# Project 4: Scalable SurfStore

## Overview

In the previous project, we built the SurfStore service with a single MetaStore server and a single BlockStore server. As discussed in lectures, we might run into scaling issues with this architecture. In this project, we consider how to make this SurfStore service horizontally scalable. The main idea is to use consistent hashing to manage a dynamic cluster of BlockStore servers for load balancing. To scale out/in the service, we could dynamically add/remove nodes to/from the cluster. Consistent hashing ensures that these operations are efficient.

## Getting Started

Starter code: https://classroom.github.com/a/mWrj_2BU

- **Please update to the latest version if you cloned before Nov 26.**
- This time we provide a more complete starter code for you. Search "todo" to locate all the places you need to modify.
- After Project 3 is due, we have released another version of starter code with Project 3 reference solutions filled in. Don't worry if you have cloned the starter code earlier. You have access to the new starter code as well.
- We don't provide Project 3 client solutions in source code. Instead, we provide a user client binary for you to use this time. You'll need Docker to run this binary cross platform.

Due date: Dec 3, 5pm

## Algorithm

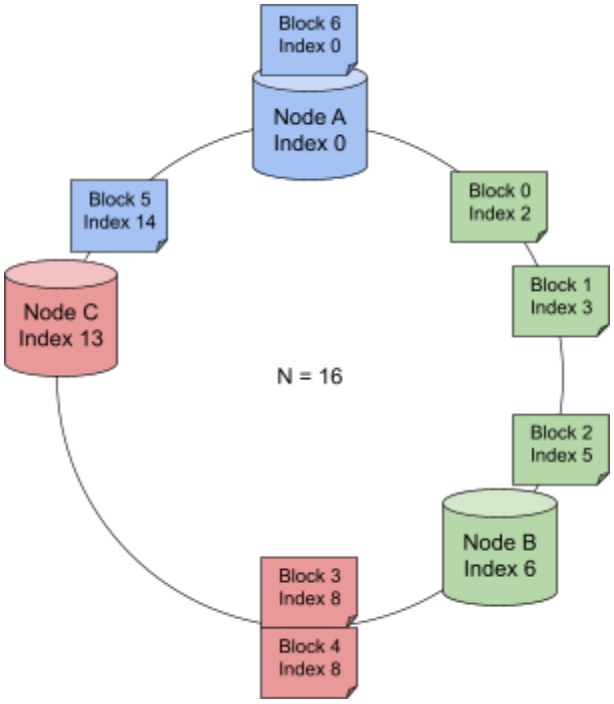
### Consistent Hashing

In this project, we use consistent hashing to maintain a dynamic cluster of BlockStore nodes. Consider a **ring** structure with indices ranging from 0 to N - 1, where N is the **ring size**. For example, a ring with a 7 bit keyspace supports indices ranging from 0 to 127. Both **block** and **node** are assigned to the ring with **index** computed in the following way:

```
blockIndex := hash(blockBytes) mod ringSize
nodeIndex := hash(nodeAddress) mod ringSize
```

For a given block with `blockIndex`, it should be stored in the nearest successor node on the ring (the node with the smallest `nodeIndex >= blockIndex` in the modulo sense).

### Example




This example shows a setup with:

- Ring size: 16
- 3 nodes: Node A - C
- 7 blocks: Block 0 - 6

Each node/block's index is also annotated. Different colors tell the association between blocks and nodes. In this example:

- Node A (index 0) is responsible for storing blocks with index in [14, 15, 0]
- Node B (index 6) is responsible for storing blocks with index in [1, 2, 3, 4, 5, 6]
- Node C (index 13) is responsible for storing blocks with index in [7, 8, 9, 10, 11, 12, 13]

## Node Addition

When we add a new node, only blocks in its successor node are influenced. Some blocks might need to be migrated from its successor node to the new node. To determine whether a block needs to migrate, we look at the index of the block and see if the new node would be responsible for storing it.

