Project 2: SurfStore

Click HERE to see updates/FAQs related to this write-up.

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 file 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 sub-services:

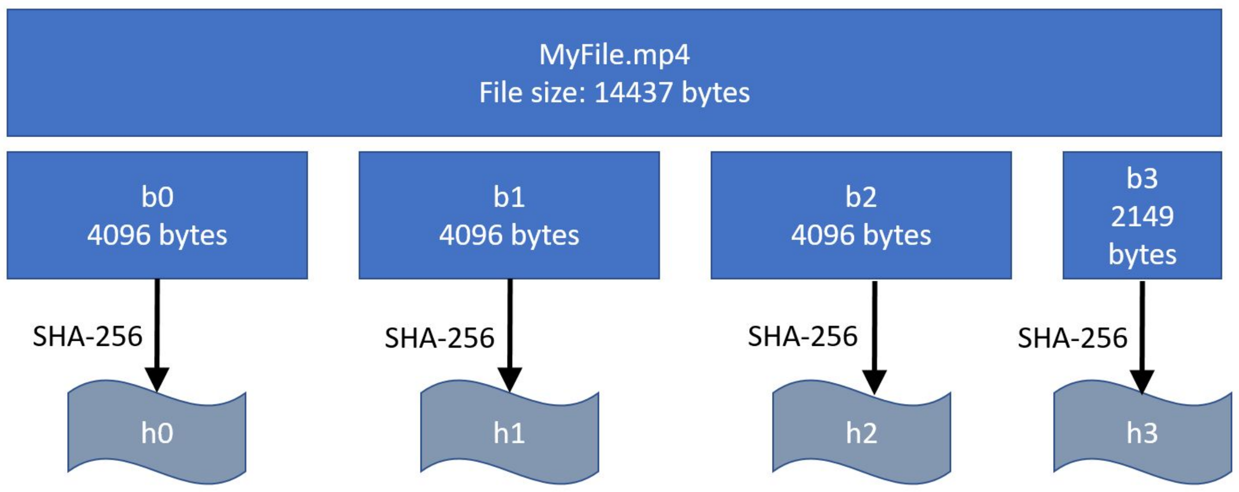
- **BlockStore**: The content of each file in SurfStore is divided up into chunks, or blocks, each of which has a unique identifier. Your service stores these blocks, and when given an identifier, retrieves and returns the appropriate block.
- **MetadataStore**: The MetadataStore service holds the mapping of filenames to blocks.

Specification

We now describe the service in more detail.

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 filenames

Files in SurfStore are denoted by filenames, which are represented as strings. For example “MyDog.jpg”, “WinterVacation.mp4”, and “Expenses.txt” are all example of filenames. SurfStore doesn’t have a concept of a directory or directory heirarchy–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 cannot contain the “/” character.

