Jack Davey

**Dissertation**

Rough Contents

1. Introduction
2. Literature review
3. Poblem Analysis
4. Software processes
5. Implementing Eventual Consistnecy
6. Implementing Basic Availibility
7. Testing
8. Evaluation and Comparison with Amazon’s Dyanmo Database
9. Evaluation
10. Conclusion

**The Problem**

The main aim for this project is to build an emulator that allows emulation of BASE properties backed over a traditional ACID Database. Acid transactions are the most commonly used form of database transactions in use today. ACID based transactions are also stronger than their BASE Counterparts.

ACID transactions have four main properties:

1. Atomicity. This means that these transactions are ‘all or nothing’, in other words if one part of the transaction fails, then the whole transaction should fail, and the database should be left exactly as it was before the transaction started.
2. Consistency. This is the requirement that all invariants and validation checks, such as primary and foreign key constraints, must still be valid at the end of the transaction. In other words, any ACID transaction must bring the database from one valid state to another, equally valid state.
3. Isolation Any database transaction going through the system must have no impact on any other transaction also going through the system.
4. Durability. Once a transaction has happened, the end users should never be able to see the old state of the application unless they wanted to.

There are only three properties to take into consideration when it comes to BASE transactions:

1. Basic Availability. This property states that the system should never go down, even when one part of the system fails;
2. Soft-state. This means that the consistency rules on BASE backed databases are more lax than their ACID counterparts.
3. Eventual Consistency a BASE database must be guaranteed to be consistent at some point in the future.

BASE transactions have proven extremely useful for the development of big cloud storage databases such as Amazon’s DynamoDB. This is because systems like Amazon might not need the latest data all of the time and so therefore consistency isn’t as important. The basic availability is also helpful as well, because every second something like Amazon is down, money si lost.

The disadvantage with BASE transactions is that because they are unpredictable, it makes it harder to perform good quality research experiments using them while keeping costs limited. This is where the emulator that I am going to be developing will come into play.

My project has three main objectives. Initially, I will start off by emulating only the eventual consistency property, I will then move onto adding in the property on basic availability, as well as comparing what I have done against a real cloud system by writing some sample programs.

**Technology Review**

**Background Information**

I started this project with very little background information. My project supervisor directed me to introductory articles and books to help me become familiar with the basic concepts and theory. The first of these, *BASE: An Acid Alternative* [1], was informative, but I found it difficult to get through at first. *Distributed Systems Principles and Paradigms* [2] helped somewhat, as did Joe Celko’s book [3]. This was the second text I read in my project research. It helped me solidify my understanding on what BASE and ACID transactions were, and why BASE transactions were preferred over ACID transactions in certain situations. It also introduced me to several important pieces of background information, such as the CAP theorem.

Finding information on the relevant project itself was difficult. This is because no one has tried to build an emulator like this before. This means that there did not appear to be a lot of material available to me initially. Once I clarified what the word ‘Models’ meant in the original project brief, I began to find more useful information.

Another hurdle that I encountered during my individual research was the fact that along with the consistency models being relatively new to me, most of the databases worked in different ways to the MySQL databases that I normally work with. An example of this would be Dynamo, which is a key value store. I therefore found the papers on NoSQL data management systems [4] and ‘Cloud-hosted databases: technologies, challenges and opportunities ’[5] to be useful in putting the design decisions found in a lot of these cloud databases into context.

I found *The Cloud Handbook* [6] to be particularly useful. The main reason for this is that it provided descriptions of real cloud systems and the extent to which they implement the BASE properties as well as brief descriptions of the algorithms themselves. This allowed me to narrow my search for information considerably.

In a similar fashion, I found the website at [7] useful in looking for alternative approaches to enforcing the three base properties. This is particularly true of basic availability, as there are more strategies around for this then there are for eventual consistency.

**Models For Implementation**

Once I had absorbed the basic background information on the topic I had to start looking at real cloud databases and systems in order to choose a model on which to base my system.

One of the most useful resources was Amazon’s cloud database [8]. The reason for this is that the paper describes the approach to ensuring consistency is achieved well and in great detail. While reading the appropriate chapter of *7 Databases in 7 weeks* [9], I also was able to download Risk, a database that is built around the same consistency model that Amazon uses. I found this to be one of the most promising sources, as the algorithms discussed here seem to be able to be emulated effectively.

Another highly useful approach was that provided by CouchDB [10]. I found here that the CouchDB handbook, freely available online, provided thorough and in depth descriptions of the technical algorithms used, and I was able to start coming up with some ideas on how I might implement this.

Google’s model of achieving eventual consistency was considered[12]. While I find the algorithms employed here to be particularly clever, I haven’t decided to implement them for this project. This is due to several reasons. Firstly, the Google ecosystem that supports GFS is highly complex, not only encapsulating the database, but also a lock server and various other components. This means that it would be too complex to emulate within the time available to me and also require the system to have a much closer access to the database. This might not be possible due to the fact that ultimately, the data will be stored on a standard ACID database.

**Practival Matters**

A key component of the literature review for me was finding out about the programming languages and tools that I would be using. *Scala for the Impatient*[13], and *Akka Concurrency*[14] were found to be essential in gaining a solid grasp of the Scala programming language.. I also found it useful to refer to Ian Somerville’s *Software Engineering*[15] for ideas and advice on how to plan such a large piece of work. *Play for Scala*[16] was useful for learning the Play Framework.