### Example

Assume we have the same blocks as the previous example, but we only have Node B and C. Then the current block distribution could be described as:

- Node B: [Block 5, Block 6, Block 0, Block 1, Block 2]
- Node C: [Block 3, Block 4]

Now if we add Node A, we need to migrate [Block5, Block 6] from its successor Node B to itself. And eventually we would expect the following block distribution:

- Node A: [Block 5, Block 6]
- Node B: [Block 0, Block 1, Block 2]
- Node C: [Block 3, Block 4]

## Node Removal

Node removal is the reverse operation of node addition. When we remove an existing node, we also need to migrate all blocks from this node to its successor node. We only allow node removal when there are more than one node. This guarantees that there is always at least one node existing.

### Example

Assume the same setup as the example in Consistent Hashing. The block distribution could be described as:

- Node A: [Block 5, Block 6]
- Node B: [Block 0, Block 1, Block 2]
- Node C: [Block 3, Block 4]

Now if we remove Node A, we need to migrate all Node A's blocks to its successor Node B. And eventually we would expect the following block distribution:

- Node B: [Block 5, Block 6, Block 0, Block 1, Block 2]
- Node C: [Block 3, Block 4]

# Interface

This section describes the interfaces you need to implement in this project in a reference style. More detailed instructions on how to approach the implementation goes into the next section. This interface is an extension of Project 3. We will focus on the updated and new interfaces in this section. For the unchanged interfaces, please refer to the Project 3 doc.

## Commands

### User Client

The command stays the same. Its usage is copied below:

```
./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
```

## Admin Client

Admin is a new client role we introduced in this project. An admin can manage the nodes in the cluster by adding/removing nodes dynamically using the following command:

```
./run-admin.sh -s <service_type> <MetaStoreAddr> <BlockStoreAddr>

Usage of ./run-admin.sh:

                -s:  (required) Admin Service: add or remove

    <MetaStoreAddr>:  (required) IP address and port of the MetaStore that
                      manages the cluster

   <BlockStoreAddr>:  (required) IP address and port of the MetaStore to
                      add/remove to/from the cluster
```

## Server

The command mostly stays the same. The only change is the added <ring_size> parameter:

```
./run-server.sh -s <service_type> -p <port> -l -d -r <ring_size>
(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

                -r:  (default = 128) Consistent hashing ring size

  (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)
```

## MetaStore Service

MetaStore is the actual manager of the cluster of BlockStore nodes.

The service implements the following API:

| | |
|---|---|
| **GetFileInfoMap**() | Stay the same. |
| **UpdateFile**() | Stay the same. |
| **GetBlockStoreMap**(*hashlist_in*) | Given an input hashlist, returns a mapping from BlockStore addresses to hashlists. It tells which BlockStore **should** be responsible for storing which blocks.<br><br>In the previous Project, we assumed only one BlockStore. In this project, we have to assume potentially multiple BlockStores. |
| **AddNode**(*nodeAddr*) | Add the specified BlockStore node to the cluster. |
| **RemoveNode**(*nodeAddr*) | Remove the specified BlockStore node from the cluster. |

### ConsistentHashRing

Inside MetaStore, we maintain a ConsistentHashRing struct to manage the BlockStore cluster. This struct has the following methods:

| | |
|---|---|
| **ComputeBlockIndex**(*blockHash*) | Compute a block's index on the ring from its hash value. |
| **ComputeNodeIndex**(*nodeAddr*) | Compute a node's index on the ring from its address string. |
| **FindHostingNode**(*ringIndex*) | Find the hosting node for the given ringIndex. It is the first node on the ring with node.Index >= ringIndex (in a modulo sense). In class we called this the *successor* node. |

| | |
|---|---|
| **AddNode**(*nodeAddr*) | Add the given nodeAddr to the ring. |
| **RemoveNode**(*nodeAddr*) | Remove the given nodeAddr from the ring. |

