COMP 360 — Programming Task 4

Jeff Epstein

Due 8 Dec 2016

1 Introduction

We will continue building our distributed file system. In this part of the project, we're going provide redundancy of the view, using a consensus algorithm.

This phase of the project does not require the features from the immediately preceding phase (distributed commits and distributed hash table). Therefore, if you choose, you may build this phase on top of your code from either the second or third phase, or use my code.

Bully election algorithm Please read section 6.5.1 in your textbook.

We'll use a bully algorithm to allow the redundant views to select a leader.

Consensus Please skim the papers *The Part-Time Parliament* and *Paxos Made Moderately Complex*, available on the course web site. I also suggest reading the Wikipedia article.

We'll use a quorum-based consensus algorithm to ensure that the view replicas are consistent.

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

Although we added data replication in the previous assignment, our application still has a single point of failure: the view leader. In the interest of availability, we want to replicate the state stored on the view leader onto *view backups*, one of which will step in to replace the view leader if the view leader crashes. We collectively refer to the view leader and its backups as the *view replicas*.

The view leader stores the following state that must be consistent across its replicas:

- The lock table – who owns which lock, who is waiting on which lock
- The view – which servers are active and when their last heartbeat was received

Because the state on the view replicas should be consistent at all times, switching to a backup should not impede the performance of the cluster (although a short delay or period of unavailability while an election is held and servers switch to the new view leader is acceptable).

Redundancy between the replicas will be ensured by a quorum-based consensus algorithm. Whenever a change in the view leader's state is made, the same change will be *synchronously* reflected on a quorum of replicas. At the same time, the replicas will synchronize their logs. If a quorum cannot be achieved, the change will not be reflected at all and the command that initiated it will be rejected.

3.1 Command line

The client program, beyond the commands required in the previous assignment, need not support any new commands.

In previous versions, `client.py` and `server.py` supported an optional command-line flag `--viewleader` that could be used to specify the hostname where the view leader was running (for example, `--viewleader localhost` or `--viewleader 10.0.0.5`). In this version, we are going to modify the behavior of this flag. Now, the `--viewleader` flag should specify a comma-separated list of endpoints where view replicas are expected to run. An example of the new format of this parameter could be, for example, `--viewleader compy1:39000,compy2:39000,compy3:39000`.

Furthermore, the `viewleader.py` program will also support the `--viewleader` flag. When `viewleader.py` starts, it should check to ensure that its own listening endpoint is among those listed. If not, it should emit an error message and terminate.

The default value of the `--viewleader` flag for all programs should be `*myhost*:39000,*myhost*:390001,*myhost*:39002`, where `*myhost*` is replaced with the hostname of the computer that the program is being run on (retrievable by a call to `socket.gethostname()`. That is, if left unspecified, all programs should assume that all view replicas are running on the current machine.

3.2 Leader election

Your `viewleader.py` program will be used to start the view replicas. Your program should already bind an available port in the range 39000 – 39010. Running this program multiple times will start a new view replica each time.

The endpoint addresses of the view replicas is static (i.e. cannot change while the cluster is running), must be known *a priori*, and must be set via a command-line parameter on the `client.py`, `server.py`, and `viewleader.py` programs. You may assume that all programs connecting to your cluster are given the same value for this parameter. When `client.py` and `server.py` want to contact the view leader, and when the view leader wants to contact its backups, they may do so by connecting only to those endpoints explicitly given on the command line as an argument to the `--viewleader` flag.

The set of specified replicas will be used to perform an election and determine which among them is the view leader. We will use a primitive form of the bully algorithm: the highest (by lexicographical sort) endpoint that corresponds to a working view replica shall be the leader. In other words, a program wishing to connect to the current leader must simply sort in ascending order the list of replicas, and choose the last. If it can't connect to the last replica, it will try the second-to-last, and so on. If no working replicas can be contacted, the command fails.

A benefit of this simplified algorithm is that it isn't necessary for your code to actually hold an election: the leader of the replicas is discovered each time a program connects to it. A consequence of this approach is that the view leader knows that it is the view leader only indirectly: programs connect to it, *ergo* it must be the view leader.

You may optionally implement an extension of the election system, wherein view replicas verify that they are, in fact, currently the leader before accepting a command. Under this approach, each replica would keep track of who it thinks the current leader is. If a command is sent to the leader, the command is replicated and processed normally. If a command is sent to a non-leader, the replica should reject the command, responding with a message indicating who the leader actually is, and the client should retry the command on that replica. If the indicated replica can't be contacted, the client can force the replicas to hold an election and determine the new leader.