I also wanted to make use of the process oriented design strategies that I’d been taught in module CO890 (Concurrency and Parallelism). Therefore, a key part of the literature review was spent reviewing this material. The problem I faced ,however, is that the language that we were taught this material in (Occam Pi) uses synchronous concurrency. This is in stark contrast to the asynchronous method of message passing employed by the Akka framework. I therefore also found it incredibly helpful to review the advanced material on this topic in module CO545 (Functional and Concurrent Programming). Despite the fact that I did not take the module, and the fact that it is presented in Erlang. I found the skills and theories here easily transferrable to my own project.

**Basic Availability**

Once I had a working prototype of eventual consistency, I then started to look at the other cloud property that I wanted to emulate if there was time. This was basic availability. First of all I considered the approach used by CouchDB. This uses incremental replication, where changes are gradually copied between other servers. If one server goes down, it just copies all the changes its missed from the other servers when it comes back up.

Another approach I considered was to use MongoDB[17]. MongoDB is another popular cloud database; it is fully consistent but it uses replication to ensure it is always available. MongoDB uses a master slave schema, with one master coordinating the replication between all of its slaves.

I also looked again at the Google file system. This used a similar sort of master/slave system, except the rules and protocols were more elaborate, with only certain kinds of data going through the master, and support from other system components outside the database. I found this approach incredibly interesting, but thought it might be too complicated for the task at hand.

Other cloud databases used variations on the same master/slave theme. Neo4J, a graph database, used a similar system to MongoDB, apart from the fact that slaves could also accept writes, and the slaves synchronized with the master over time[18]. RethinkDB also used a sharding scheme similar to MongoDB19]. I started to question why several databases used the same scheme. I wondered if I might be able to find out over the course of the project.

DynamoDB uses a system that borrows from several approaches. Like CouchDB, updates are made consistent gradually, but, the choice of server is performed by using a hash on some of the data. Each node is also responsible for looking after some of the other nodes in the system, so if one goes down, data can be recovered. The Redis database also uses a similar replication strategy.

**Real Cloud Plaftorms**

As I am aiming to eventually run my application on a real cloud platform, I also did some research of real cloud systems to guide me in writing the example programs. I first of all started investigating the cloud offerings that were provided by Google[21]. The reason for this was because I already had limited experience with the platform as part of the Cloud computing module. I found that Google’s datastore was not very well suited to my needs, because it was not eventually consistent.

I then started looking at Amazon’s Cloud system[22]. because thought that it would be a good point of comparison as I am loosely basing the eventual consistency implementation on Amazon’s Dynamo Database, I then would be comparing my system to the real thing.

**Problem Analysis**

**Eventual Consistency**

When I first started developing the emulator for eventual consistency, I had several questions to answer. Firstly, I needed to decide how I was going to cope with multiple values in the same system. I also needed to work out how any inconsistent values would be resolved.

The first approach I considered was the CouchDB approach of MVCC. Of 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 rectify it.

The other major approach that I will consider is that provided by Amazon’s Dynamo database. At first glance, this has several similarities to the model described in CouchDB above, but the algorithms used to ensure eventual consistency here are much more complex. Every time a piece of data is written, the server that writes it 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 servers 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 Dyanmo approach seemed to be the simplest to implement. This is because it would be relatively easy 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 [23].

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:

1. 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.
2. I could store the various versions of the data in application memory. This would avoid the issue mentioned above, but would mean that the memory usage for my application would be rather large.

Because I thought storing the data in application memory would be the simplest 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 keep 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.

I then moved onto looking at Dynamo. 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, with Amazon, the vector clocks and not the data itself 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 hand, I was worried that basing my emulator on the behavior of Amazon’s 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 be the worth the risk if I was able to create an application that performed 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.

**Basic Availability**

The second phase of the project was all about implementing Basic Availability, Therefore, I went back to the models identified in the literature/technology review and compared them all to decide upon the best way of approaching the problem. The questions I needed to answer were how to determine what ‘availability’ meant in the context of the system, and how I was going to implement this notion of ‘availability’ using only one server.

I considered several different approaches to basic availability, I started by looking at the approaches taken by the systems I had researched for eventual consistency, such as Amazon’s DynamoDB and CouchDB. I then moved on to looking at approaches taken by alternative databases such as MongoDB, Neo4j and Redis.

I ruled out the approach taken by CouchDB straight away, The reason for this is because the basic availability model employed here requires that all servers have a copy of the database. This is not the case in my system, the replication servers store database updates, and these updates are applied to data from the database when it reaches the replication server. Because of this, I believe that it would be difficult to combine a model based on CouchDB with the eventual consistency model I already have

A better alternative would be the approach taken by MongoDB. As mentioned previously, MongoDB uses a variation on a Master/Slave replication scheme. The servers are divided into replica sets. Each replica set has a primary, and several secondaries. If a primary went down, then the secondaries hold an election to determine the new primary.

I like this approach for several reasons. Firstly, I could simply modify my replication servers to have several child servers implementing this scheme. Secondly, because all communication in an Akka based system is done between objects of ActorRef type, I would need to modify very little of the remainder of the system in order to get this approach to work effectively.

Other databases I mentioned are too similar to approaches I’ve already covered to be considered separately. As an example, Neo4J and Redis are too similar to MongoDB. I did find some useful refinements of the general concepts embodied in MongoDB that I could add to my solution if I chose to. As an example. Neo4j allows you to add writes to slave nodes. These then propagate their data back up to the master, which then propagates this down to all the other slaves.

Another approach I was keen to investigate was that utilized by Amazon’s Dynamo database. This is partly because I used Dynamo as my model for eventual consistency, so I was hoping that the algorithms used for basic availability would be complimentary to what I already implemented. When I looked further into this I found that there are two main concepts to Amazon’s setup. Firstly, there is the ring of nodes, each of the main server nodes has a ring of sub partitions. The data is split between those nodes. Data is sent to one server and is then sent to all others through the application of Amazon’s vector clock consistency algorithm. Amazon also allows specification of the amount of consistency that is needed This includes setting the amount of writes before an update can be deemed as valid.