## BlockStore Service

The service implements the following API:

| | |
|---|---|
| **PutBlock**(*b*) | Stay the same. |
| *b* = **GetBlock**(*h*) | Stay the same. |
| *hashlist_out* = **HasBlocks**(*hashlist_in*) | Stay the same. |
| **MigrateBlocks**(lowerIndex, upperIndex, dstAddr) | Migrate the specified blocks from this node to the specified destination node. The blocks with ring indices in the range [lowerIndex, upperIndex] (inclusive) need to be migrated. All the indices shall be interpreted in a (mod ringSize) sense, so we could use [0, -1] to represent all blocks in a node for example.<br><br>Whether to remove these blocks locally in the BlockStore after migration doesn't influence the correctness of your solution. And we won't check this in our testing. So you can make your own decisions. |

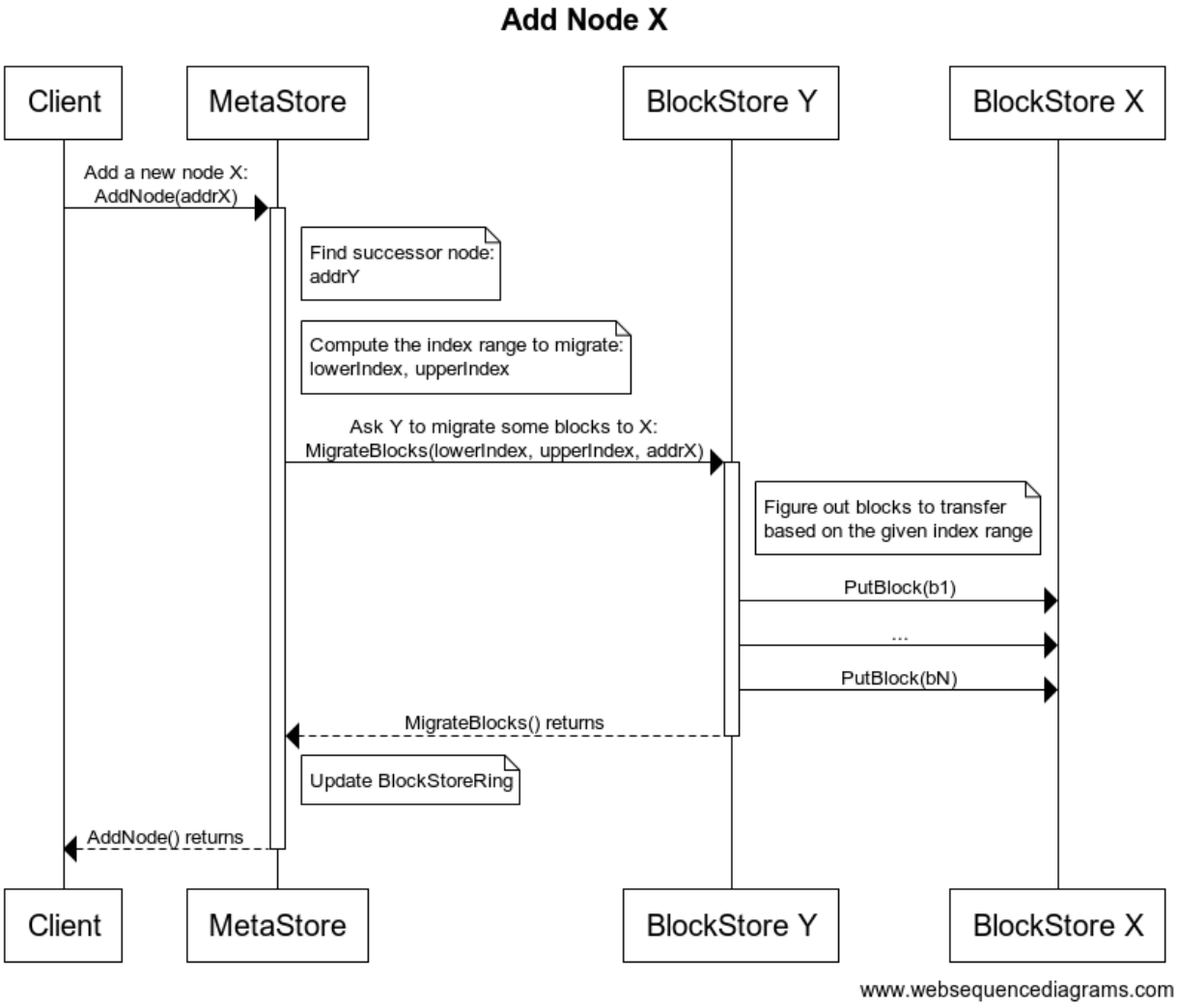
# Implementation Guide

This time we provide a more complete starter code for you. Search "todo" to locate all the places you need to modify. Here is a complete list of the methods you need to implement in this Project:

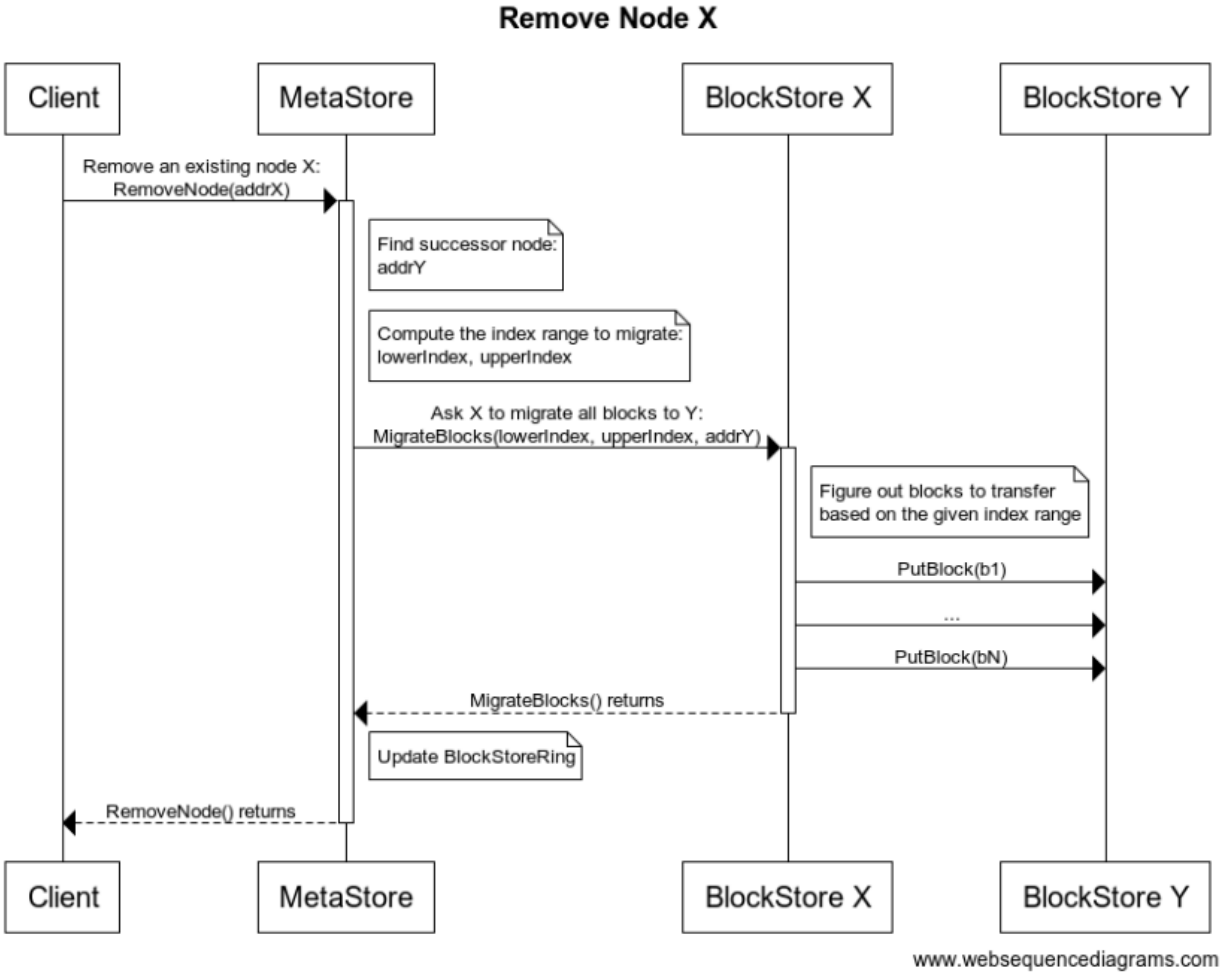
- MetaStore.go
  - GetBlockStoreMap()
  - AddNode()
  - RemoveNode()
- ConsistentHashRing.go
  - FindHostingNode()
  - AddNode()
  - RemoveNode()
  - NewConsistentHashRing()
- BlockStore.go
  - MigrateBlocks()