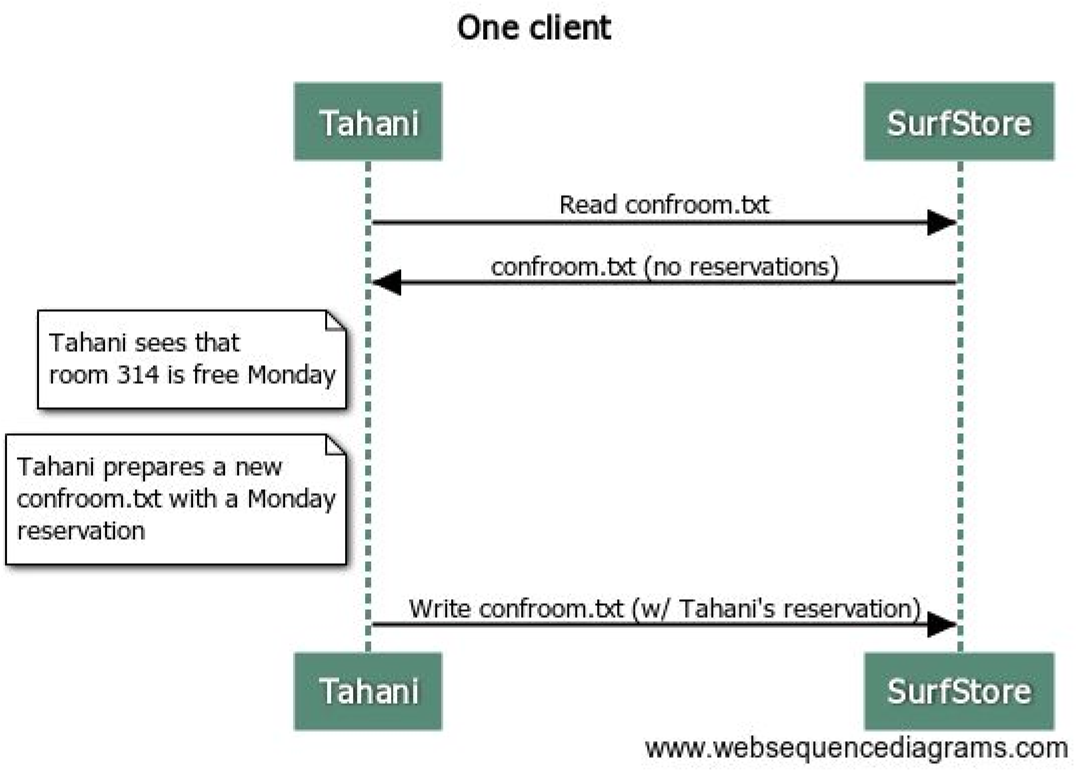
The base directory

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 directory specified by that command line argument.