When it came to deciding which model I will implement, I decided to use elements from several different approaches. I will use the MongoDB approach for the simplicity of its implementation and the fact that I wouldn’t need to modify what I am already doing that much in order to get everything working. I will also incorporate elements from other cloud databases that use similar setups. In particular, I will allow slaves to receive updates. This is partly to allow myself more flexibility in implementation, as it would be difficult for other actors in the system to distinguish between masters and slaves.

Finally, I will borrow from Amaozn’s model in splitting servers up into groups. This will help me build on top of what I already have, and will also contain each update to one group of servers.

**Software processes**

This chapter of the dissertation outlines the major software development tools and processes I used during the development of this project and also gives explanations on why I chose those particular tools over other alternatives.

When I began the practical component of my project I needed to make decisions about how I would actually go about doing the practical work. A set of good software processes is essential so I set this out early on.

I decided against using a waterfall based development methodology. This is because I have never completed a project of this size on my own before, and I am using a large number of technologies, which are unfamiliar to me. Because of this, I did not feel that the waterfall approach would give me enough space to move things around if arose.

I therefore chose to use an agile methodology with elements of Scrum built in. I chose an agile approach so that I could split the work up into manageable increments. This had two major benefits. Firstly, it allowed me to work on each feature in isolation, meaning that I only needed to focus on integrating into the main system at the end of each iteration. Another benefit of using an agile approach is that it meant that I could easily keep an eye on the progress I was making, and make adjustments to deal with this. As an example, during my first iteration, I completed all the work ahead of time, so I was able to plan to complete more work in the second iteration.

Once I made this decision, I created an iteration plan that detailed exactly how I would manage my time in each iteration. Iterations were two weeks long. Originally I had planned to use one-week iterations and have each one self-contained. After my first week working on the project however, it became clear that actually spreading this over two weeks would be beneficial. This helped me for two main reasons: firstly, it allowed me to vary the tasks that I completed each day. Secondly, I could spend more than one or two days completing each task if needed. I then developed a detailed project plan, that detailed exactly which tasks were to be developed as part of what iterations.

Another important consideration I had to take into account was the programming language that I would be using. The first choice that sprang to mind for this project was Java. Java was the first programming language I was ever taught, so I felt that I would be able to get something going pretty quickly in Java. The language can be used in most situations, because it has so many libraries and plugins available for it, I was particularly confident that I could write web services in Java after reading ‘Java Web Services”[24]. On the other hand, however, I was concerned that the project might involve dealing with concurrency, and the model based on threads and locks that Java provides is notoriously hard to get right. I also felt that I would be playing it safe a bit with Java, and I wanted to learn a language that I had never used before.

Another choice I considered was PHP. PHP is ideal for web-based projects like this, because you insert it directly into HTML code, and most web servers support it. I also thought that the fact that the language was easy to deploy would be a bonus. On the other hand, however, the fact that PHP is dynamically typed makes PHP programs harder to debug. To add to this, PHP has no built in support for concurrency. I therefore decided that the negatives of PHP outweigh the positives for this project.

The third language I looked at was Scala. Scala was built on top of the Java ecosystem, so all the benefits of programming in Java were also true of Scala. Another major plus point for the language was that it supported the actor model of concurrency. This involves not sharing any state at all, and communicating between different parts of the system through the use of immutable messages. I could see this approach working well for this sort of system, because it would be easy to translate a real system full of servers communicating with each other to an actor-based system. Another good point about Scala is that it is both functional and object oriented. This means that if I wanted to write safe code for concurrency, Scala would let me do that, but if I wanted to take advantage of object oriented design methodologies, then I could do that as well.

These were not the only tools that I needed. One tool that I found I needed in particular was version control. Although I wasn’t working as part of a team, I did find that it would be useful to keep track of old versions of documents. Another reason for using version control was the fact that it allowed me to ensure my work was regularly backed up to a server to ensure that nothing was lost.

I therefore created an account on Github for this project. This allowed me to perform the functions mentioned above, and also allowed me to easily share progress with my supervisor about my work.

Another tool that I found particularly helpful during my project was Jira. Jira is an issue tracking tool that allowed me to log and monitor the work I was doing as I was doing it.

As well as the Scala programming language, I used two other libraries to complete the project. The first of these was Akka. Akka provides an implementation of the actor model of concurrency that can be used within Scala or Java programs. Actors mainly communicate via passing messages to each other, and do not share any mutable state. Because of this, it is not possible for an Akka program to suffer from bugs that involve race hazards. Another benefit of the Akka system is that it is very well suited to the project, in that it is easy to map a cloud server in a database onto an akka Actor.

The Play framework was another invaluable tool during this project. This was the main supporting library that I used. It provided tools for developing the actual web service itself, as well as parsing the JSON requests needed as input to my application. This allowed me to put the low level details to the back of my mind and focus on the actual task of implementing eventual consistency.

Design was another key element to my project. I used two different design approaches. Firstly, I used UML class diagrams to model the different types of SQL queries that my system could process, so that I could visualize the inheritance relationships between these before I started the coding work.

When designing the overall system architecture however, I realized that the very nature of the system that I produced in the previous chapter involves a great deal of concurrency. This meant that the standard design mechanisms that are normally used to design software systems, are not normally that effective. The Actor model that underpins the Akka framework which I am using to build my system usually guides systems to be thought of as a network of communicating processes. I therefore found that it made sense to use a process network diagram as taught to me in CO890 to model the system.