The ConsistentHashRing struct is used in MetaStore to represent the BlockStore cluster as BlockStoreRing. GetBlockStoreMap() could use this BlockStoreRing to tell the mapping from BlockStore servers to blocks. AddNode() and RemoveNode() are the main new methods you need to implement, they utilize BlockStoreRing and MigrateBlocks() under the hood. Their entire workflows are illustrated below.

## Node Addition



## Node Removal




# Testing Guide

All tests are visible to you on Gradescope this time, so testing shall be easier than previous projects. These tests could be categorized into 3 sets. We describe the details of each set in the following.

## Sync Tests

This set of tests are adapted from Project 3 directly. They have the same name as Project 3 tests, and they check the same things. The only difference is that these tests use multiple BlockStore servers instead of one. For these tests to work properly, you need to implement the **MetaStore.GetBlockStoreMap()** RPC.

# ConsistentHashRing Tests

This set of tests include:

- TestFindHostingNode4
- TestFindHostingNode8
- TestAddOneNode
- TestAddMultipleNodes
- TestRemoveOneNode
- TestRemoveMultipleNodes

These are unit tests of the ConsistentHashRing struct. More specifically, they test the following methods:

- **NewConsistentHashRing()**
- **FindHostingNode()**
- **AddNode()**
- **RemoveNode()**

TestFindHostingNode tests would create a new ConsistentHashRing with multiple BlockStore addresses, and test FindHostingNode() on all valid ring indices (from 0 to ringSize - 1) to see if the hosting node found is correct.

The last 4 tests work similarly. They would first create a new ConsistentHashRing with multiple BlockStore addresses, add/remove some nodes, and test FindHostingNode() on all valid ring indices (from 0 to ringSize - 1) to see if the hosting node found is correct.

For these tests to work properly, please make sure you don't change the signatures of these methods. And also don't change the Node struct definition, because it is used as the return type of FindHostingNode(). You are free to change other things in ConsistentHashRing.go.

# Scaling Tests

This set of tests include:

- TestBlockDistribution
- TestBlockDistributionAfterAddNode
- TestBlockDistributionAfterRemoveNode
- TestSyncTwoClientAfterAddNode
- TestSyncTwoClientAfterRemoveNode

As the name suggests, the first 3 tests check the block distribution. More specifically, we would call your MetaStore.GetBlockStoreMap() RPC to get a BlockStoreMap, and compare it with an expected BlockStoreMap. When we compare the two maps, the specific ordering of hashlists does not matter. We will first sort the hashlists and then compare them.

In the first test, we would sync a directory containing some files, and check the block distribution. For the second and third tests, we would sync a directory containing some files, add/remove a node, and check the block distribution.

For the last two tests, we would sync a directory containing some files, add/remove a node, sync another empty directory, and see if the files in the two directories are the same.

For these tests to work, you'll need to implement the following RPCs:

- **MetaStore.AddNode()**
- **MetaStore.RemoveNode()**
- **BlockStore.MigrateBlocks()**

## Suggestions

We suggestion approaching these tests in the following order:

1. ConsistentHashRing tests are the simplest, try to pass all of them first. You only need to have a correct ConsistentHashRing implementation to pass these tests.
2. Then try to pass all the sync tests. All that involves is a correct implementation of the MetaStore.GetBlockStoreMap() RPC. With the ConsistentHashRing struct, this implementation shall be pretty straightforward. (See FAQ for some clarification of what MetaStore.GetBlockStoreMap() does)
3. Finally try to pass all the scaling tests. These might require more careful debugging if they don't pass right away.