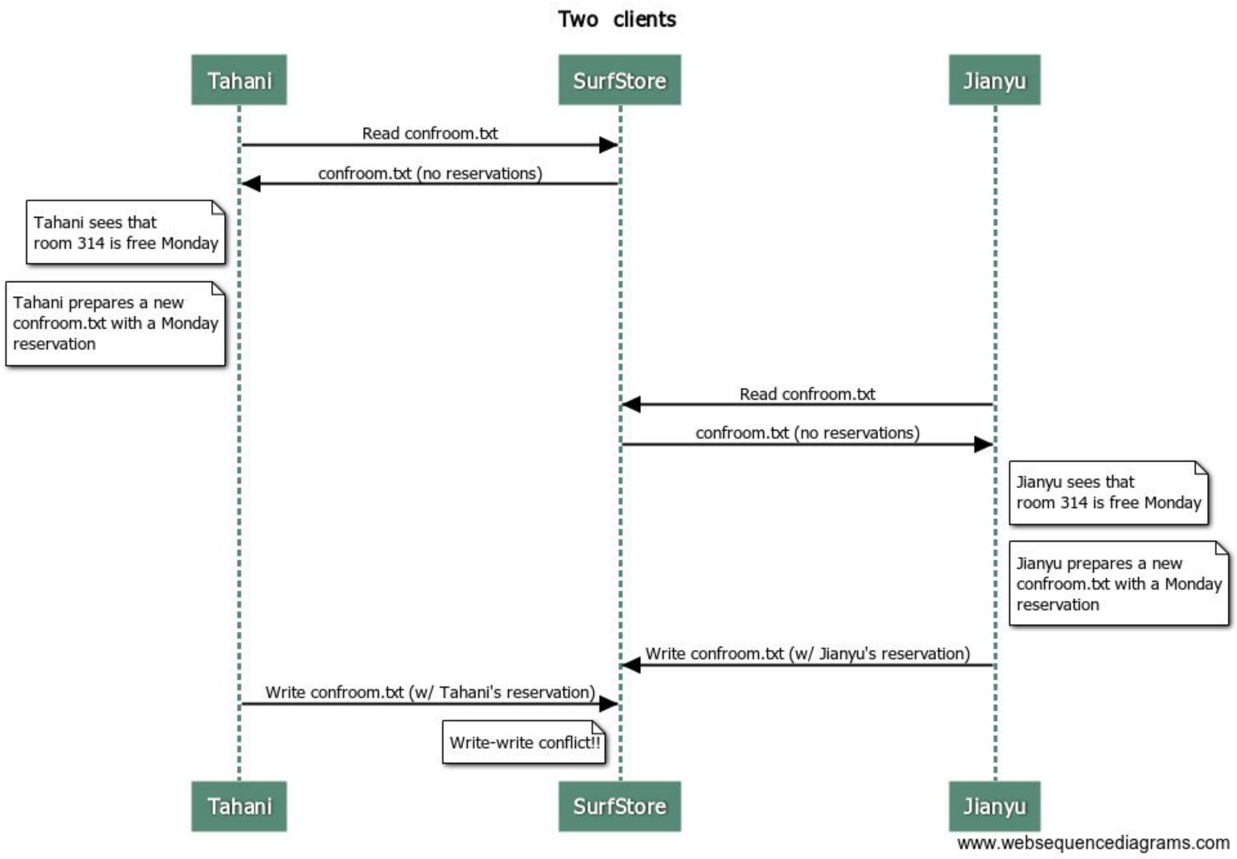
File versions

Each file/filename is associated with a *version*, which is a monotonically increasing non-negative 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.

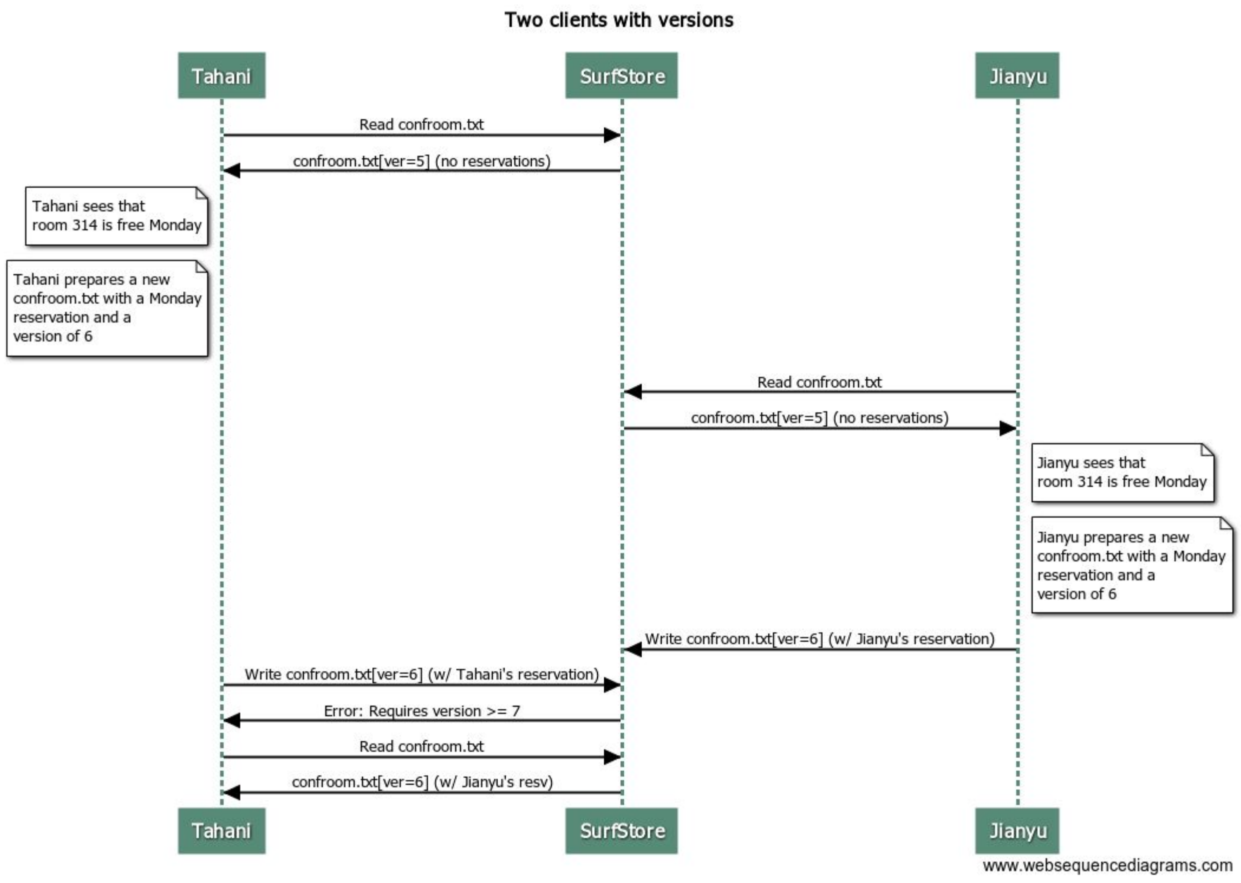
For example, imagine that Tahani wants to update a spreadsheet file that tracks conference room reservations. Ideally, they would perform the following actions:


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 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” (zero).