A process network diagram basically represents a diagram of the various concurrent p in the system and the connections they have to each other. Along with the diagram, text is normally given to explain what each of the various processes do.

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.

I should also probably briefly mention IntelliJ IDE here. While not directly related to the success of my project, it did make things much easier because it allowed me to use all the tools from one place and meant I saved much more time than if I had had to switch between each tool individually.

I had to adjust the algorithms significantly though in order to fit the situation that I was working in. I will discuss this further in the chapter on implementing eventual consistency.

**Implementing Eventual Consistency**

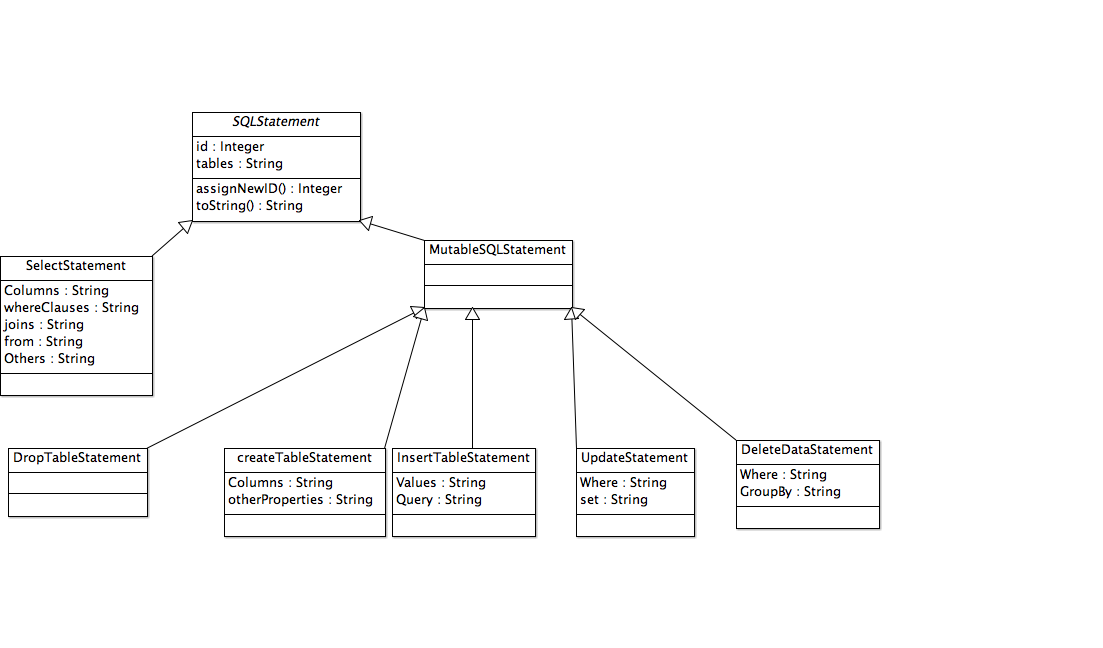
In this chapter, I will explain the work I did for implementing a basic version of the project that only implements eventual consistency.

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 Scala’s 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 also useful in other ways. Firstly, it reaffirmed 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.

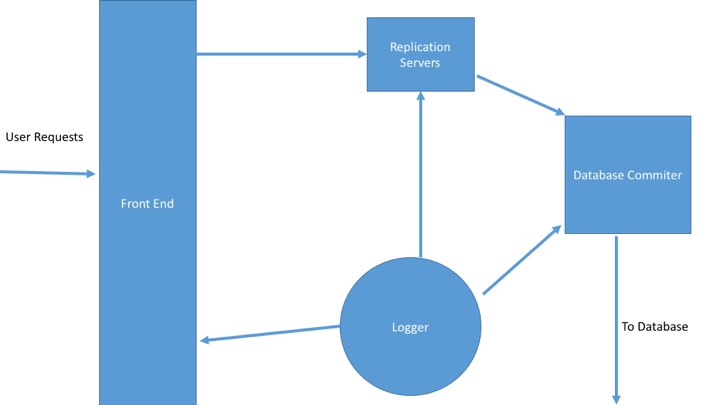
It further made me rethink the way I was planning to do database access. Originally, I had planned to do this using the standard JDBC library for Java. I made this decision based on the fact that this tool was part of standard Java, meaning that it would be likely to be well documented and easy to use. 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 proved 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. The inheritance hierarchy for this is shown below. The great advantage that this approach had was that the vast majority of the code needed to be written only once, which could then work for all of the various types of queries. This hierarchy is depicted below.



Once this was done, I moved into implementing these designs in Scala. It was fairly straightforward to convert from the UML class diagrams to code. I verified that the textual SQL representations of my classes were correct by running the code in the Scala REPL and running it on the database.

Once this was done, I then moved onto the main system design. In my initial versions of the design document I tried to keep the interactions with the system as asynchronous as possible. In other words, the user would be able to send a request to the system, but would then need to call another web service to find out the result of the request on the overall state of the system. This approach was taken partly to ensure that response time for the overall system could be relatively fast. The intent was to avoid the need for users having to wait long periods of time waiting for the system while it finished completing other work. Having parts of my system wait for other parts of my system to finish would also have created additional concurrency challenges, as it was proved in CO890 that a system without cycles (processes sending messages and waiting for a response) is deadlock free. The design I came up with is shown below.



Originally I had four main processes in my system: the frontend, which was a standard Play controller, was the component that responded to viewers’ requests, and checked to see if they were valid, and then sent them onto the rest of the system; the replication servers were the section of the system responsible for managing the eventual consistency within the application; the database committers were responsible for taking all the eventually consistent requests and then passing them onto the database; and finally, the logger was responsible for receiving status messages from the user and then sending them to the frontend so that they could be displayed on user request.

