COMP 360 — Programming Task 2

Jeff Epstein

Due 20 Oct 2016

1 Introduction

We will continue building our distributed file system, which we started in the previous assignment.

In this part of the project, we're going to extend our program to support multiple, coordinated servers. The server group will be managed by a *view leader*, which will receive *heartbeats* from the servers. In addition, the view leader will host a centralized lock server. We will also extend our RPC interface to support these new features.

Failure detection Please read section 8.2.4 in your textbook.

The server will, at regular intervals, send a heartbeat message to the view leader. If a server misses a few heartbeats, the view leader will consider that server to have failed.

Group view Please read section 8.2.1 in your textbook.

Based on the receipt of heartbeats, the view leader will be able to determine the set of currently available servers. This set will be known as the *group* or *view*.

Centralized locking Please read section 6.3.2 in your textbook.

We will provide a centralized locking mechanism, whereby clients can acquire mutex locks from the view leader.

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

3.1 View leader

In addition to the `client.py` and `server.py` programs that you worked on last time, we introduce a third program, `viewleader.py`, that, when run, will accept RPC requests on the first available port in the range 39000 to 39010. The exact RPC requests to be supported by the view leader are described below. The view leader, like the server, runs indefinitely, unless it is killed by typing Control-C. It may output diagnostic messages to the console, but it is not required to output anything at all.

3.2 Heartbeats

We will use heartbeats from the servers to the view leader as a mechanism for failure detection. The new version of our distributed application will support multiple instances of the server program. As in the previous version, each server will attempt to listen for RPCs on a port in the range 38000 to 38010. In addition, when the server starts, and at 10 second intervals thereafter, it will attempt to call the `heartbeat` RPC on the view leader. Each heartbeat call should include:

- The server's unique identifier. On startup, each server will select a pseudorandom unique identifier for itself, to be sent in all heartbeats. We use the ID to distinguish crashes (with data loss) from temporary network failures (without data loss): if the server process crashes and loses data, when it restarts, it will have a new ID, and will be treated as a new server, even if it is listening on the same address and port.
- The port on which that server accepts RPCs. (It is permissible, but not necessary, to include the address on which the server is listening, since the address is returned by the `socket.accept()` call when the heartbeat is received.)

If the server is unable to call the heartbeat RPC (because the view leader isn't running, or has crashed, or is rejecting the server's heartbeats), it should continue attempting to send heartbeats anyway (in case the view leader comes back).

3.3 Views

The view leader, upon receiving a heartbeat from a server, should store the heartbeat information, including the time that the heartbeat was received, and the address and port where the corresponding server listens for RPCs. If more than 30 seconds have elapsed since the last heartbeat was received from a given server, that server is considered to have failed. Once a server has failed, it will not be allowed to un-fail: that is, the server will permanently be marked as failed by the view leader, and as a consequence, any future heartbeats from the same server will be rejected, and the view leader will not consider that server (with the same ID) as working. However, it is possible to kill the server and restart it, which will result in a new server process running at the same address, but with a new ID; the view leader should accept heartbeats from this server.

In response to receiving a heartbeat, the view leader should send the server an acknowledgment, containing an indication about whether the heartbeat was accepted or not. If it was not, because the server had previously been marked as failed, the server should print out to the terminal a message indicating that its heartbeat has been rejected, but it should continue to try to contact the view leader.

We can add servers to the group. The view leader should be prepared to accept heartbeats from a new, heretofore unknown server at any time.

Note that the view leader will constantly receive heartbeats from multiple servers. If one server has stopped sending heartbeats, it should not affect any other servers. That is, the time of the last received heartbeat must be recorded separately for all servers. The set of currently active, non-failed servers, as seen from the view leader, is known as the *view* or *group*.

The view leader will also keep track of the *epoch*, which, starting from zero, will count changes to the view. That is, every time the view is changed, by adding or removing a server, the epoch will be incremented.

The view leader's current view of non-failed servers may be queried via the `query_servers` RPC. This function should return (a) the view, i.e. a list of currently active (i.e. non-failed) servers in the group view and (b) the group view's current epoch. The servers should be identified by their address and RPC listening port, separated by a colon, for example `127.0.0.1:38000`.

3.4 Locks

The view leader will provide an interface to centralized mutex locks, which are accessible via the `lock_get` and `lock_release` command on the client.

Each lock has a name (a string), identifying the resource to be locked. Each lock must always be in one of two states: *held*, which indicates that the corresponding resources is locked; or *unheld*. If a lock is held, it must be held by a *requester*, an entity (possibly a client), identified by a string.