3.3 Consensus

We will use a quorum-based consensus algorithm to replicate the view leader's state among its replicas.

Most of the logic affecting view leader's state can remain unchanged since phase 2 of this project. However, each event affecting the state of the view leader (specifically, any lock command, as well as heartbeats) must be replicated to the view backups. Replication is coordinated by the view leader, and should be transparent from the perspective of other programs (that is, heartbeats will be sent from the servers only to the view leader; lock commands will be sent from the client only to the view leader).

Each view replica must maintain a *log* of commands. We distinguish three commands that can affect the state of the view replicas:

- A heartbeat, sent from a server to the view leader. Contains the time the heartbeat was initially received by the view leader, the identifier and endpoint of the server it pertains to.
- A lock acquisition request. Sent from a client to the view leader. Contains name of lock and name of requester.
- A lock release request. Sent from a client to the view leader. Contains name of lock and name of requester.

When the view leader receives one of these commands, it must attempt to synchronize the event with its replicas *before* acknowledging the initial command. (In terms of our RPC system, this means that the view leader must perform synchronization before the RPC implementation completes and returns its result.) If synchronization fails (due to failure to achieve quorum), the view leader must reject the command. In case of rejecting a command, the view leader should not update its state and should return a failure message to the invoker (the client or server).

We will now describe the consensus algorithm, which is similar to Paxos. In order to accommodate the command log, we will declare an equivalency between the *proposal number* and the *length of the command log.*

1. The view leader sends a Prepare message to all replicas. The message should include the command itself, as well as the proposal number, which is equal to the position in the view leader's log of the latest committed command.

2. The view replicas receive the Prepare message.

   - If the proposal number is equal to the length of the backup's log, it means that the proposer (the view leader) and the acceptor (the view backup) are equally up-to-date; they have the same log. The replica should respond positively to the leader's proposal.
   - If the proposal number is less than the length of the backup's log, it means that the proposer (the view leader) is behind, relative to the other replica; the backup has more logs than the leader does (this situation can arise if the leader becomes temporarily unavailable, but then recovers). In this case, the backup must provide the leader with the missing logs along with its positive response.
   - If the proposal number is greater than the length of the replica's log, it means that replica is behind, relative to the leader; the leader has more logs than the replica. In this case, the replica should respond positively, and must specify in its response how many additional log entries it needs in order to become up-to-date.

   In any case, when the replica responds positively to the Prepare message, it promises that it will not accept another proposal of a lower number.

3. The view leader collects the replicas' responses. If any of the responses provide missing log commands, the leader should *replay* the commands and them to its own log. If the view leader does not receive a response from a quorum of replicas, then the command fails, and the consensus algorithm does not proceed further.

4. If the view leader does receive a response from a quorum of replicas, it should respond by sending an Accept message to each of the replicas that responded. The Accept message should include the revised proposal number (based on the updated length of the leader's log, which should now be as long or longer than the log on any of the view backups). The Accept message should include any missing logs requested by that replica, as well as the new command.

5. Upon receiving the Accept message, each replica must ensure that it is allowed to accept the given Accept message, in that it has not yet received a Prepare message for a higher number. If the Accept message contains additional log entries (requested by the replica in its response to the Prepare message), they should be appended and replayed before applying the new command.

When a view replica receives missing log entries from another replica, it should *replay* them and add them to its own log. By "replay," I mean that the receiving replica should execute the commands as if they were received by that replica directly, even though in fact they are being relayed by another replica. Heartbeats in particular requires special care when catching up, because they are timing sensitive: *when* a heartbeat is received is critical in determining whether the corresponding server will be marked dead or alive. To ensure consistency of state between replicas, therefore, the replica that initially receives the heartbeat from the server must mark the corresponding log entry with the time that the heartbeat was delivered, and other replicas must apply the command to their states as if they had received at the same time. For example, let's assume that replica 1 has the following state:

```
Lock X is held by requester M
Last heartbeat from Server 98 at time 1479624187.1194036
Server 54 is dead
```

Replica 1 receives from replica 2, the view leader, the following log entries:

```
lock_get X N
heartbeat 98 1479624197.101123
heartbeat 11 1479624101.023232
```

The final state stored at replica 1 should be as follows:

```
Lock X is held by requester M, requester N is waiting
Last heartbeat from Server 98 at time 1479624197.101123
Last heartbeat from Server 11 at time 1479624101.023232
Server 54 is dead
```

Note that the heartbeat from server 11 caused that server to be added to the view, even though the heartbeat was initially received in the relatively distant past. Replica 1 should then apply the usual rules to determine that server 11 is dead.

