput: should have the appropriate fields as described in the Raft paper
SetLeader()	Emulates elections, sets the node to be the leader	The autograder, your testing code	Should return ERR_SERVER_CRASHED error; procedure has no effect if server is crashed	Success: True if the server was successfully made the leader

SendHeartbeat()	Sends a round of AppendEntries to all other nodes. The leader will attempt to replicate logs to all other nodes when this is called. It can be called even when there are no entries to replicate. If a node is not in the leader stateit should do nothing.	The autograder, your testing code	Should return ERR_SERVER_ CRASHED error; procedure has no effect if server is crashed	We will not check the outputof SendHeartbeat. You can return True always or have some custom logic based on your implementation.
getblock(), putblock(), hasblocks()	Procedures related to the contents of files	Your client	N/A	Same as P4
GetBlockStoreA ddrs()	Returns the block store addresses	Your client	If the node is the leader, and if a majority of the nodes are working, should return the correct answer; if a majority of the nodes are crashed, should block until a majority recover. If not the leader, should indicate an error back to the client	Same as P4

GetBlockStoreMap()	Returns the block store map	Your client	If the node is the leader, and if a majority of the nodes are working, should return the correct answer; if a majority of the nodes are crashed, should block until a majority recover. If not the leader, should indicate an error back to the client	Same as P4
GetFileInfoMap()	Returns metadata from the filesystem	Your client	If the node is the leader, and if a majority of the nodes are working, should return the correct answer; if a majority of the nodes are crashed, should block until a majority recover. If not the leader, should indicate an error back to the client	Same as P4
UpdateFile()	Updates a file's	Your client	If the node is the	Same as P4
	metadata		leader, and if a majority of the nodes are working, should return the correct answer; if a majority of the nodes are crashed, should block until a majority recover. If not the leader, should indicate an error back to the client	

GetInternalState()	Returns the internal state of a Raft server	The autograder; Your testing code	Always return the state, don’t change this function	RaftInternalState The correct output is implemented in the starter code, do not change
Crash()	Cause the server to enter a “crashed” state	The autograder; Your testing code	Crashing a server that is already crashed has no effect	Success: Value does notmatter, if you want to changefor your testingyou can
Restore()	Causes the server to no longer be crashed	The autograder; Your testing code	Causes the server to recover and no longer be crashed	Success: Value does notmatter, if you want to changefor your testingyou can

Please consult Figure 2 in the Raft paper, and the SurfStore.proto and RaftInterfaces.go files for more information.

Changes to the command-line arguments of your server

Your server code needs to handle different command line arguments. For this project, your server should take in a path to a configuration file, the ID number of that server, and an address to a BlockStore server. For example:

```
$ go run cmd/SurfstoreRaftServerExec/main.go -f configfile.txt -i 0 -b
localhost:8080
```

Would start the server with a configuration file of configfile.txt. It would tell the server that it is server 0 in the configured servers. And the server that starts will assume that there is a BlockStore running on localhost:8080.

The client now also takes the same configuration file so that it knows where the servers are running. You can run the client with:

```
$ go run cmd/SurfstoreClientExec/main.go -f configfile.txt baseDir blockSize
```

A Makefile is provided to things easier, for example to run a BlockStore you should type:

```
$ make run-blockstore
```

Then in a separate terminal you can run a raft server with:

```
$ make IDX=0 run-raft
```

Configuration file

Your server will receive a configuration file as part of its initialization. The format is a json file, with the Metastore addresses and Blockstore addresses as lists. An example is given in the starter code (example_config.txt)

The metadata numbers start at zero. Note that all of the configuration files and command line arguments we provide to your system will be legal and we won't introduce syntax errors or other errors in your configuration files. You should be able to handle a variable number of servers, though you don't need to handle more than 10. Make sure to test with an even number of servers.

Leader election

Elections do not take place; the leader is set deterministically through a SetLeader API. The SetLeader function should emulate an election, so after calling it on a node it should set all the state as if that node had just won an election. Because elections do not take place, there are only two node states, Leader and Follower. The candidate state does not exist. You can guarantee that a SendHeartbeat call will be issued after every call to SetLeader. There will not be any tests where these functions are called in such a way that the nodes enter a state that is impossible in the Raft protocol. Leader conflicts are handled on Heartbeat or AppendEntries call. For example:
(Node A).SetLeader() (Node
A).SendHeartbeat()
…. Here node A is the leader in term 1(Node
B).SetLeader()
… here node A and B both think they are the leader, however node B is in term 2 and node A is term 1
(Node B).SendHeartbeat()

… Node A gets a heartbeat message from node B which has a higher term, node A steps down

Heartbeat

There is no heartbeat timer. Heartbeats are triggered through a SendHeartbeat API. There is no heartbeat countdown, so once the node is in the leader state, it stays in that state until there is another leader. A node could find out about another leader either through receiving a heartbeat from another leader with a higher term, or receiving a RPC response that has a higher term.

Clients

The client will keep a list of Raft servers, when making a request if the client gets an ERR_NOT_LEADER error, it can simply try the next server in the list. The server does not need to include the last known leader in its response. If the client is not able to successfully complete its GRPC request from any servers in the list it can return an error.

RPC limits

A single node should be able to handle up to O(10) RPC calls per second. Do not create an implementation with an excessive number of RPC calls (well above this limit), as that might cause our testing framework to malfunction.

Persistent storage

Because we’re only simulating crashes, you do not need to persist your replicated log on the filesystem.

Testing

Several example tests are provided in test/raft_test.go and test/basic_test.go. Some example config files are in test/config_files and example files are in test/test_files. You should also create your own files and configs to test your code. Any test code you write will be overwritten before running the autograder. The recommended way to test your code is through go test, command line testing is not recommended.

Debug Logging

Before submitting your code, try to minimize excessive logging. If there are a ton of log messages, it can interfere with the grader. Please make sure that all log statements end with a newline. Our autograder script separates results by newlines and having a logging statement can interfere with the output.

Design notes

Committing entries

When a client sends a command to the leader, the leader is going to log that command in its local log, then issue a two-phase commit operation to its followers. When a majority of those followers approve of the update, the leader can commit the transaction locally, apply the log to its state machine, and send the success response to the client. The commit behavior described in the Raft paper, section 5.3, says that the leader will keep trying indefinitely to replicate logs, in case some servers are crashed.

These “repeated tries” happen along with the Heartbeats/AppendEntries calls to other functional followers. Since SendHeartbeat(), which controls when AppendEntries() gets called, is called by the autograder, the expected behavior by your submitted code may NOT include trying indefinitely.

Versions and leaders

updatefile() should only be applied when the given version number is exactly one higher than the version stored in the leader. Every operation, including updatefile() and getfileinfomap() needs to involve talking to a majority of the nodes.

Note that the followers (and leaders) need to always implement GetInternalState, even when they are crashed.

SendHeartbeat

You are guaranteed to have SendHeartbeat called:

1. After every call to SetLeader the node that had SetLeader called will have SendHeartbeat called.
2. After every UpdateFile call the node that had UpdateFile called will have SendHeartbeat called.
3. After the test, the leader will have SendHeartbeat called one final time. Then all of the nodes should be ready for the internal state to be collected through GetInternalState.
4. After every SyncClient operation

Errors

There are two errors defined in RaftConstants.go, ERR_SERVER_CRASHED and ERR_NOT_LEADER. You should return those error variables when the server has crashed, or when the server is not the leader.