Initially, all possible locks are unheld. A requester may request a lock with the `lock_get` RPC. If the lock is unheld, the lock will become held by the requester. If the lock is already held by the same requester, nothing happens. If the lock is held by another requester, then the requester is *enqueued* in that locks waiter queue, and the RPC returns a "retry" status.

A requester may release a lock that it holds with the `lock_release` RPC. When a lock is released, if there are no other requesters waiting in the lock's queue, the lock becomes unheld. If there is a waiting requester in the queue, the first waiting requester will hold the lock.

A locking walkthrough To illustrate the complicated semantics of locks, let's take a concrete example. Let's imagine a hypothetical situation where we are using locks to ensure serialized access to bank accounts by ATMs. In this case, the lock name will be the account identifier (the resource to be locked), and the requester identifies the ATM (the entity requesting exclusive access).

Therefore, the following command shows how the requester named "atm1" requests, and is granted, access to a certain account.

```
eppie@tenger:~/phase2$ ./client.py lock_get bob-smith-account atm1
{'status': 'granted'}
eppie@tenger:~/phase2$
```

At this point, atm1 holds the lock, and assuming that all clients obey the protocol, atm1 has exclusive access to resource "bob-smith-account". For example, assuming that corresponding account's balance is stored in a key named `bob-smith-balance` on the server, atm1 could now safely issue a `get/set` transaction that would be unsafe if executed concurrently with another transaction.

What happens if we run the same command again?

```
eppie@tenger:~/phase2$ ./client.py lock_get bob-smith-account atm1
{'status': 'granted'}
eppie@tenger:~/phase2$
```

Because "atm1" already holds the "bob-smith-account", re-locking is idempotent, and simply confirms that atm1 still holds it.

What happens if another requester tries to get access to a locked resource?

```
eppie@tenger:~/phase2$ ./client.py lock_get bob-smith-account atm2
{'status': 'retry'}
Waiting on lock bob-smith-account...
{'status': 'retry'}
Waiting on lock bob-smith-account...
{'status': 'retry'}
Waiting on lock bob-smith-account...
```

Here, "atm2" is trying to get exclusive access to the same account, but it can't. The client sends the `lock_get` RPC, which replies with "retry," indicating that the lock is held by someone else, but that atm2's request has been enqueued. Because there are no other requesters waiting for that lock, atm2 will automatically get the lock when atm1 releases it. The above command will wait indefinitely, retrying the `lock_get` RPC every 5 seconds until you either kill it (with Control-C) or it gets the lock. While it's waiting, let's open another window and release the lock:

```
eppie@tenger:~/phase2$ ./client.py lock_release bob-smith-account atm1
{'status': 'ok'}
eppie@tenger:~/phase2$
```

As soon as we do this, atm2's pending request is granted, and the client returns success:

```
eppie@tenger:~/phase2$ ./client.py lock_get bob-smith-account atm2
{'status': 'retry'}
Waiting on lock bob-smith-account...
...
{'status': 'retry'}
Waiting on lock bob-smith-account...
{'status': 'granted'}
eppie@tenger:~/phase2$
```

Now atm2 holds the lock, and can release it with `lock_release` when it's ready.

A lock may have multiple waiting requests at the same time. In the interest of fairness, requests are granted to waiters in the order that the initial request is received. If a requester is already holding a lock or is already in its wait queue, it may not be entered a second time into the queue. A given requester *may* hold or wait on multiple locks at the same time.

3.5 Client command line

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

- The `query_servers` command should invoke the `query_servers` RPC on the view leader, and return the view leader's current epoch and active servers. This command takes no parameters.

```
eppie@tenger:~/phase2$ ./client.py query_servers
{'epoch': 3, 'result': ['127.0.0.1:38000', '127.0.0.1:38001']}
```

- The `lock_get` command takes two parameters: a lock name (identifying the thing being locked) and a requester ID (identifying the process that is requesting it). Both are arbitrary strings. This command should invoke the `lock_get` RPC on the view leader. If the RPC returns `"retry"`, indicating that the lock is currently owned by another process, the client should wait 5 seconds and then automatically repeat the RPC, until it achieves either success (meaning that it has acquired the lock) or failure (for example, if it can't contact the view leader).

```
eppie@tenger:~/phase2$ ./client.py lock_get bank_accounts atm1
{'status': 'retry'}
Waiting for lock bank_accounts...
{'status': 'retry'}
Waiting for lock bank_accounts...
{'status': 'retry'}
Waiting for lock bank_accounts...
{'status': 'granted'}
```