Another issue that I found with the system is that I needed to find a way to measure time in the system. This is because the system is designed to make itself strongly consistent within a set time interval. Originally, I tried using the Java data and time classes. This worked on the outset and was simple to implement, but was imprecise, and I noticed that the consistency runs were either happening seconds too late or too early. After exploring the documentation linked in with the Akka framework, I found that that the framework already provides a built in mechanism to send messages to Actors at specific intervals. I therefore used this to solve the problem in a rather straightforward manner.

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 from the replication server and make them consistent. It would then take this stream of fully consistent requests and pass it onto the waiting database committers, so that these requests could be persisted to the database.

Once I had implemented the eventual consistency algorithms themselves, I also had to implement retrieving information from the database. This section of the application proved more challenging then initially first thought. The main reason for this was that I needed to revisit the rule that I had set myself about not waiting for anything to complete. The reason for this was that in order to retrieve inconsistent results. My application needed to contact the replication servers of the system and then wait for them to reply. In order to account for this problem, I made use of the timer facilities provided by Akka. In order to ensure that the application could not be kept wafting for too long a period.

Once I finished the main implementation of the system, I then went about adding additional pieces of functionality to my implementation. These included the ability to change the period between the whole system being made fully consistent and the ability to see all the inconsistent updates currently in the system. Rather than implementing these as concurrent processes, I implemented these as static backing objects with limited interfaces, so that the frontend could query them without waiting for an other part of the system. This also fitted in with the rule about avoiding blocking as much as I could during the development of the system.

Evaluation

Once I submitted my early deliverable, I then undertook a comprehensive evaluation of the original system. I then was able to make improvement sin several areas.

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, there were also issues with the system receiving multiple copies of repeat messages. I solved these problems by using the state machine built on top of the Akka framework. I implemented a merging state, that allowed for a much cleaner operation of the merging process and a standard state for standard operations of the process.

When I showed the system to my supervisor, she commented don the fact that my system very rarely produced inconsistent results. I therefore added, in several features to help with this side of things. Firstly, I added the ability to show all possible results for a particular request, so that you did not need to query the system multiple times. Secondly. I allowed the number of severs to be customizable. When I changed that to only use two servers, I was able to get more inconsistent results.

The final major challenge I had to overcome with eventual consistency is a bug I found while doing some in depth testing of my system. I noticed I was getting erroneous results when only selecting one or two columns. The reason for this is that sometimes it was impossible to apply certain updates because my system couldn’t tell they were needed due to certain columns being left out of the results. I therefore changed this part of the systems so that the system always retrieved all columns during database updates, and then removed all of those that weren’t relevant.

**Basic Availability**

Once I finished the eventual consistency section of my dissertation. I then needed to turn my attention to Basic availability. I began by deciding on which cloud model I would base this piece of functionality on. Once this was decided, I then began thinking about implementing this in Scala.

Almost immediately, I had to overcome a major stumbling block regarding how the new code would be integrated into the rest of the system. This was difficult because a lot of the components that I had implemented thus far were very specialized components. By this I mean that they were designed to communicate in very specific ways with very specific processes. This meant that on the one hand, I did not want to change too much, because that could potentially mean rewriting large sections of the eventual consistency functionality that I had just implemented. On the other hand, not changing enough and bolting things onto what I already had would result in a system that would be hard to maintain.

I therefore went back to my design and replaced the original replication servers with new replication clusters that would mange groups of original replication servers. These replication clusters would implement the basic availability solution described in the problem analysis chapter. This solution had several advantages. Firstly, it meant that the vast majority of the code that I would be adding to the system would be new code, rather than modifications to existing code. This is because the cluster severs themselves communicated directly with the original replication overseer, whereas the slave replication Servers communicated with the replication marshaller. Rather than creating a whole new actor for the slave nodes, this was impmented by making small modifications to the original replication servers. The main changes I needed to add were that this class now only communicated with the rplication marshaller sif it was the master., and also had the mena sot contact the other slave serves if a master went down. i

Implementing this server was fairly straightforward,. I had the replication clusters send all messages received that they were not irectly interested in to the child servers. The dwonsid e of this is that it meant that many duplicate messages were being sent around the syste. These are mostly harmless however, and I thought that ensuring the functionality of the sytem is working correctly was much more important than ensuring the system was completely quit al of the time. It also kept the oveall complexity of the system down.

The main work that rpplicaton clusters did was the following of the protocls needed to ensure availability. This compised of several small takss, including managing the voting process to elect a new master, and also was responsible for reviving dead masters. In a lot of thd real cloud systems that I studiesd, the slave nodes do this themselves, but I thought that having the cluster overseer did this would be sassier to ensure all nodes in the system had an up to date view of the network.

Whren designing Basic availability, I realized that the number of paramters that could be provided to the system for customization purposes was about to grow dramatically. As part of the eventual consistency iteration > I had implmeneted a mechanism that would allow the user to change the gap between consistency runs. This service was unsatisfactory however, because it was not very extendable. I fi wanted to add any more paramters, I would need to add another web service to manage this. I therefore set about designing another web service to manage this need. It worked by taking the name of a parameter and a value, and changing the parameter to the new value. The impmenentation was very simple, as all it needed was a map to store the relationships between the keys and the values. I also wrote a static method that allowed other applications to look up the user supplied values as needed. This rpoved enormously beeficial for the rest of the project, as it meant that I could add future customizations by simply adding a signle line of code.

