COMP 360 — Programming Task 3

Jeff Epstein

Due 17 Nov 2016

1 Introduction

We will continue building our distributed file system.

In this part of the project, we're going to extend our program to support redundancy, via a distributed hash table. Updates will be performed using a distributed commit algorithm.

Distributed hash table Please read section 5.2.3 in your textbook.

The distributed hash table will determine which servers will store particular keys.

Distributed commit Please read section 8.5.1 in your textbook.

To ensure that all replicas and correctly synchronized, each key value will be updated using a distributed commit algorithm.

2 Rules

Operating system I will evaluate your program submission by running it on a computer running the Debian Linux operating system. Your grade will therefore reflect the behavior of your work in a Linux environment.

While you are welcome to develop your projects under any operating system you like (such as Windows or OS X), I urge you to test it in an environment similar to mine, in order to assure yourself of getting the highest possible grade. Even if Linux is not your primary operating system, you can run it in a virtual machine under your primary operating system.

Network A distributed system, by definition, runs on multiple computers. Although we are building distributed systems, I assume that most students do not, in fact, have easy access to multiple computers. As with the previous assignment, you may develop your application using a virtual network or the loopback interface in lieu of a genuine network.

Language You should use whichever programming language you feel most comfortable with. However, if you want to use a language other than Python or Java, please talk with me first. If you choose to use Python, I recommend Python 3.

Academic honesty You should write this assignment entirely on your own, without consulting code from any other source, including from other students.

Code quality You should adhere to the conventions of quality code:

- Indentation and spacing should be both consistent and appropriate, in order to enhance readability.
- Names of variables and functions should be descriptive.
- All functions should have a comment in the appropriate style describing its purpose, behavior, inputs, and outputs. In addition, where appropriate, code should be commented to describe its purpose and method of operation.

If I think that the quality of your code (independent of the correctness of your program) is not of professional quality, I may ask you to make corrections and resubmit your work before I will give you a grade.

3 Assignment

The functional requirements are as follows. As always, you have a wide degree of latitude in how you implement the assignment. In particular, I leave to your judgment questions of which RPC functions to implement in order to achieve the required functionality.

This phase of the assignment is concerned with *replication*: stored key/values pairs will be replicated across multiple servers, to reduce chance of data loss in case of failure. We use two techniques to this end: a distributed hash table will tell us *where* (that is, on which servers) to store a particular key/value; and a distributed commit will ensure that values are updated on the servers in a consistent manner.

For the purposes of this assignment, you may assume a replication count of three. That is, every value is normally replicated across three servers, if enough servers are available.

3.1 Client command line

The client program, in addition to the command-line parameters required in the previous assignment, must support some new commands:

- The `setr` (replicated set) command should take two parameters: a key and a value. It work analogously to the `set` command. Unlike the `set` command, `setr` uses the distributed hash table to identify the servers where the value will be stored. These servers are termed the *replicas*. The value will be set on a number of servers equal to our *replication count*. The command will perform a distributed commit to ensure that the replicas are updated together.

  If there are no servers available, then the value will not be set, and the command should emit an error. If there are three or fewer servers available, then the command will attempt a distributed commit on all of them. If there are more than three servers available, then the command will attempt a distributed commit on three of them. If the distributed commit fails on any of the replicas, then it must not be performed on any of them. That is, the distributed commit must have all-or-nothing semantics.

  If the commit is performed successfully, the command should emit a confirmation message. If not, it should emit an error message.

- The `getr` (replicated get) command should take one parameter, a key. It work analogously to the `get` command. Unlike the `get` command, `getr` uses the distributed hash table to identify the servers where the value should be read from. Then the command will try to read from each replica, sequentially, until it finds one that is working and is storing the desired key.

  If the key's value was looked up successfully, the command should emit the value of the requested key. If not, it should emit an error message.

3.2 Distributed hash table

The `getr` and `setr` commands will store key/value pairs on multiple servers. A given key may be stored on up to *replication count* number of servers; each such server is called a *replica*. The set of keys stored on a given server is a *bucket*. Thus, each server has a bucket. The union of all buckets gives us the totality of stored data in our system.

