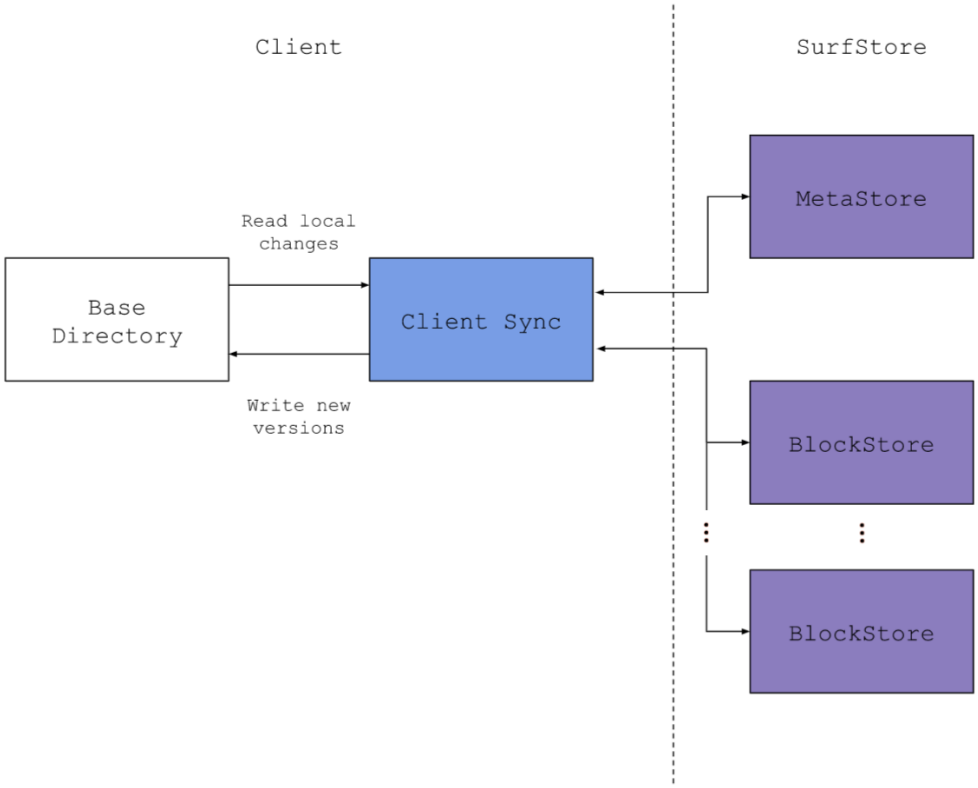
Project 4: Scalable SurfStore

Overview

The previous surfstore project worked on a small scale. In a real deployment, you would need thousands, perhaps 10s of thousands, of block store servers to handle the contents of a large service 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.



- Starter code: https://classroom.github.com/a/r10N4aCY

Follow the instructions in README.md to extend your project 4 code. We added 2 files and changed 5 files, you need to
Copy over:

cmd/SurfstorePrintBlockMapping/main.go
pkg/surfstore/ConsistentHashRing.go
pkg/surfstore/SurfStore.proto
pkg/surfstore/SurfstoreInterfaces.go

Manually change:

pkg/surfstore/SurfstoreRPCClient.go
pkg/surfstore/BlockStore.go
pkg/surfstore/MetaStore.go

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. Follow the README.md to extend your project 3. Generate new gRPC client and server interfaces 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

The client usage remains the same with project 3, 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.

Submitting your work

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

```
Submission Files:

|— cmd
```

```
|— pkg
    |— ...
|...

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
  - Please include generated .pb.go files in your submission
```

Due date/time

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

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

###