A key part of my stategy for emulating basic availability ws that requests could fai, and then providing information on the percentage that succeeded or failed. In order to do this, I needed another actorf to receive and collate reports about sucsesful server acquasitions and failures. I therefore wrote a failure manager system that could reieve success or failure reports from tehe replication servers. I then logged these failures to a singleton object, which was responsible for computing the percentage of sucssess and failures. I decided to seprate these bits fo funcionallity for two main reasons. Firslty, it is good programming practice that each component of a system should be responsible for one main taks. Secondly, if the failureActor dealt with the stats, then it would also need to talk to the UI. This would introduce the same risks of blocking into the system as I would have had with the other customizations.

Implementing this part of the syste was not without its challenges. Implmeneting the failure manager itself didn’t pose too much of a problem, what was tricky however, was that making it accessible to the various parts of the system that needed it. Most system components took a Logger as a parameter, and I considered passing around the failure actor in much the same wa. This idea was abaondoned realitvely quickly however, as it would have mean tthat the faiure actor would need ot be passed throrough several alyers of actors before it reaced the desired location. As an alternative to this therefore, I looked into using the Akka lookup service. The akka lookup service is a mechanism built inot the Akka framework by enabling lookup of an actor by passing in its unique actor name. My solution to this problem was to have all actors that required this actor retrieve it from their code paths at construction time.

Challenges and evaluation

WEhren I first started on this section of the code, one of the things that frequenstly blcked my prograess was Akkas message matching system. This is because ethe vast majority of my knowledge of this style of programming comes from Erlang. Because Akka behaves similarly to Erlang in many respects, I wasted a great deal of time assuming that som of the coding patterns for erlang also work here. As an example, Erlang matches message sbased on the first one it finds that will match. Wheras Akka matchs things in a slightely lesser known and more mysterious way. This mean tthat in some situations I had to have multiple message handlers matching for the same thin. This means that thers a great deal fo code repititiions, which is neve r agood thing in a large software engineering project, but it was unavoidable in this case.

Another issue I had with my intial version of Basic availability is wht when a server went down, the object was removed and garbage collected. This had two downsides, not only did it mean excessive usage of memory, but it also meat that a lot had to be done to keep al the referencs to that sever valid. In otherwords, whenever a server failed and was resotred, a message had to be sent to all other severs in the cluster so that they could update their reference to that server. I fixed the probem by introducing a dead state for actors. In this state, all the actor would do is respond to certain messages signaling its return to tlife, anything else would be discarded. This gave me the est of both worlds, severs would stay dead as I had intended, but would also not need recreating every time one of them came back to life.

Another challenge I faced with basic avilibility was the it was difficult ot debug system effectively. This is becase I used my original replication servers as my internal serveds. This meant that all of the debugging output that I had previously generated was being reproduced several times over. I thereforore found the simplist way of getting around this was giving each server an identity number, so that we coul I could see each of the servers in the message.

The final major challenge that I faced while develpping this aspect of the system were timings. The reasons for this ws that if the wrong parameters were passed into the system, then it is very easy to unintentionally disable other parts of the system. An example of this is that if you set the time to wait before a new master is elected as too soon, and the time faiure rate as 0, then all the system will be doing is groing around creating mosre master slaves. I got around this in two ways. Firslty, in the user interface, I gave advanice on what to set each parameter to, so as they wyouldnt be misued. On the other hand, I also implemented some rudimentary checks inot the actual application code so that some of the more obvious errors could be deal twiht at the schene.

Once I completed the intial basic availibilty, I began to evaluate it to see wh how it worked. The first thing I noticed was that the statistics the application was generating wernt what they were supposed to be. As an example, when I set the application to have a 50% rate of availability,, often the application said that 75% of the requests succeeded and 30% failed. When I explained the problem ot my supervisor however, she suggested that this was because the amount I was of data I wqas using was two small to be statistically significant. As an example, instead of sening three updates to the database, it was suggested I send 100. This helped immensely.

Another rissue that I noticed witht e application sitht sometimes database updates would be lost form the system befor etheh had a chance to be recooverws sucseffully. In order to fix this issue, I had all of the master sterves check for any missed update son the slaves before sednging things through to the database. This helped resolv eth eproblem gratly.

**Testing**

When I first started the p roject, I laid down a comprehensive quality assurance document and Risk assessment to ensure the overall quality of my project was high.This covered everything from static analaysis to user acceptance testing. In this chapter I will discuss my tsting in more detail.

User acceptance testing was useful throughout the project. Firslty, it allowed me to measure my progress to both me and my supervisor. I loaded these tests into my Issue tracking system, and could then compare my actual progress vs my expected progress to ensure that I was on the right track.

Now I will look at testing the application. Firstly, I used unit tests to test each of the core pieces of functionality. This form of testing served two purposes. Firstly, it allowed me to verify that each component was working as I intended it to as I developed the system. Secondly, it also meant that I could assess rapidly whether that piece of software broke the rest of the system by running all the unit tests that I previously created.

I used the Spec2 framework provided with Play to do this. The main reason for this was that it allowed me to test the actual http requests in my code. This turned out to be incredibly useful, as otherwise I would have had to do this manually, something that would have taken a great deal more time and effort. I was able to integrate my unit tests whenever I rebuilt the system, so that I always had a good idea of what I’d broken whenever I changed a version of the software.

Another valuable tool that I made heavy use of during my project was static analysis. This was useful because it caught lots of stupid typos before I even ran the code. A prime example of this kind of situation was a time when I wrote a Boolean function that always returned false. I ran my project under several different static analysis tools each time I compiled my project, so as to catch the most errors possible.