Client

Your client will “sync” a local base_dir 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. Resolving conflicts will be described below.

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
```

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. For example:

```
File1.dat 3 h0 h1 h2 h3
```

```
File2.jpg 8 h8 h9
```

Storing blocks

The SurfStore server keeps a map of data blocks in memory, stored in a key-value store. 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 a list of hashes "in", returns a list containing the subset of *in* that are stored in the key-value store

Metadata

The SurfStore server maintains the mapping of filenames to hashlists (and version numbers) called a FileInfoMap. All metadata is stored in memory, and no database systems or files will be used to maintain the data. When we 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 map of the files stored in the SurfStore cloud service. The keys in the map are filenames (as strings), and the values are FileInfo structures, which are tuples that have the file's version as the first element of the tuple, and a hash list as the second element of the tuple. Hash lists are represented as a list of strings.
- **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).

To create a file that has never existed, use the update_file() 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.

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 1 first scan the base directory, and for each file, compute that file's hash list. The client should then 2 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 3 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 3.1 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 3.2 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 update_file() 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, brining 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 syncd to the server. So we download any required blocks and bring our local version of the file up to date with the cloud version.

Implementation details

Configuration

For this project, you will use command line arguments to specify the server IP address, port number, the directory to use as a "base directory", etc. Further, the block size is specified as a command line option as well, and you should use that instead of a hard-coded number.

Grading rubric

- Sanity checking: 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

- Mixtures 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.
- Verify that your program works with binary files (images, video, etc). In particular, if you read binary data into a C++ std::string object, make sure to use the two-argument constructor (with the explicit length) so that the string class doesn't treat NULLs as terminators
- 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.

Starter code

- Starter code for Python: https://classroom.github.com/g/5W5SFys5
- Starter code for Java: https://classroom.github.com/g/313Mj12g
- Starter code for C++: https://classroom.github.com/g/9N1pq2hU

We have written up a little guide to working with XML-RPC, which is available here.

Submitting your work

Log into gradescope.com and associate your GitHub repository with your GradeScope account. Make sure you submit after you push your commits as the changes will not automatically reflect on gradescope.

Due date/time

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

FAQ / Questions / Updates

Update (11/7): The size of the block will not change in between calls to run-client.sh and run-server.sh. Once a particular block size is used, it will stay consistent from then on.

Update (11/6): You don’t need to support spaces in the filename.

Update (11/5): The MetadataStore APIs (modifyfile and getfileinfo) only deal with hashes--not the actual contents of the blocks. The Python starter code used the word blocklist but that should be hashlist. Only getbock and putblock should have the block contents (everything else is hash values).

Update (10/31): The file “index.html” is special, and will only be managed by your client. You can assume that an end user will never try to sync a file called index.txt to/from the cloud.

Update (10/30): We put together a guide for how to implement the SHA-256 hash for Java, Python, and C/C++. It is located at this link.

Update (10/29): Clarified that we’re using command-line arguments to configure the client and server, not a configuration file.

Where on the server are files kept?

- File blocks and the FileInfoMap are kept in the server’s memory. The files, on the server, are never “reconstituted” onto the server’s actual filesystem.

Does the next version of a file have to be ==(current version + 1) or just >=(current version + 1)?

- It has to be exactly current+1

If there is a version change for a file after the client gets an updated FileInfoMap, and the client determines it has to download the file, should it download the version from the FileInfoMap or the current version on the server?

- It should download the most recent version from the server.

Can we assume that the files on the client wont change for the duration of sync?

- 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.

What should we do for directories in base_dir?

- Subdirectories/directories aren’t part of the PA2 assignment, as mentioned in the spec.

Since we are given that “the last block in the file … may be smaller (but must be at least 1 byte large)” does this mean we wont have to deal with empty files?

- That is correct

Assuming only one client (and no other changes occur outside from the programs), by the end of sync() the FileInfoMap stored on the client and server should be the same, right?

- That’s correct.

Can we change the starter code?

- Yes, so long as it compiles in the same way

###