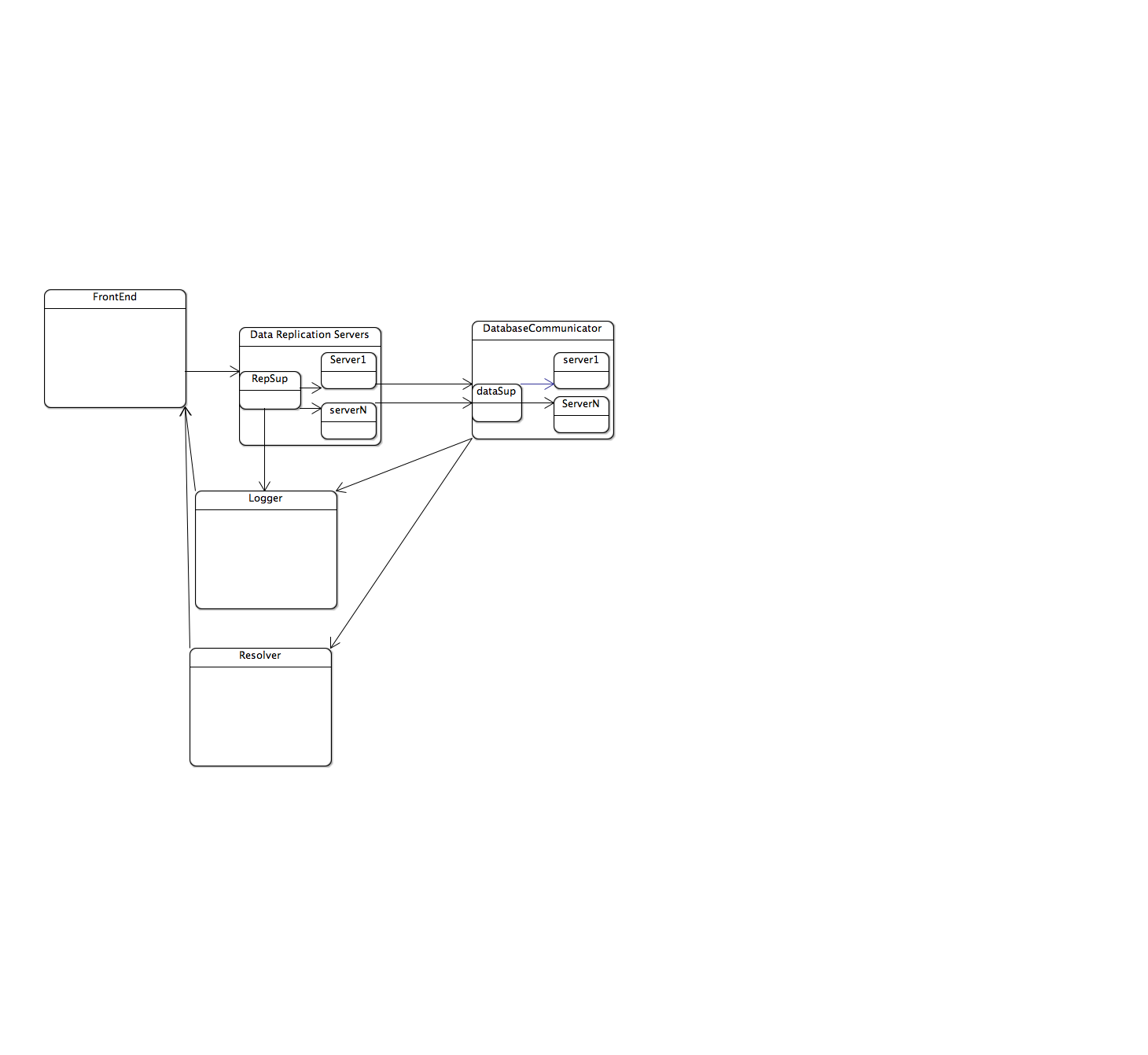
Jack Davey

**Eventual Consistency Design Document**

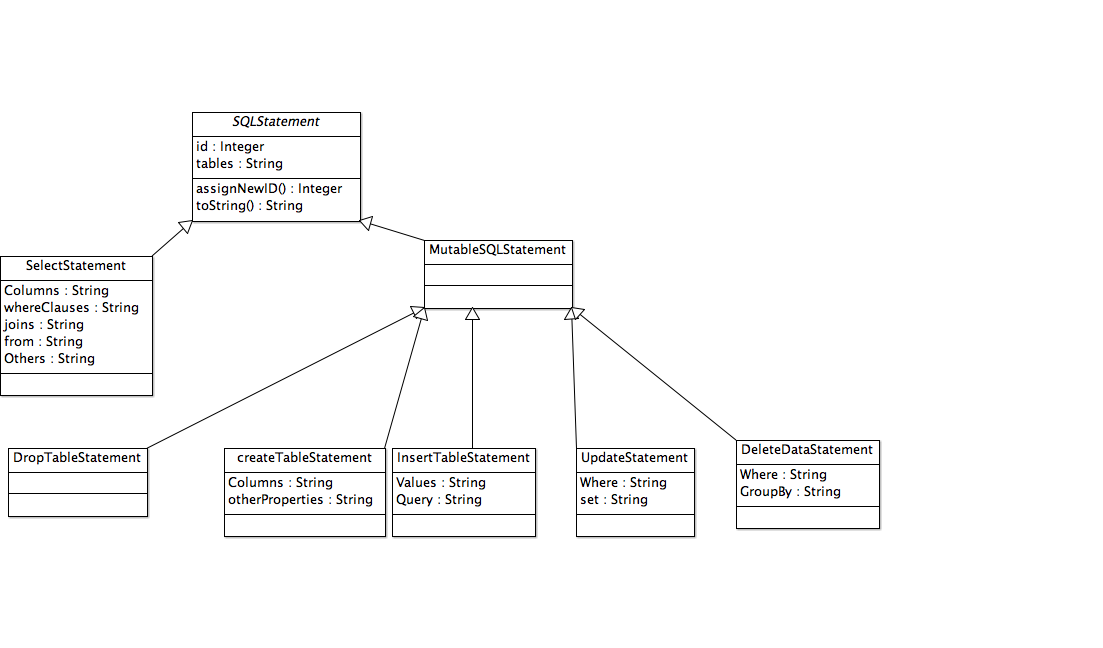
This document sets out the overall architecture for my system. As I will be using the Akka framework to implement this system, I decided the best way to represent my architecture would be using a process network diagram. This allows me to identify the main processes that the system is made up of and show how they communicate together.



**Outline of main component functions**

**FrontEnd**

This is the only part of the system that may not be an Akka Actor, The reason for this is because it needs to communicate with the end user, and therefore the communication cannot be asynchronous. Its main responsibility is to take messages from the end user and send them on to the appropriate parts of the system. In doing this, it should also make sure that these pieces of data are encoded in the right form. This mainly consists of taking the web service requests that are coming into the system and translating those into an SQL object. These SQL objects have a version number associated with them, and it is these that we use to ensure eventual consistency. The class diagram for this is shown below.



**State replication Servers**

These processes are responsible for keeping multiple copies of the same data in a lot of the main query operations. These are queried at random to find out what pieces of data they have available to them. All requests to these servers are initially sent to a supervisor actor. This supervisor passes jobs to the worker servers. At set intervals, these servers send data to the database server to make their data consistent.

**Database Communicator**

This process has the job of communicating with the database, it creates a new communicator actor for each new request it receives. It is only responsible for dealing with requests directly pertaining to the eventual consistency part of the application. Some requests that need to be fed back to the frontend are carried out by the frontend section of the application.

**Logger**

Because the actor model relies on asynchronous communication, a user may not always get all the information associated with a request. As an example, when an update is made to the database, that update may not be made fully consistent for a while after the user interaction has finished. In order to counter this, all processes in the system send messages to a logger process. Upon receiving a message from the frontend, the log can provide a textual representation of all the logging messages it has had.

**Resolver**

The resolver process is responsible for reporting all instances where the eventual consistency algorithm has had to choose one value over another. It receives notifications from the database communicator process when this happens. Then upon receiving a message from the front end, it can send all these to the user. Another message to the front end will force this process to choose the discarded data state.

