Project 5: Fault-tolerant SurfStore

Due: Friday, June 7th at 11:59pm

TA in charge: Jamshed

Starter code

https://classroom.github.com/a/y97ilt5L

Overview

In Projects 3 and 4, you've been building 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 all clients need to see the same version of the file's metadata. 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. In particular you are going to adjust your metadata server to keep a log of operations, and only apply operations when they commit. We will be running multiple copies of your metadata server, and your job is to use RAFT to elect one of them as leader, accept requests from clients, and replicate the operations into the log of all other metadata servers using RAFT. 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 will re-use your P4 solution for the "starter code" for P5. Instructions on how to copy the files over from the P5 starter code are in the README. You **must** implement the interfaces in RaftInterfaces.go and your server **must** be defined as RaftSurfstoreServer. The autograder will be calling metadata functions (updatefile and getfileinfo) directly (i.e., the autograder tests will not invoke the putblock, getblock, etc. functions, but they still need to be functional 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 reachable 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 will not receive responses 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

In order to facilitate testing your implementation under a wide variety of instances, we will use the RaftTestingInterface which defines 3 functions for “chaos” testing and one function to access the internal state. The idea is that the autograder–or your own testing–will invoke these functions to simulate nodes crashing or becoming unreachable. To determine whether your code responds to such events properly, the testing harness will inspect each node’s internal state afterwards.

Said another way, the testing harness will not actually crash your program (e.g. by typing “Control-C” or sending it the kill command). Instead, it will invoke the Crash() function, which should cause your server to enter a “crashed” state. Similarly, the MakeServerUnreachableFrom(serverIds) function causes a server to pretend it is unreachable from the specified server IDs. If a server receives an AppendEntries(input) call while crashed or unreachable from the leader, it should respond with the ErrServerCrashedUnreachable error and refuse to update its internal state. Additionally, if a client attempts to contact a crashed node, the node should return the ErrServerCrashed error, prompting the client to search for the actual leader. The Restore() function restores the server state back to follower status and makes the server reachable from all other servers.

Our autograding code will call the Crash(), MakeServerUnreachableFrom(), Restore(), and GetInternalState() methods as part of testing your codebase. To ensure your code works correctly, you should implement your own tests in the Golang testing framework (see the Testing section below) 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 ***ErrServerCrashedUnreachable*** error. It should also return ***ErrServerCrashedUnreachable*** error if it is unreachable from the source server; procedure has no effect if server is unreachable|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 ***ErrServerCrashed*** 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 state it should do nothing.	The autograder, your testing code	Should return ***ErrServerCrashed*** error; procedure has no effect if server is crashed	We will NOT check the output of 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

GetBlockStoreAddrs()	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 metadata	Your client	If the node is the leader, and if a majority of the nodes are working, should return the	Same as P4

			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 not matter, if you want to change for your testing you can
MakeServerUnreachableFrom(servers)	Cause the server to be unreachable from the specified server IDs.	The autograder; Your testing code	Doesn't make a difference, since the crashed server is unreachable from every server.	***Success***: Value does not matter, if you want to change for your testing you can
Restore()	Causes the server to no longer be crashed	The autograder; Your testing code	Restores the server state back to follower status and makes the server reachable from all other servers.	***Success***: Value does not matter, if you want to change for your testing you 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 than in Project 4. For Project 5, 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 in Project 5; the leader is set programmatically 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() unless the server crashes before it can send heartbeats. 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 in Project 5. 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 ErrNotLeader 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.***

Design notes

Two-Phase Commit

When a client sends a command to the leader, the leader logs the command in its local log, then initiates a two-phase commit operation with its followers.

Log Replication

The first step is log replication among the **majority** of the followers. Here's ***one*** approach to handle this:

Define a custom enumerator, **PeerUpdateStatus**, with four states: **PeerUnknown, PeerUpdating, PeerUpdated, PeerUnreachable**. Initially, all followers are in the **PeerUnknown** state.

Initially all the followers are in the PeerUnknown state.

Within your functions (e.g., **UpdateFile**, **GetFileInfoMap**, etc.), once the leader starts sending heartbeats to the followers, the transition can proceed in two ways:

1) PeerUnknown -> PeerUpdating -> …… -> PeerUpdated

2) PeerUnknown -> PeerUnreachable

The function should only return when **none** of the followers are in the **PeerUnknown** or **PeerUpdating** states, and the **majority** of the servers are in the **PeerUpdated** state. If any servers remain in the **PeerUnknown, PeerUpdating,** or **PeerUnreachable** states, continue sending heartbeats.

Note that when you are sending a repeated heartbeat to a server that is in **PeerUnreachable** state, add a sleep mechanism—anything between **100-500 milliseconds** should suffice.

Commit

The commit phase is triggered via a call to **SendHeartbeat()** (The autograder will call **SendHeartbeat()** as needed).

The implementation of **SendHeartbeat()** can be either blocking or non-blocking when the majority of servers are unreachable, based on your design. The critical requirement is that any server that is reachable must be in the **PeerUpdated** state before the function returns.

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.

SetLeader()

According to the RAFT algorithm, the Leader Completeness Property ensures that a leader possesses all committed entries at the start of its term, but it may initially be unaware of which entries are committed. To resolve this, each new leader must commit a no-op entry in its log at the beginning of its term. This process not only confirms the leader's log up to that point but also assists in establishing authority among the followers.

For our project, the structure of the no-op entry is defined as follows:

```
entry := UpdateOperation{
    Term:         s.term,
    FileMetaData: nil,
}
```

You are encouraged to implement this by making a simple call to **UpdateFile()**, passing **nil** as the argument for file metadata. This ensures that the no-op entry is committed to the log, thereby also committing any preceding uncommitted entries. The specifics of how **UpdateFile()** processes this request are up to you, but it is essential that by the conclusion of this function call, the leader has successfully committed the no-op entry and ensured the integrity of prior log entries.

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

UNLESS the server crashes right after any of the scenarios described above.

Errors

There are three errors defined in RaftConstants.go, ErrServerCrashed, ErrServerCrashedUnreachable, ErrNotLeader. You should return those error variables appropriately. Note that the ErrServerCrashedUnreachable error should only be returned from the AppendEntries() function.

Starter code

Follow the instructions in the starter code to extend your Project 4 code. From our Project 4 solution, we changed RaftSurfstoreServer.go, RaftConstants.go, RaftUtils.go, and SurfstoreRPCClient.go.

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