Project: AWS S3 Local Implementation

This is a group project.

In this project, you will develop a local implementation of Amazon Web Services (AWS) S3. This project is designed to develop a further understanding of fault tolerance and quorum-base protocol. This project also provides additional practice with the Go programming language.

There are a few steps to this project. The first step is to create a bucket in each node. Then, files must be written to a specified node and bucket. This file must also be read from, given the node index, bucket name, and file name. Finally, quorum-based protocol must be implemented for generic `ReadFile` and `WriteFile` methods. These methods must take into consideration if a node is faulty, as well as multiple versions of the same file across the nodes.

Overview

The **starter code** is structured as following:

- `src/replication`: this directory contains all source code for the AWS S3 local implementation
- `src/main`: this directory contains all text files used in testing
- `src/go.mod`: specifies the module name and go version
- `README.txt`

For this project, you will be asked to add code to places that say **TODO:** across multiple .go files. Before starting this implementation, we recommend reading the below sections, the textbook, and the lectures to ensure an understanding of AWS S3, fault tolerance, and quorum-based protocol.

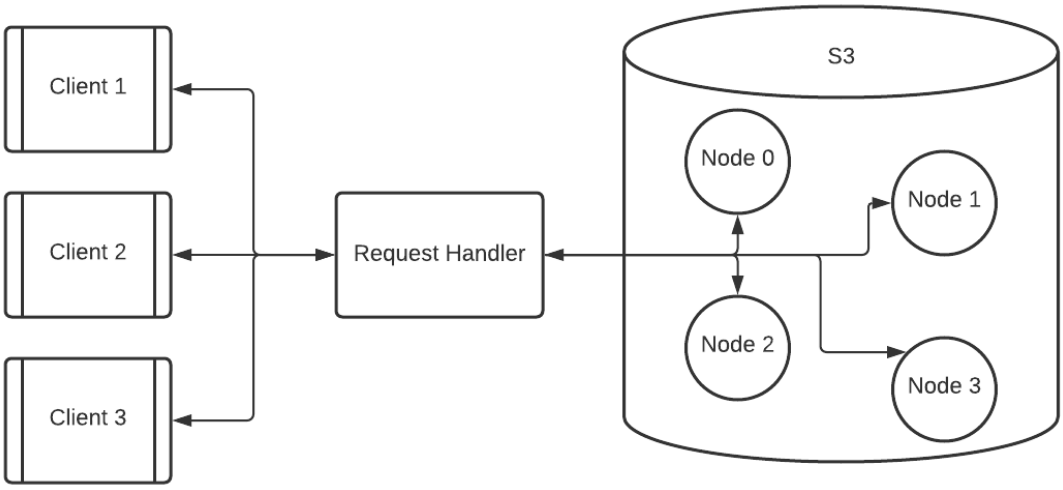
Glossary

It is important to understand the following terms before beginning this project.

- **Bucket**: a folder in which files are stored. A bucket is not a path to a physical location, but rather a location to a folder in the cloud.
- **Node**: a single computer that stores copies of buckets. In this project, a node is implemented as a folder inside the `src/replication/nodes/` directory.
- **Client**: a process sending requests and receiving responses from S3. In this project, clients are implemented as **goroutines** inside of the tests.
- **Request Handler**: The middleware between clients and the S3 service. This is the main portion of the project that you will be implementing. It receives requests, conducts operations on the nodes, and sends responses back.
- **Write quorum**: the number of nodes in which a file should be written to.
- **Read quorum**: the number of nodes in which a file should be read from.
- **Version**: the version of the file at time of writing. The version of a new file should be `time.Now()`.

AWS S3

Below is a diagram representing the local S3 implementation.



Clients

Clients are processes that send requests and receive responses from the request handler. In this project, clients are contained inside of `src/replication/test_test.go`. Each client will request either a write to a file, or a read to a file. A client does not care about quorum-based protocol; when a client requests to read a file, it should only receive the newest file and nothing more. Nothing needs to be implemented at the client end.

S3 and Nodes

There is a set of functions for interfacing with the local S3 service and its nodes. The following functions must be implemented in `src/replication/common_bucket.go` (labeled by a **TODO**).

