Jack Davey

**Dissertation**

**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 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 tractions. The first of these is basic availability. This property states that the system should never go down, even when one part of the system fails. Another property of most BASE systems is soft-state. This means that the consistency rules on BASE backed databases are more lax than their ACID counterparts. There is a catch however, and this is that a BASE database must be guaranteed to be consistent at some point in the future, which is the third BASE property of eventual consistency.

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, they are losing money.

The downside with BASE transactions is that because they are unpredictable, it makes it harder to perform good quality research experiments on them without spending a lot of money on a well known Cloud platform. 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 systems by writing some sample programs.

**Technology Review**

When I started this project, I didn’t have a clear picture in my heads of where to go to find more information. My project supervisor pointed me in the direction of several 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. A book on distributed systems, *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 for 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 propertied 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.

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 rovided 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. It is also worth noting that the algorithms used by CouchDB were so effective that variants of them have been produced, such as in *CouchDB The Definitive Guide*. [11].

Another approach that I considered was Google’s model of achieving eventual consistency[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 as well. 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.

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 getting a solid grasp of the Scala programming language down. 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. I also found the book *Play for Scala*[16] useful for learning it.

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 back from each other 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 used a sharding scheme similar to MongoDB as well[19]. I found it interesting that several of the other cloud databases used this approach, and started to question why this is. I wonder 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 preformed 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.

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 systems[22]. The reason for this was that I am loosely basing the eventual consistency implementation on Amazon’s Dynamo Database. So I thought that it would be a good point of comparison. Originally, I started looking at the S3 service but changed to using Dynamo itself, because then I would be comparing my application against the real thing.

**Problem Analysis**

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. 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 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[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. 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 of data in 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 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.

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 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 hand 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 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. 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.

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.

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.

The reason as to why I like this approach is due to several factors. Firstly, I could simply modify my replication servers to have several child servers implementing this scheme. 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 as a model of their own right. 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 go down that route. 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 the data down to all the other slaves.

Another approach I was keen to investigate was that utilized by Amazons 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 Amazons 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 Amazons vector clock consistency model. Amazon also allows specification of the amount of consistency that can be specified. 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 would use Amazon’s model o using hashcodes to assign servers as well as the various customization oprations supported by their system. This I so that I can make the customization of the user experience based on real cloud systems as wella s just the implementation. It will also allow me to formalize what it means for the master slve relationship for a write or read to be sucsesfull.

**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. I believe that a set of good software processes is essential for doing this 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 difficulties hit my project.

I therefore chose the 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 too quickly, so I was able to plan to complete more work in the second week.

Once I made this decision, I created an iteration plan that detailed exactly how I would manage my time in each iteration. 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, it meant that I could give each task more than one or two days to do.

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.

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 mine. This is 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 major selling point. 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. The reason being that 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 was equally able to 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 reasons as to why I chose 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. It also allowed me to provide weekly progress updates to my project supervisor, as I could indicate in a graphical way exactly how much work I had done.

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 mode 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 plus point 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 in my project. 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. There were two main kinds of UML diagrams I needed here. 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 syste,m usually guides systems to be thought of as a network fo communicating processes. I therefore found that it made sense d 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.

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.

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.

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.

**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 useful in other ways however. 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 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 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 once, which could then work fro all of the various types of queries.

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 a 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 additionall concurrency challenges, as it ws proved in CO890 that a system tht has no ‘cycles”, where once process is wating for another to be deadlock free.

Origainally I had four main processes in my system, the frontend, wihc was a standard Play controller, was the compoenent that responeded to viewrs requests, checked to se 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.

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 onot 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. Th reason for this was that in order to retrieve inconsistent results. My application needed to to cntact the replication servers fo 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 wating for too long a period.

Once I finished the main implementation of th system, I then went about adding additiaonl 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 waitng for an other part of the system. This

Evaluation

Once I submitted my early deliverable, I then undertook a comprehensive evaluation of the original system. I then was able to maeke 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 waere also issues with the system receiving multiple copies of repeat messages. I solved these problems by using the state machie built on top of the akka framwrok . I implemented a merging state, that alloed for a much lceaner operation of the merging process and a standard state for standard operations of the process.

Another issue that I identified with this iteration simy project planning, sometimes I would often allocate too little time to a particular task, and this would mean I would end up falling behind. Other weeks however, I would end up spending too much time doing ap aprticualr task, so thing would even themselves out in the end.

When I shoed the system to my supervisor, she commente don the fac that my system very rearely produced inconsistent reusslts. I therefore added,

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