Jack Davey

**Problem Analaysis**

As I previously mentioned In my technology review, The two main techniques for ensuring eventual consistency that are currently most commonly in use for real cloud systems are MVCC, currently used by CouchDB, and a system based on Vector clocks, used by Amazons Dynamo DB. In this chapter, I will look at these two consistency mechanisms and explain why I chose the approach I did for this project.

The first approach I considered was the CouchDB approach of MVCC. IF the two major approaches I considered. This approach works by having a new copy of the data created every time an update is made, the newer copies simply supersede the old versions of the data, and whenever a user finds that a mistake has been made in ensuring consistency, they can just go back through the revision history to find the information they need.

This seemed to be the simplest to implement. This is because it would be relatively to keep multiple copies of the same data, and then use the dates of the various versions to find out the ones that need to be made consistent. This is particularly possible thanks to the rich API that Java (and therefore Scala) introduced, as part of Java 8 for dealing with dates[x].

Balanced against that simplicity however, is the fact that I believe it would be much trickier to implement such an algorithm based off CouchDB straight away. When I considered the proposal, I identified two possible approaches that I could take, on the one hand, I could store the different versions of data on the database, which would mean that the data model I would be using on the backend would not be very flexible. On the other hand storing the various versions off data n application memory would avoid this problem, but would mean that the memory usage for my application would be rather large.

Because I thought that this option would be the simplest option to implement initially, I decided to see if I could find a way around these problems. I considered modifying the database so that I could make this solution workable. Eventually, I came to an approach where I would only keep multiple copies of the same data for a set period of time after they became fully consistent, once that period of time had elapsed, I would delete the stale data from the applications memory. This approach seemed to work at first glance, however when I actually started trying to implement this and produce designs for it, I found that I needed to kep track of what data was stored on the database and what was inconsistent in the application memory, While I did think this was possible, I decided to look for cleaner solutions first before coming back to this method.

The other major approach tat will consider is that provided by Amazons Dynamo database,. At first glance, this has several similarities to the model described in CouchDB above, the algorithms used to ensure eventual consistency here are much more complex however. Every time a piece of data is written, the sever that write sit adds a new vector clock to that object,. This new vector clock is made up of the sever id and a timestamp to show when it was written. When we eventually try to make all the severs consistent, we go through and remove any pieces of data that have vector clocks equal or less than our own, as these have been superseded.

The first major positive thing that I noticed about this algorithm is that it seemed a lot less memory hungry than MVCC was. This is because the data itself is not the star of the show with Amazon, it is the vector clocks that are being used to determine consistency. This translated into big savings in memory for me, as it meant that I only needed to modify the vector clock, rather than the data itself. Another major plus point was that because the data could be removed once they had been made consistent, I could make memory savings here as well.

On the other ahnd I was worried that basing my emulator on the behavior of Amazons cloud platform would be harder to implement in terms of code. This is because the algorithms used are more complicated than those used for CouchDB and I was worried that the additional complexity here would translate into additional complexity for my project overall.

I therefore decided that the additional work here would the worth the risk if was able to create an application that preformed efficiently. I was also worried that my application might not be able to scale very easily, For these reasons, I chose the approach on vector clocks. I had to adjust e algorithms significantly though to fit the situation that I was working in, I will discuss this further in the chapter on implementing eventual consistency.