- `CreateBucket`: This function should create a bucket. This entails creating a folder inside of each node. For example, if a bucket by the name of `test_bucket` were to be created in a fresh version of S3 with two nodes, then the S3 file system should look like this:
  - `src/replication/nodes/0/test_bucket/`
  - `src/replication/nodes/1/test_bucket/`
- `WriteNodeFile`: This function should write a specified byte array to the specified bucket directory in the specified node to a file with the specified file name. The specified file should be created if it does not exist, and completely overwritten if it does exist. This function should keep track of the version of the file that it writes. One solution is to create a "paired" version file (ex. `test.txt`: `test.txt.version`). For example, writing `"Test"` to a file named `text.txt` in a bucket named `test_bucket` inside `node 0`, then the S3 file system should look like this:
  - `src/replication/nodes/0/test_bucket/test.txt`

  This function should return the number of bytes written to the file.
- `ReadNodeFile`: This function should read the specified file from the specified node from the specified bucket. It should return the entire contents of the file in a byte array.

Request Handler

The request handler is the middleware between clients and the S3 service. This is the main portion of the project that you will be implementing. It receives requests, conducts operations on the nodes, and sends responses back.

The code for the request handler is contained within `src/replication/service.go`. There are two methods to implement within this file.

- `RequestWriteFile`: This function writes a specified file to a specified bucket. This function must consider that multiple clients may using S3 at the same time. If two clients want to write to the same file at the same time, then the client that requested to write first gets to write first. We recommend implementing some sort of scheduler. `RequestWriteFile` must also keep track of the version of the file it is writing. A new file's version must be `time.Now()`. You may keep track of a node's file's version how you like.
- `RequestReadFile`: This function gets the contents of a specified file from a specified bucket. `RequestReadFile` must retrieve the local file from each node (if it exists) and reach a quorum based on the files' versions before it returns the newest version of the file. Additionally, this function must consider multiple clients using S3 at the same time. If one client wants to write to a file while another client wants to read the same file at the same time, then the client that requested first gets to do its action first. We recommend implementing some sort of scheduler.

Phase 1: Writing and Reading with Individual Nodes

The first phase of this project is to implement the methods in `src/replication/common_bucket.go`. No quorum-based protocol code need be implemented to pass the tests for this phase. You only need to be able to create buckets, write files to specified nodes, and read files from specified nodes.

Testing

In Phase 1, you should attempt to pass `TestCreateBucket` as well as every `ReadWrite` test.

```
$ cd src
$  go test -v ./... -run Bucket
=== RUN   TestCreateBucket
--- PASS: TestCreateBucket (0.01s)
PASS
ok      repl/replication        0.030s
$
$  go test -v ./... -run ReadWrite
=== RUN   TestReadWriteOneClientOneNode
--- PASS: TestReadWriteOneClientOneNode (0.01s)
=== RUN   TestReadWriteTwoClientOneNode
--- PASS: TestReadWriteTwoClientOneNode (1.01s)
=== RUN   TestReadWriteOneClientTwoNodes
--- PASS: TestReadWriteOneClientTwoNodes (0.01s)
PASS
ok      repl/replication        1.072s
```

If these tests hang for longer than a few seconds, you likely have an issue with your implementation.

Phase 2: Quorum-Based Protocol and Fault Tolerance

The second phase of this project is to implement the methods in `src/replication/service.go`. In this phase, quorum-based protocol must be implemented so that the newest version of a specified file is the one that is return to a requesting client. Also, if a node is faulty, then the S3 system cannot fail.

Testing

By the end of phase 2, every test must pass.

```
$ cd src
$ go test -v ./...
=== RUN   TestCreateBucket
--- PASS: TestCreateBucket (0.01s)
=== RUN   TestReadWriteOneClientOneNode
--- PASS: TestReadWriteOneClientOneNode (0.01s)
=== RUN   TestReadWriteTwoClientOneNode
--- PASS: TestReadWriteTwoClientOneNode (1.01s)
=== RUN   TestReadWriteOneClientTwoNodes
--- PASS: TestReadWriteOneClientTwoNodes (0.01s)
=== RUN   TestQuorumOneClientFiveNodes
--- PASS: TestQuorumOneClientFiveNodes (0.02s)
=== RUN   TestFaultyNode
--- PASS: TestFaultyNode (0.02s)
```

```
=== RUN   TestQuorumThreeClientsFiveNodes
Info: if this test hangs, your code is likely broken. Beware any messages below.
--- PASS: TestQuorumThreeClientsFiveNodes (6.02s)
PASS
ok      repl/replication        7.125s
```

If these tests hang for longer than 15 seconds, you likely have an issue with your implementation.