One thing that was incredibly difficult to do during the course of the project was debug effectively. The reason for this is because of the massive concurrency involved in the system. This had a serious impatc because if I paused the debugger at a particular point in my code, the rest of the application would keep on running, . This mean that it was very difficult to test the application as it was running, and instead I had to rely mainly on my unit tests for debugging purpsoes, as these allowed me to run individual components of the system in isolation without fear of them getting messed up by the rest of the system.

Because I was using the Akka concurrency framework in order to make my application scalable I could not use the standard unit tests provided by Play framework to test all aspects of my application. This is because the actors in play are protected by the special ActorRef class, meaning that you can only send and receive messages to them. The Akka testKit enables you to get access to the underlying actors methods, meaning I could make full use of Spec2 to enable correct testing of my application.

This also made integration testing difficult for the same reason., In order to get around this problem, I would design manual integration tests that I would preform by hand. I would then write test plans for these and record what happens in there. While this is not perfect, and it did not allow me to ensure that all the elemet of the system had ben tested. It did allow me to ensure th all of the compeonets were doing as they were supposed to.

After every iteration I re ran all my UAT and unit tests again. This is commonly known as regression testing. This allowed me to check that any of the new functionality that I had added had broken any of the previous functionality. IF any of these tets now failed, then I could go through and fix the bug before it caused any major damage to the rest of the system.

**Running on A real Cloud platform**

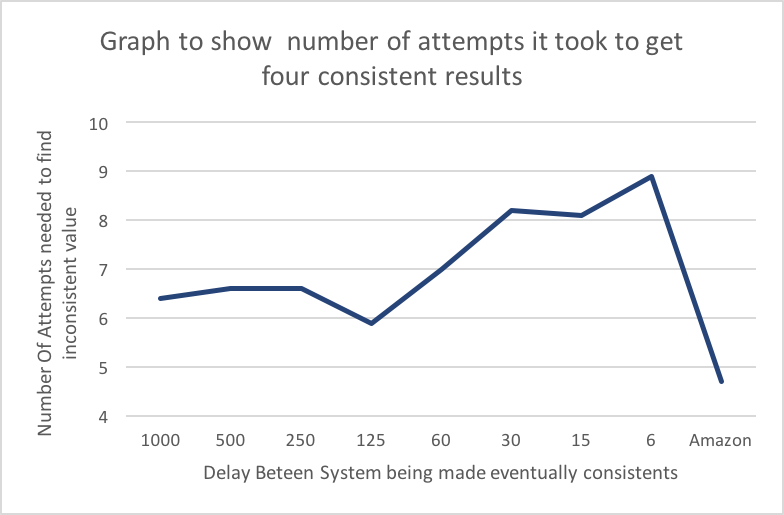
Once I Had completed the amin implementation work of the project, I wanted to test my work ona real cloud system. To do this, I would run various experiements on moth my system and a real cloud platform. I chose Amazon to run these experiements on. This was mainly due to two reasons. Firslty, it is known to be eventually consistent, which means that It should provide a good comparison tomy own system. Also based my system on it,. I decided not to do basic availiblity in the same way., due to the fact that, it is very difficult ot make an amazon system fail, so there would not be any real comparisons.

My first port of call was to design a few sample programs that I could run on both Amazon and on my own system. I based all of these examples around updating one field in the database. SO as to minimize code duplication, all database access was done through an object implementing the following trait.

*/\*\*  
 \* simple trait to  
 \* manage connections ot the database,  
 \* this is here so that my samples  
 \* can make use of it, whther they be from anazon or my own system  
 \** ***@author*** *Jack Davey  
 \** ***@version*** *3rd August 2015  
 \*/***trait** DatabaseConnector  
{  
 */\*\*  
 \** ***@return*** *a value from the underlying database where the name is  
 \* "dad"  
 \*/* **def** read():Int  
  
 */\*\*  
 \* writes a value to the record whose name is  
 \* "dad in the database  
 \** ***@param age*** *the new age  
 \*/* **def** write(age:Int)  
  
}

The first property I wanted to test was the time it takes to get an inconsistent value. This program operated by reapeatedly writing values to the database, and making a read immediately afterward, if the read did not match the number just written, then a counter was incremented. Once the counter reached 4. Then the number of attempts taken to achieve this goal was reported back to the user.

Before I ran this experiment, I made a hypothesis of what to expect. I thought that it would take longer to get inconsistent values the shorter the gaps between consistency runs were.



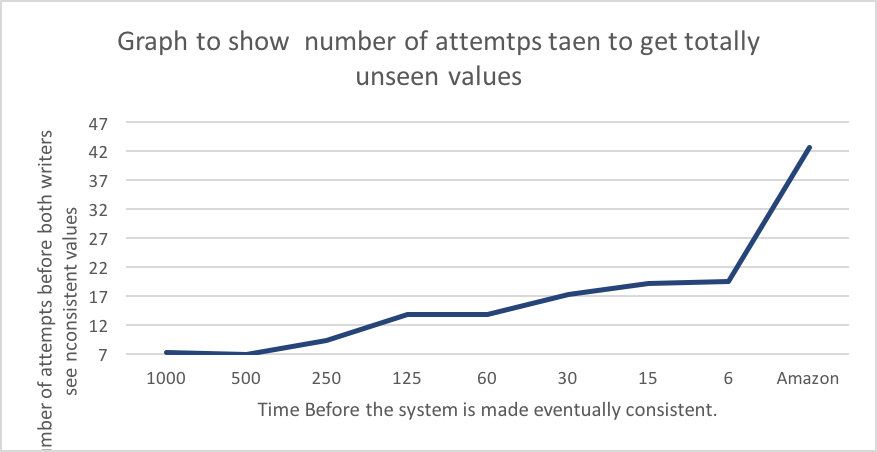
The graph above shows that I was correct in predicting this observation. The strange thing that I noticed whoever was the fact that although shorter times before the system was made consistent did affect things, the rest of the time, the system remained very fast.

