Project 3: Fault-tolerant SurfStore metadata

FAQ/Updates

- You can use your PA2 code for this project. If you do **not** want to use that code, you can check out this starter code instead:
  - Python starter code
  - Java starter code
  - C++ starter code

Overview

In Project 2, you build 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 MetadataStore service is quite challenging, because multiple clients can update the Metadata of a file in a concurrent manner. 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 the first part of the assignment, you will implement leader election, and in the second part you will implement the replicated state machine aspect.

You are welcome (and encouraged) to re-use your P2 solution as the “starter code” for P3. However, if you had problems with your project 2 solution, you are welcome to check out the P2 starter code again and start over. In P3, we’re only testing the metadata functions (updatefile and getfileinfo), we will not test the putblock, getblock, etc. functions. Also, we’re only testing the implementation of your server--we won’t require that you invoke your client to update files on the disk, upload actual file contents, etc. Of course, you’re free to integrate your client with your new fault-tolerant server to build a fault-tolerant end-to-end system.

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 set of N processes (in C, C++, Java, or Python) that will communicate with each other via libxmlrpc to implement the RAFT protocol. The processes all know about each other (thanks to a provided configuration file), and the membership of this set does not change. Processes first work to elect a leader, and when a leader is elected, that leader process will begin accepting updates and queries to the filesystem metadata (i.e., via getfileinfomap() and updatefile()). 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 up 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 should block and not receive a response 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

To test your implementation, you will need to implement a "chaos monkey" script (separate from your testing client), which will call into servers and "crash" them by calling the crash method. This simulates the server crashing and failing. Note that we won't really crash your program (e.g. by typing "Control-C" or sending it the kill command). When the crash() call is given to your program, your program should enter a crashed state, and if it gets any appendentries or requestvote calls, it should just reply back that it is crashed and not update its internal state. If a client tries to contact a crashed node, it should just return an error indicating it is crashed (letting the client search for the actual leader).

Our autograding code will call the isleader/crash/restore/iscrashed methods as part of testing your codebase. Furthermore, we will call a `tester_getversion(string filename)` procedure as part of that testing.

To ensure your code works correctly, you should implement your own "chaos monkey" script that invokes these methods to ensure that the safety and liveness properties of RAFT hold up in your solution.

API summary

As a quick summary of the calls you need in P2, please consult Figure 2 in the paper and this summary:

RPC call	Description	Who calls this?	Response during "crashed" state
AppendEntries	Replicates log entries; serves as a heartbeat mechanism	Your server code only	Should return an "isCrashed" error; procedure has no effect if server is crashed
RequestVote	Used to implement leader election	Your server code only	Should return an "isCrashed" error; procedure has no effect if server is crashed
getblock(), putblock(), hasblocks()	Procedures related to the contents of files	Not used in PA3	N/A
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
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 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
tester_getversion()	Returns the version of the given file, even when the server is crashed	The autograder	Returns the version of the given file in the server's metadata map, even if the server is in a crashed state.
isleader()	True if the server thinks it is a leader	Your client; the autograder	Always returns the correct answer
crash()	Cause the server to enter a "crashed" state	The autograder; Your testing code	Crashing a server that is already crashed has no effect
restore()	Causes the server to no longer be crashed	The autograder; Your testing code	Causes the server to recover and no longer be crashed
iscrashed()	True if the server is in a crashed state	The autograder; Your testing code	Always returns the correct answer

Changes to the command-line arguments of your server

Your server code needs to handle different command line arguments than in project 2. For project 3, your server should take in the name of a configuration file, and then the ID number of that server. For example:

```
$ run-server.sh myconfig.txt 3
```

Would start the server with a configuration file of myconfig.txt. It would tell the newly started server that it is server #3 in the set of processes.

Configuration file

Your server will receive a configuration file as part of its initialization. The format is as follows:

```
M: 3
metadata0: <host>:<port>
```

```
metadata1: <host>:<port>
metadata2: <host>:<port>
```

As an example:

```
M: 5
metadata0: localhost:9001
metadata1: localhost:9002
metadata2: localhost:9003
metadata3: localhost:9004
metadata4: localhost:9005
```

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.

Make sure that your code works with the host being localhost, as well as a non-localhost port (e.g. a port in the AWS cloud network).

Leader election

You should start by reading the code to determine which functions are responsible for conducting Raft leader election, if you haven't already. For election to work, you will also need to implement the RequestVote() RPC handler so that servers will vote for one another.

To implement heartbeats, you will need to implement a portion of the AppendEntries RPC (though it will not convey any real payload yet), and have the leader send them out periodically. You will also have to write an AppendEntries RPC handler method that resets the election timeout so that other servers don't step forward as leaders when one has already been elected.

Make sure the timers in different Raft peers are not synchronized. In particular, make sure the election timeouts don't always fire at the same time, or else all peers will vote for themselves and no one will become leader.

Implementing leader election and heartbeats (empty AppendEntries calls) should be sufficient for a single leader to be elected and–in the absence of failures–stay the leader, as well as redetermine leadership after failures.

Timing

When a majority of the nodes are up and running and can communicate with each other, one of those nodes should be elected as leader within *5 seconds*. The paper proposes some timeout values. You should adjust your timeout values so that a leader gets elected within this 5 second time limit. We recommend starting with timeout values in the O(100) millisecond range. For example, heartbeat messages every e.g. 250ms, and an election timeout of a few hundred milliseconds. The 5 second rule for having a successful election is based on the low likelihood that several elections will fail back-to-back.

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 via `fsync()` or `fdatasync()` calls.

Debug Logging

Before submitting your code, try to minimize excessive logging. If there is a ton of log messages, it can interfere with the grader.

"Strawman" Testing strategy

In addition to your tests that you develop, your program should at a minimum act in the following ways. For this example, let's imagine that N=5.

- Initially, start with no instances of your program
- Start one instance. It should start up, become a candidate, but not become a leader
- Start a second instance, it should also not become leader
- Start the third instance. At this point, one of the three nodes should be elected leader (since now a majority of the 5 nodes are up, i.e., 3).
- Start the fourth and fifth nodes. These nodes should become followers, and the previously elected leader should remain the leader.

Once all N nodes are up:

- Kill the leader. One of the remaining (N-1) nodes should be elected to leader within 5 seconds
- Kill the next leader. Continue doing this, noting that new leaders will get elected within 5 seconds

- Once you kill enough leaders that a majority of the nodes are no longer up, then no nodes should become leader (they will remain candidates or followers, and keep failing elections).

Design notes

Committing entries

When a client sends a command to the leader, the leader is going to log that command in its local log, then issue a two-phase commit operation to its followers. When a majority of those followers approve of the update, the leader can commit the transaction locally, and send commit messages to the followers. When the leader successfully issues the commit to a majority of the followers, then it can reply back to the client. Now, what happens if a follower is in a crashed state? The leader should attempt to bring it up to date each heartbeat, meaning that every half second the leader should call into the follower with updated information.

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 (why is that?)

Note that the followers (and leaders) need always implement GetVersion, even when they are crashed. GetVersion should return the most up-to-date *committed* value of the filesystem. It should not include any logged (but not yet committed) updates.

Starter code

You are free to start with your PA2 code. “Fresh” PA2 code that includes the config file is provided up at the top in the FAQ.

Rubric

- 7.5/15 points: Leader election
- 7.5/15 points: Replicating state to the followers and recovering from leader failure