We have also provided a debug command (run-debug.sh) to help you examine blocks in each BlockStore. Its usage is explained in the starter code README. You are free to modify this tool for your needs.

# Submission Guide

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

```
Submission directory structure requirement:

|— src
    |— surfstore
        |— ...

Notes:
  - Yes, we only care about what's under the src/surfstore/ directory. All
    the other files are ignored in testing. It doesn't matter whether you
    include them in your submission or not.
  - If submitting with a zip archive, make sure it contains the
    src/surfstore/ directory at the top level (instead of nesting under
```

```
some-repo/src/surfstore).
```

# FAQ

## Behavior

**Q: Do we need to worry about hash collisions? For example, what if 2 BlockStore servers get assigned the same ring index?**
**A:** You don't need to worry about this. To make this project simpler, we make sure this would never happen in any Gradescope testing.

**Q: What happens when there is only one node in the ring and we try to remove it?**
**A:** You don't need to worry about or handle this case. Our tests on Gradescope won't try to do this.

**Q: What does MetaStore.GetBlockStoreMap() do? I'm still very confused.**
**A:** Here is an example. Assume `hashlist_in = [h1, h2, h3, h4, h5]`, and we could decide from the ConsistentHashRing that their corresponding hosting nodes are `[nodeA, nodeB, nodeB, nodeA, nodeA]`. Then the returned BlockStoreMap should be `{nodeA.Addr : [h1, h4, h5], nodeB.Addr : [h2, h3]}`.

## Code

**Q: What data structure should we use for the ConsistentHashRing?**
**A:** For best performance, it would require a data structure that supports O(log N) search/insert/delete operations, like a balanced binary search tree or a skip list. Golang doesn't have a standard library implementation of these and we don't require you to implement these. A reasonable objective to go for is a sorted slice that supports O(log N) search through binary search, and O(N) insert/delete operations. The sort package (especially sort.Search() and sort.Sort()) might be useful if you want to implement this. If you find this still too difficult to implement, it's fine to use an O(N) search solution. We don't test code performance, so don't worry about it.

**Q: I feel I need a method for ConsistentHashRing to find the previous node given an index, can I add it?**
**A:** Yes, you can. Just make sure all the existing methods' signatures are left unchanged. You can add the new method however you want.

**Q: Could I modify the Node struct in ConsistentHashRing.go?**
**A:** Actually, as long as your Node struct has an Addr field, our testing code shall work fine.

**Q: Why do the comments in the NewConsistentHashRing() function suggest using the HashMod() function?**
A: The comments provide one possible implementation. There are actually other implementations that can avoid calling the HashMod() function. So it's fine to not use the HashMod() function and implement NewConsistentHashRing() in the way that makes sense to you.

## Testing / Submission

**Q: How to install Docker on AWS EC2?**
**A:** https://docs.aws.amazon.com/AmazonECS/latest/developerguide/docker-basics.html

**Q: "The autograder failed to execute correctly. Please ensure that your submission is valid." What should I do?**
**A:** The first thing to check is if your code builds (./build.sh) locally, and if all the commands (run-server.sh, run-client.sh, run-admin.sh) work without errors locally. If the autograder still fails after all these checking, let us know by making a private post on piazza.

**Q: I'm passing all but the last 2 tests TestSyncTwoClientAfter(Add/Remove)Node. How could I debug?**
**A:** This means your block migration implementation has some issues, which involves `MetaStore.AddNode()`, `MetaStore.RemoveNode()`, and `BlockStore.MigrateBlocks()`.
First, try to reproduce the issue locally following the Testing Guide. Here are some suggestions:

- Set a reasonable block size (think 1024)
- Use a large file (with > 1000 blocks)

Once you could reproduce the issue, try to check the following things:

- Some blocks are actually migrated from one BlockStore to another after block addition/removal.
- The correct BlockStores are involved in the block migration.
- The migration direction is correct.
- All the blocks that should be migrated are actually migrated.