We need a mechanism to determine which buckets should store each key. Assuming there are enough servers available, each key will be replicated in three different buckets. The bucket allocation mechanism must be stable; that is, if an invocation of `setr` puts a given key in buckets X, Y, and Z, an invocation of `getr` querying the same key, given the same view, must look for it in the same buckets. The function determining which bucket stores a given key is called the *bucket allocator*. The inputs to the bucket allocator are the key in question, and the current view (that is, the set of currently active servers); the bucket allocator will give which buckets the given key should be allocated into.

$$bucket\_allocator(key, view) \rightarrow [bucket]$$

Note that the bucket allocator is idempotent, and other than its parameters, it may not depend on mutable state. Your bucket allocator must use a circular distributed hash table, similar to the one depicted on page 64 of your book. It must base the output buckets on the hash of the given key and the hash of the identifiers of the available servers.

Note that the bucket allocator depends on the current view. After a view change (when a server is added or removed), the buckets for some keys may change. In that case, when a view change is detected, keys should be migrated to their new correct location. This process is called *rebalancing*. The bucket allocator should be designed in such a way as to minimize the number of keys that need to be rebalanced after a view change. During rebalancing, it's not safe or necessarily meaningful to perform `setr` and `getr` operations, and therefore the client program should refuse to execute these commands until rebalancing is finished.

The view leader should, as in the previous assignment, store the current epoch; the lock server; and the state of server heartbeats. However, the view leader must not store any key-specific information, as that would defeat the purpose of our distributed hash table. In particular, the view leader may not store a mapping of keys to buckets. Key-to-bucket allocation must be determined solely by your idempotent bucket allocator.

In designing your bucket allocator, a reasonable choice would be to store the value of key k at the server having identifier s such that $hash(s) >= hash(k)$, for the smallest value $hash(s)$, unless there is no such s, in which case the hash table "wraps around" and the smallest $hash(s)$ will be used. The server for secondary and tertiary replicas could be those having consecutively-numbered identifier hashes.

Your system should preserve data (including replication, wherever possible) even in the presence of (simulated) hardware failure. For the purposes of this assignment, you must assume that any server can fail; you may assume that the view leader never fails. Therefore, if we have several servers, with various keys distributed on them, we should be able to kill off the servers, one by one. After each server dies, copies of the keys that it stored should be rebalanced onto the remaining servers. As long as we allow enough time between server deaths for rebalancing to complete, no data should be lost. In the end, we can have one one server that stores all the keys that were previously stored with redundancy on several. At that point, we can gradually add more servers and the keys will be automatically rebalanced into the servers of the expanded cluster.

In your README file, you must document certain design choices in your implementation:

- Your bucket allocator algorithm. Show that it can (roughly) evenly distribute keys among available buckets. Describe precisely under which circumstances your algorithm can lead to the necessity to move keys during rebalancing.

- Your rebalancing implementation. Describe how your algorithm minimizes unnecessary copying; how it reduces load on the view leader; and how it coordinates view changes and rebalancing between the view leader, servers, and clients. Describe how a server determines which keys to copy during rebalancing.

3.3 Distributed commit

When the `setr` command writes values to multiple buckets, we need all-or-nothing semantics. If one of the servers has crashed, we must ensure that the value is not written to any of them. Therefore, we will use the two-phase commit protocol described in pseudocode in your book on pages 393-394. In this case, your client program will act as the "coordinator" by requesting and collecting votes from the replicas, who will act as "participants." The client will send a final commit messages when all votes have been collected, and an abort message if any participant is unable to commit (indicated by either explicitly voting against the commit or by timing out). A participant should vote against a commit if there is currently a pending commit on the same key, but multiple outstanding commits on different keys should be possible.

Your distributed commit implementation should be implemented in terms of direct communication between the client and the server. The view leader should be involved only to the extent that it provides the view to the client via the `query_servers` RPC. In particular, your distributed commit implementation should not depend on distributed locks. Your distributed commit implementation should fail cleanly if any of the constituent servers dies, whereupon the client should timeout and send abort messages to the remaining servers.

You are *not* required to implement participant-to-participant communication for recovery after a failed coordinator. That is, you may assume that client will not fail after beginning the voting phase. However, you may implement this functionality for extra credit, in which case you must explain your approach in your README file.

A consistent distributed commit requires that you consider various edge cases. For example, a view change might happen after the `setr` starts, resulting in some servers being in different epochs. Or, a server might fail and restart, without the view leader noticing, resulting in a server having a different identifier but the same TCP endpoint. To ensure consistency, a distributed commit should fail if any participant does not have the epoch and identifier expected by the coordinator.

In your README file, describe how you implemented the distributed commit, and which, if any RPC functions you used or extended.