**Algorithms**

I will now look at the algorithms that I will use to achieve eventual consistency. Creating tables and dropping tables are done by sending the SQL object straight through to the database communicator. If we are dropping a table, then a message is sent to the replication table to ensure that all updates in the system relating to that table are removed from the system.

Because the eventual consistency algorithm that I am implementing requires me to have versions of data, I need to design a mechanism to cope with that requirement. Therefore I propose to have a BatchQuery class. This has a list of vector clocks, as well as a list of SQL queries, all of these queries must share at least one piece of data in common. It will be possible to merge queries into the batchQuery, as well as finding out if a query can be merged. Whenever this happens, the vector clock for that node is updated.

When it comes to removing, updating, or inserting a piece of data, then we start off by running the request on the backend database. If this works, then we send the data to the replication servers, if it fails; we send an error back to the user. This is done so that the user gets some feedback from the initial request as to whether the update will or won’t be applied.

Once at the replication servers, a server is chosen at random,, the replication supervisor sends the request through to the server. If the request is new, which it will be in this case, we create an empty list of vector clocks and insert into it a vector clock containing this initial change.

A vector clock is a pair containing the identity number of the server and the number of times it has accessed that particular piece of data. The server then holds onto the completed operation for a period of time, after which it commits all data to the back end database.

If another request comes in that affects information in the same set of data, then either one of two things happens: if the request gets allocated to an actor which has never seen that request before, then the original operation described above is carried out; if it goes back to the same host, then that host increments the vector clock for the set of data and adds the new SQL query to the piece of data. It also sends a message to the other hosts in the network, asking if they have seen that particular piece of data or not. If they have, they send all affected pieces of data to the requester and mark the old copies as invalid.

When a piece of data reaches the maximum allowed time, a final request is sent to ensure that there are no invalid pieces of data floating around, and then the piece of data is sent to the database. Any pieces of data that are marked as invalid are deleted.

For selects, the user runs the query on the database, we then choose on eof the replication servers at random, if the data from the query contains any data from the sever, the stale data is replaced.