Note that under this approach, a replica's command log will grow indefinitely. You may optionally implement an extension of this system, under which the log will be *flushed* (i.e. cleared out) at intervals, to prevent indefinite growth. To accomplish this, a view replica, upon noticing that its log is larger than some threshold value (say, 100 entries), can issue a flush command to the other replicas. Note that it is safe to flush the log only when we know that the log entries we are removing will never again be needed and have already been applied on all replicas. In other words, flushing the log requires agreement among *all* replicas, not merely among a quorum.

Note that the epoch is considered part of the view, and should be updated together with other view information in response to heartbeats. It might be easiest to avoid storing the epoch as a variable, and instead to recalculate it on demand from the log, when a `query_servers` RPC requests it.

Rather than replicate each heartbeat to a quorum of replicas, you may optionally choose to optimize the process. Consider that when the view leader receives a heartbeat from a server, most of the time, the view will not change. Applying the replication algorithm for each heartbeat introduces unnecessary delay while a quorum of replicas is brought up to date. It would therefore be more efficient to replicate only view changes. Under this approach, the view leader will receive heartbeats, but will not synchronize them with its replicas unless a view change results. The ultimate effect should should be the same (i.e. synchronized view) but with less network traffic. Keep in mind that since only the leader has a complete record of heartbeats, only the leader can determine when a server should be removed from the view, and this decision will need to be replicated, as well.

Properly speaking, the state of consensus is set only in a majority of the available replicas. Therefore, a client reading state from the view leader is assured of getting up-to-date information only in the case that the view leader happens to be up-to-date. A fully correct client, wishing to call the `query_servers` RPC, would need to get identical results (including proposal number) from a quorum of view replicas in order to know that the result is up-to-date. In your program, however, you may omit this detail, and you may assume that it is isafe to call `query_servers` on the view leader.

Note that if your implementation of phase 3 requires the view leader to explicitly notify the servers of a view change, that may cause problems when there are multiple view replicas. That is, we don't want each replica to kick off a separate rebalancing every time there is a view change. Therefore, for the purposes of this phase, you may disable rebalancing without penalty. Alternatively, you can suppress such redundant notifications by disallowing view change notifications from view backups.

3.4 Test cases

Some situations that you must test for:

- Death of the view leader should not impede the operation of the cluster, as long as the cluster still has quorum. In particular, cluster state (lock table and view) should be maintained. For example, given three active view replicas, one of which is leader, and assuming that a client Z acquires lock X, if I kill the view leader, causing another replica to become leader, the state of the lock table should be identical across the change in leadership. This can tested by having a client W try to acquire the same lock X: if the lock table is correctly replicated, the new client will be notified that the lock is not free.
- Death of a non-leader replica should not impede the operation of the cluster, as long as the cluster still has quorum.
- When number of active replicas drops below quorum, the cluster stops updating. View changes, as well as lock operations, will fail.
- When the number of active replicas is below quorum, and some replicas recover (i.e. become active again), cluster operations should resume.

4 Hints

- Any operation changing the view's data store (acquire lock, release lock, heartbeat) will require a consensus operation with its replicas, and may trigger an election, as well (if the previous view leader isn't available). It's important that both the
- The default value of the `--viewleader` flag on `client.py`, `server.py`, and `viewleader.py` should indicate view replicas listening on ports 39000, 39001, and 39002 of the current machine. The host name of the current machine can be acquired from the `socket.gethostname()` function. Therefore, an appropriate default value of `--viewleader` will be returned by the following function:

  ```
  def default_viewleader_ports(n=3):
      hostname = socket.gethostname()
      servers = [ "%s:%s" % (hostname, port) for port in range(VIEWLEADER_LOW,
          min(VIEWLEADER_HIGH, VIEWLEADER_LOW+n))]
      return ",".join(servers)
  ```

  In the above code, we assume that `VIEWLEADER_LOW` is 39000 and that `VIEWLEADER_HIGH` is 39010.

  When `viewleader.py` starts, it should find out its own hostname and listening port, and ensure that its endpoint is among those given by the `--viewleader` flag (or its default).
- To prevent timeouts from the client side while the view leader is synchronizing with its backups, ensure that the view leader is using relatively short timeouts to contact its replicas. This way, if a view backup has failed, the view leader will be able to quickly detect this failure and continue synchronization (or not), rather than let the client request time out.

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.