Jack Davey

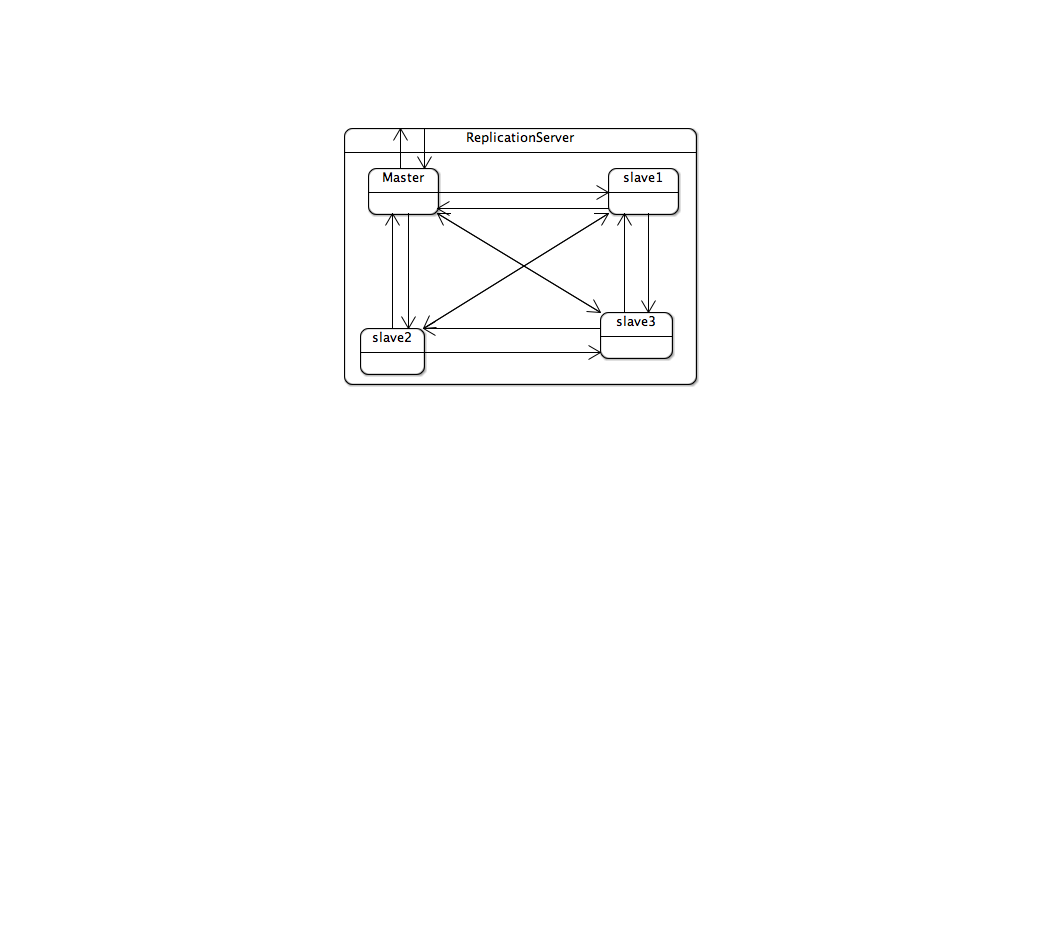
**Design Document for Basic Availability**

This document sets out the design for the basic availability functionality that I will implement to complete the second phase of the project.

**Simulation Parameters**

Unlike with the eventual consistency that I previously implemented, there is a lot more scope for customization with basic availability. Because of this, I plan to move all these parameters into their own object. When a system component needs one of these parameters, they will access the value through accessor methods. Mutator methods will be package private, so that they can only be accessed from the user facing code.

**Main Architecture**



The overall architecture of the application will remain unchanged for the most part. The key difference is in the individual replication servers. In the eyes of the replication overseer and the replication marshaller, nothing will have changed. These processes still send and receive exactly the same messages as they did before. Internally however, these processes now use a lot more concurrency, and can also send back different messages in response.

Each server has one Master node. This is the node that deals with all of the same work as the original replication servers. The master also maintains a list of several slave servers. These servers are responsible for keeping track of all the same data as the master does.

Every so often, each server pings the master, if a master has not been heard from for a preset period, then all the slave servers elect a new master from amongst themselves and the old master is relegated to being a slave. all the servers vote by exchanging messages with each other to decide upon the new master.

In order to have this voting, all these child processes are wrapped inside one main process. This handles the passing of the messages onto the various child processes, and is also the place where the children send their votes.

When an update reaches a replication server, it sends it to all of its slaves. In the case of a slave receiving the update, it sends it back to the master so that it can propagate the update as before. When an update reaches a slave in this manner, it randomly decides to accept the update. This is done through the use of random numbers. If the value is accepted, then it is stored in the slave. if not, then it is rejected. An update must be written to a certain number of the slaves for an update to be successful. Similarly when data is read off a server; a certain number of reads must see the data for it to be effective.

All of these parameters would be configurable as mentioned above, so as to simulate different levels of basic availability.

Another key change that I plan to implement is the ability to introduce the possibility of an replication server potentially failing to deliver a message. This is done by each message having a certain possibility of it being corrupted when it reaches a particular replication server. The likelihood of message corruption can be configured.