3.4 Test cases

Some situations that you must test for:

- Verify that key lookups work. `setr` operations should succeed as long as there is at least one server available. `getr` should be able to retrieve values set by `setr`. Both `setr` and `getr` are allowed to fail in a window of time after a view change, during which (a) servers have inconsistent epoch or (b) rebalancing is underway.

- Verify that adding or removing a server triggers a view change. The most recent epoch is always stored in the view leader. Each server learns the current epoch in the acknowledgment of its heartbeat.

- Verify that key lookups work after a replica fails. Since each key should be stored on three replicas, a `getr` operation should still work, even if the primary replica has failed.

- Verify that key lookups work across view changes and rebalancing. That is, if a value is set by `setr`, and then the view changes (either by adding or removing a server), `getr` should still be able to find the value.

- Verify that values are migrated to the appropriate server in case of view change. The server where a given value is expected to be found may change depending on the identifiers of the servers currently in view. When a view change happens, the system must re-distribute values to the expected server.

  For example, assume we are using the $hash(s) >= hash(k)$ bucket allocator described above, with secondary and tertiary buckets on consecutively-numbered servers. Let's say we have servers with identifiers hashes 2, 5 10, and 15. . We want to `setr` a key with hash 7. So the primary replica will be server 10, with secondary replicas on 15 and 2. If we now add a server with identifier hash 9, this

means that our key should be stored on servers 9, 10, and 15. Therefore, the new server, 9, may become the primary replica for some keys whose primary replica was originally 10 (specifically, for those keys whose hash is between 9 and 10). In addition, all keys stored primarily on server 2 or 5 will now use 9 as a tertiary or secondary replica, respectively. Server 9 must therefore, during its initialization, copy certain values from those servers.

- Verify that distributed commits fail if servers don't agree on the epoch or server identifier. After a view change, there may be a short delay during which time some servers are not yet informed about the view change. In addition, the view may change in between the time that the client requests the server list from the view leader and the time that it attempts to perform the commit. In either case, a mismatch in expected epoch (between the client and any of the servers) should result in a failed commit.

- Verify that in simultaneous attempts to commit to the same key, at most one attempt will succeed. If, between the time that a client begin a `setr` operation on a key and the same client completes (by either committing or aborting) the operation, another client attempts to initiate a commit operation on the same key, the second operation should fail. Any of the servers should recognize that there is a pending operation on the given key and not vote for it. Simultaneous operations on other keys, even those hosted on the same server, should succeed.

- Verify that failed commits are correctly aborted. If, for any reason, a `setr` fails to commit, all waiting servers must be notified of the abort so that the operation can be re-tried in the future.

4 Hints

- You may use the SHA1 algorithm from Python's `hashlib` module to calculate hashes of keys and server IDs. For example:

```
def hash_key(d):
    sha1 = hashlib.sha1(d)
    return int(sha1.hexdigest(), 16) % HASH_MAX
```

- The most challenging part of this assignment is rebalancing. For the rebalancing, as well as for the participant-to-participant recovery (if you choose to implement it), it may be necessary to use concurrency within a server, depending on how you approach the problem. You may use Python's thread facility, which allows you to have multiple threads of execution within a single process. Unfortunately, concurrency increases the complexity of your program, similarly to how concurrency makes distributed systems more complicated.

  Assuming that we want to run a function named `f` concurrently with the main thread, we can do something like this:

```
import threading

def f():
    print("Do stuff here")

thread = threading.Thread(target = f)
thread.start()
```

  Just as we use locks to coordinate critical sections in distributed systems, we can use Python's thread locks to coordinate critical sections within a single process:

```
import threading

# create the lock
lock = threading.RLock()

with lock:
    print("Do critical section here")
```

5 Submission

You are obligated to write a `README` file and submit it with your assignment. The `README` should be a plain text file containing the following information:

- Your name
- Instructions for running and testing your program.
- The state of your work. Did you manage to complete the assignment? If not, what is missing? If your assignment is incomplete or has known bugs, I prefer that students let me know, rather than let me discover these deficiencies on my own.
- Any additional thoughts or questions about the assignment. Were the instructions clear? Were you given enough support in developing your solution? Was the assignment interesting?
- The additional design considerations described above.

Please save your client program as `client.py`, your server program as `server.py`, and your view leader program as `viewleader.py`. Submit these files, as well as your `README` file any other source code necessary to run your project, on Moodle.