When I ran the same experiement on Amazon The system outperformed my system significatny. Takin gon average 4 attempts to retrieve an inconsistent value, wheras the lowest my system achieved was 6 attempts. Upon further reflection however, this is to be expected, Amazons Dynamo database uses many replicas spread over many servers worldwide, wheras my system is limited to one computer. Amazon also begins to make data consistent immedialely after landing on the server. Therefore, it is nonly natuarual that their system would yield better results than mine.

I believe the reason as to why these results are the way they are is because longer the period is between stale data being removed, the more there is to clod up the system. This means that on higher time values, it should be relatively easy to find stale data. On the other hand, when the stale data is being removed more regularly, then the suer is more likely to receive the latest values

.

I then wrote a second program using two writer objects. This measured the numb rof attempt taken for both writers to see values that they had never written before. The graph is below.



As with the single writer example, the results for my own system are unsurprising. The longer the time given for eventual consistency, the more inconsistent results can be found. I believe that this is due to the same reasons mentioned above.

The thin Is suprpising about this experiment however was the results gleaned from running the same program on amazon. Rather than outprfroming my system, Amazon took much longer than my program to find the inconsistent results. I believe that several factors contribute to this.

Firstly, because eamazon starts making its data fuly consistent whenever a new result hits the database it s mh harde rot get two inconsistent request. Findind one inconsistent request seems to be fairly easy, but finding two seems to be a lot harder.

**Evaluation**

It is also worth noting however, that this plan did not always run smoothly every week. As an example, my work for the second iteration (which involved getting a basic version of eventual consistency working in my project) took much longer than I had expected it to, and therefore ended up spilling into the time that should have been used to develop iteration three. In contrast to this however, some iterations, such as iteration one, took much less time than I expected, so everything evened itself out in the end.

References

1. 1. eBay and Pritchett, D. (2008) *BASE: An Acid Alternative - ACM Queue*. Available at: http://queue.acm.org/detail.cfm?id=1394128 (Accessed: 30 March 2015)
2. Tanenbaum, A. S. and van Steen, M. (no date) *Distributed SystemsPrinciples and Paradigms*. India: Prentice-Hall of India Pvt.Ltd
3. Celko, J. (2013) *Joe Celko’s Complete Guide to Nosql: What Every SQL Professional Needs to Know about Nonrelational Databases*. United States: Morgan Kaufmann
4. Kuznetsov, S. D. and Poskonin, A. V. (2014) ‘NoSQL data management systems’, *Programming and Computer Software*, 40(6), pp. 323–332. doi: 10.1134/S0361768814060152
5. Sake, S. (2013) ‘Cloud-hosted databases: technologies, challenges and opportunities’,
6. Furht, B., Escalante, A. and editors. (2010) *Handbook of Cloud Computing*. Edited by Borko Furht and Armando Escalante. Boston, MA: Springer Science+Business Media
7. Edlich, P. D. S. *NOSQL Databases*. Available at: http://nosql-database.org (Accessed: 2 April 2015)
8. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W. (2007) ‘Dynamo: Amazon’s Highly Available Key-value Store’, *ACM SIGOPS Operating Systems Review*, 41(6),
9. Redmond, E. and Wilson, J. R. (2012) *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*. Lewisville, TX: The Pragmatic Programmers
10. Anderson, C. J., Lehnardt, J. and Slater, N. (2010) *CouchDB The Definitive Guide*. Sebastopol, CA: O’Reilly Media
11. Ghemawat, S., Gobioff, H. and Leung, S.-T. (2003) ‘The Google file system’, *ACM SIGOPS Operating Systems Review*, 37(5), doi: 10.1145/1165389.945450
12. Burckhardt, S., Leijen, D., Fähndrich, M. and Sagiv, M. (2012) ‘Eventually Consistent Transactions’, *Programming Languages and Systems*, pp. 67–86. doi: 10.1007/978-3-642-28869-2\_4
13. Horstmann, C. S. (2012) *Scala for the impatient*. 1st edn. United States: Addison-Wesley Educational Publishers Inc
14. Wyatt, D. K. (2013) *AKKA Concurrency*. Canada: Artima Inc
15. Sommerville, I. (2011) *Software Engineering*. 9th edn. Harlow: Pearson Education (US)
16. Hilton, P., Bakker, E. and Canedo, F. (2013) *Play for Scala:*. United States: Manning Publications
17. https (no date) *Sharding Introduction — MongoDB Manual 3.0.4*. Available at: http://docs.mongodb.org/manual/core/sharding-introduction/ (Accessed: 25 June 2015)
18. *Neo4j, the World’s Leading Graph Database* (no date) Available at: http://neo4j.com (Accessed: 25 June 2015)
19. *Sharding and replication* (no date) Available at: http://rethinkdb.com/docs/sharding-and-replication/ (Accessed: 1 April 2015)
20. *Partitioning: how to split data among multiple Redis instances. – Redis* Available at: http://redis.io/topics/partitioning (Accessed: 25 June 2015)
21. Google Cloud DataStore Available at <https://cloud.google.com/datastore/docs/concepts/overview> (Accessed: 25 June 2015)
22. Anazon Web Services Availible at: <http://aws.amazon.com> (Accessed: 25 June 2015)
23. Evans, B. and Warburton, R.*Java SE 8 Date and Time*. Available at: http://www.oracle.com/technetwork/articles/java/jf14-date-time-2125367.html (Accessed: 24 June 2015)
24. Kalin, M. (2009) *Java Web Services: Up and Running*. United States: O’Reilly Media, Incorporated