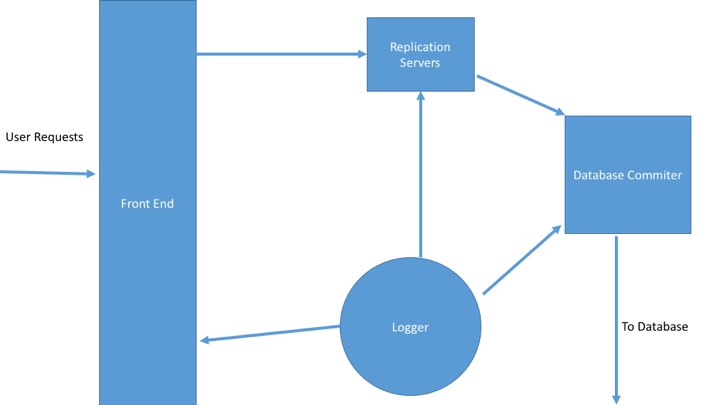
Jack Davey

**Eventual Consistency Design Document**

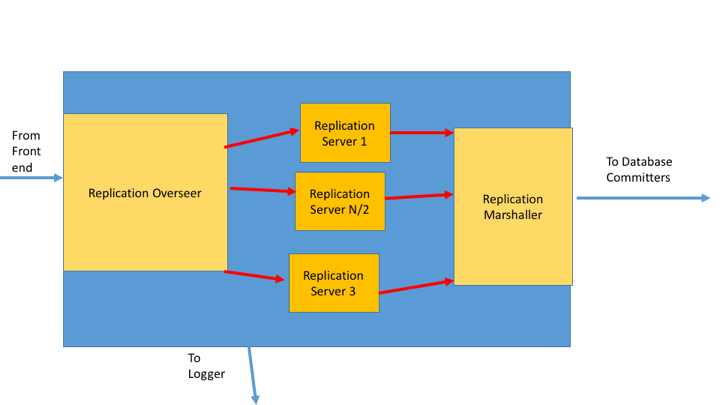
This document sets out the overall architecture for my system. As I will be using the Akka framework to implement this system, I decided the best way to represent my architecture would be using a process network diagram. This allows me to identify the main processes that the system is made up of and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



**Outline of main component functions**

**Front end**

This is the only part of the system that is not 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.



**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. If the server receives an integer, it changes the period between consistency checks to the number of seconds specified by the parameter.

There is also a replication Marshaller actor. This actor sits between the individual replication actors and the database committer. This is because the method I use to resolve consistency checks can cause duplicate data. This process is responsible for ensuring that no duplicate queries get through to the database.

**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. Examples of these include creating and dropping tables.

**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.

**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. 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. It should be possible to apply all these queries to a set of results by simulating the effect of the updates on these results.

When it comes to removing, updating, or inserting a piece of data, then we start off by attempting to create a MutableSQLQuery object for it. This does some simple validation. 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 will not be applied. We also keep a record of this update, so that we can let the user know how many updates are in the system at a given point in time.

Once an update reaches 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 backend 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 list of queries.

At set intervals, a request for all servers to become eventually consistent is issued. At this time, all servers send all of their querysets to all other servers. The record of all the updates currently in the application is also reset at this time. The other servers check that all the queries they receive are consistent and I remove any that aren’t, and then send all their work onto the replication marshaller. Once this has done its work, it then passes all the queries onto the database. The database committer then sends all queries it receives to the database.

For select statements, the user runs the query on the database, and we then choose one of the replication servers at random. The query is then run on the database and a list of maps is produced, representing the retrieved database table. The list of maps is then passed to the replication server that was chosen. The server then applies each of the queries it has stored to the set of results. In this case applying means modifying the list to make sure the effects of those queries is returned to the user.

There is also a service that allows the user to change the amount of time to wait between each consistency sweep. When this is activated, the frontend just passes that through to the replication supervisor. The replication supervisor then schedules all future requests to use that interval.