Jack Davey

**Implementing Eventual Consistency**

In this chapter, I will explain the work I did for implementing a basic version of the project that just implements eventual consistency.

As previously discussed in the section on planning, I followed a weekly iteration plan for the development of this project. The first week was spent building an extremely simple prototype that took in SQL queries in plain text and passed them through to the database. This was mainly to get me used to working with the Play framework. I had done web development before. In particular, I had developed web applications as part of the CO539 Web Development module that I undertook as part of my final year of undergraduate study at the university of Kent. This was done in PHP using the Codeignitor web framework however. The play framework takes full advantage of Scalas functional capabilities, meaning that it is much more complex than Codeignitor was. I therefore used the first week to become acquainted with the framework.

This time was useful in other ways however. Firstly, it re affirmed my decision to use the Play framework in the project. This is because it made sending results to and from the user relatively straightforward. It also provided libraries to help me accomplish common web development tasks easily, such as transforming data to and from Json.

It also made me rethink the way I was planning to do database access. Originally, I had planned to do this using the standard JDBC library for Java. I made this decision based on the fact that this tool was part of standard Java, meaning that it would be likely to be well documented and easy to use. When I implemented these features however, I found that JDBC on its own required a lot more code just to run simple queries. I therefore decided to try Anorm, the standard Database connectivity library bundled with the Play framework, and this provided to be both reliable and easy to use.

I then moved onto the second iteration. This iteration was due to be the one in which I produced a basic implementation of eventual consistency.

I therefore had to do a full design of the whole system. I started off by designing a hierarchy of SQL classes to encapsulate database queries. I made sure to make these immutable, so they could be passed around between different actors, and also made heavy use of inheritance, so that a lot of the code for all the different query types could be shared. More information is available as part of the design document included in appendix A.

I then moved on to the design of the system itself. Because my system is made up of a set of communicating process, I decided to use a process network diagram as taught to me in CO890 to model the system. This had two main advantages. Firstly, a key advantage of designing systems this way is that because you are showing how concurrent processes communicate, you can design the system in such a way that you have a better chance of avoiding the major concurrency issues , such as deadlock and livelock. This is not foolproof, but it did make me more confident than if I had used other forms of design. A second major plus point is that because the implementation tends to naturally follow the design with this strategy. I was able to start thinking about the implementation early on and deal with any perceived problems before they became a reality.

The biggest issue by far that I had to deal with in this iteration was how to merge the inconsistent requests. The issue was that in the original version of Amazon’s vector clock algorithm, all the various replication servers are constantly sending copies of the data to each other. Because my system only has a few replicas, and those replicas have no direct knowledge of each other, this would be hard for me to implement directly. I therefore had to introduce a marshaller into the system, to take the inconsistent requests form the replication server and make them consistent.

Evaluation

Even though I got everything working for the system in the end, there are several areas on which I feel I could improve. Firstly, I could extend the system to include the features offered by the plan, because I was using this week to ensure that my early deliverable is ready, and because the features of the second iteration took longer to implement than I had originally planned.

Another area on which I need to improve is the merging section of the project. Currently, this involves all replication servers sending each other messages to ensure that they do not have inconsistent data. And then sending everything through to the replication marshaller to do the final check.. While this system does wok, it is very inefficient, and I believe that there could be a simpler solution. For this reason, I want to spend some time looking at this after the early deliverable deadline date.

I also think I need to do some more work on my project planning abilities. This is because currently, Some weeks I am overestimating what I can do within the time, other weeks I am underestimating this. Therefore, it would be good if I can get to a level where I feel comfortable with the amount of work I set myself each week, but I am sure that will come in time.