- The `lock_release` command takes the same parameters as `lock_get`. This command should invoke the `lock_release` RPC on the view leader. If the specified requester currently holds the given lock, it will release it. If the specified requester is waiting for the given lock, it will be removed from the wait queue. If the given lock does not exist, or if the specified requester is neither holding or waiting for that lock, an error should be displayed.

```
eppie@tenger:~/phase2$ ./client.py lock_release bank_accounts atm1
{'status': 'ok'}
```

3.6 RPCs

In addition to the RPCs implemented in the previous assignment, your application must support the following:

- RPCs from the server to the view leader
  - `heartbeat` – This RPC is made automatically at intervals from each server to the view leader. It may not be invoked directly by the user from the command line.
- RPCs from the client to the view leader
  - `query_servers` – Invoked by the client command of the same name. This RPC takes no parameters.
  - `lock_get` – Invoked (possibly repeatedly) by the client command of the same name. This RPC takes two parameters: a lock name, and a requester ID. Both are arbitrary strings.
  - `lock_release` – Invoked by the client command of the same name. This RPC takes the same parameters as `lock_get`.

In addition to the command-line parameters in the previous assignment, both your server and client programs must support an optional `--viewleader` parameter that specifies the address of the view leader. The default value should be `localhost`.

Please note that the changes described above to the client program mean that the client can communicate directly with both the view leader and the servers. The `print`, `set`, `get`, and `query_all_keys` RPCs are sent to the server, while the `query_servers`, `lock_get`, and `lock_release` RPCs are sent to the view leader.

4 Hints

- To generate the random server ID, you can use the `uuid.uuid4` function or an appropriate function from the `random` module.
- To determine the current wall-clock time of the computer, you can use the `time.time` function, which will return the number of seconds since January 1st, 1970. You can compare timestamps simply by subtracting them.
- As discussed above, your view leader should be prepared to handle the following cases, among others, pertaining to heartbeats:
  - new server is created and starts sending heartbeats – new server is recognized and heartbeats are accepted
  - server stops sending heartbeats, due to crash or permanent network failure – server is marked as failed
  - server temporarily stops sending heartbeats, due to temporary network failure or server load, but then resumes sending heartbeats (with the same server ID) – server is marked as failed anyway, and heartbeats after recovery are rejected

  It's important that you test all of these cases before submitting your work. Simulating a crashed server is easy enough: just type Control-C to kill the server program. But how can we simulate a temporary network failure, resulting in missing or delayed heartbeats, eventually followed by correct heartbeats?

  In a Linux environment, it's possible to suspend a program without killing it. If you suspend your server, it will still be in memory, paused, but it won't be running (and therefore won't be sending heartbeats). To suspend a program, just press Control-Z. You should see something like this:

  ```
  eppie@tenger:~/phase2$ ./server.py
  Trying to listen on 38000...
  ^Z
  [1]+  Stopped                 ./server.py
  eppie@tenger:~/phase2$
  ```

  The "Stopped" message indicates that the program has been paused and moved to the background, while you are given a command prompt. To resume the program, use the `fg` command to move it back into the foreground, at which point it will continue running normally.
- As discussed above, each server must send regular heartbeats to the view leader. But we have a problem: the server is usually waiting for incoming RPCs of its own, via the `accept` function. How can we ensure that we can send regular heartbeats, while still accepting new RPC connections?

  There are several possible solutions. A complicated and error-prone technique involves using multi-threading; I will not discuss this technique here. You could also use non-blocking I/O, which is outside the scope of this class. Instead, I recommend (but do not require) a single-threaded approach based on timeouts.

  It's sufficient to interrupt the server's `accept` call after a certain time interval. For this we can use the socket's `settimeout` method, which takes as a parameter a number of seconds after which `accept` will be interrupted. If an incoming connection is accepted before the given time period has elapsed, `accept` will return the connection's socket as usual. If no new connection is accepted, `accept` will will throw a `socket.timeout` exception, which your code should catch. In handling the exception, you have the opportunity to send a heartbeat and immediately resume `accept`-ing.
- The view leader must be able to provide the list of non-failed servers in response to the `query_servers` RPC. In particular, the view leader must return the servers' TCP endpoints, in the form *address:port*;

for example `127.0.0.1:38000` or `10.0.0.50:38005`. The view leader's response should represent its knowledge about the state of the group, based on its receipt of heartbeats from servers. Each heartbeat includes the server's listening port, but how does the view leader know the host address whence the heartbeat was sent? This is returned by the `accept` call:

```
sock, (addr, accepted_port) = bindsock.accept()
```

In the above code snippet, `addr` will be a string containing the address of the computer that is initiating the connection. Note, however, that `accepted_port` will *not* be the server's RPC listening port, but rather an arbitrary port chosen by the operating system for the source endpoint.

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?

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.