Forbes Lindesay
Robinson College
Computer Science Tripos, Part II, 2013

Wrench – A Distributed Key-Value Store with Strong Consistency

16/05/2013

Proforma

Name: Forbes Lindesay

College: Robinson College

Title: Wrench - A Distributed Key-Value Store with Strong Consistency

Examination Year: 2013

Word Count: 8, 500 (approx.)

Project Originator: *Forbes Lindesay*

Project Supervisors: *Malte Schwarzkopf* and *Prof. Jean Bacon*

Original Aims:

The original aim was to create a fully distributed key-value store that used Paxos to combine high availability with strong consistency.

Work Completed:

A distributed key-value store was built with a model for transactions that supports consistency via optimistic concurrency for write transactions and snapshot isolation for read transactions. It uses my own implementation of Paxos to serialize the transactions and distribute the status of commit or abort. It supports changing the number of nodes at runtime by allowing for nodes to be replaced completely transparently and allowing new nodes to have a different configuration for the number of nodes in the system.

Special Difficulties:

None

Declaration of Originality:

I Forbes Lindesay of Robinson College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extend for a comparable purpose.

Signed

Date

Table of Contents

Appendices

1 Introduction

Many modern applications require extremely high availability even in the face of multiple simultaneous failures. There are systems that provide this but currently most of them rely on eventual consistency. CAP theorem (Brewer, 2000) states that it is impossible to build a system with consistency, availability and partition tolerance. Using eventual consistency makes it relatively easy to provide high availability because one can just read and write to whatever node of a distributed system one can contact, even if that node is not fully up to date.

Writing software that only uses eventually consistent storage does, however, require more care as the application logic must handle potential inconsistency. Many applications, such as banking systems, may require very strong guarantees of consistency. This project provides a distributed storage system that gives both high availability and strong consistency. It is a fully distributed key-value store and it is possible to read and write to any node at any time.

From the point of view of CAP theorem, the system's reliance on Paxos means that it has consistency and partition tolerance. It also has availability though, providing the node that is hit is in the majority partition. That is sufficient to mean that in most practical environments it has high availability.

Although the system is written in C# and the client I wrote is also written in C#, it would be simple to create a client in almost any programming language as the core interface is just a simple JSON protocol. The broad structure of the system can be seen in Figure 1.

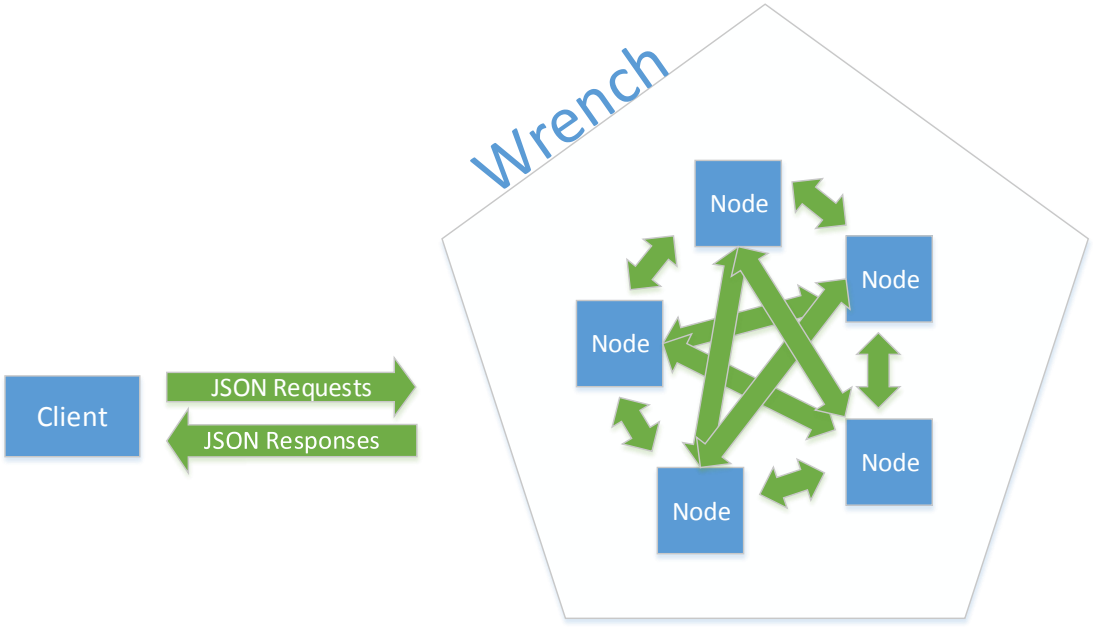

Figure 1: Diagram of the system, showing the networking between nodes and between clients and the system.

2 Preparation

Undertaking a project of this size necessitates careful planning. The necessary resources need to be identified in advance and a strategy to minimise risks from equipment failure needs to be established.

This chapter starts by outlining the basic principles in building distributed databases and consensus systems. It then explains the approach I took with my project, as well as the decisions on programming languages and tooling used in its realisation.

2.1 Distributed Databases

2.1.1 Requirement for Agreement

Transactions are often used to isolate updates in concurrent systems and provide ACID[1] semantics. In the following I will assume a system based on transactions.

Creating a distributed database requires agreement on a range of matters related to concurrency of access. It is essential that all nodes agree:

1. The order in which the updates were committed (see Figure 2).
2. If we use transactions to manage ordering and provide ACID semantics, we need to agree whether each transaction committed.
3. If we are using locking then every node must agree on who holds which lock.

The ordering constraint is demonstrated for transactions in Figure 2. In order to know what value of "A" gets read at the end, we must know the order in which two transactions were committed.

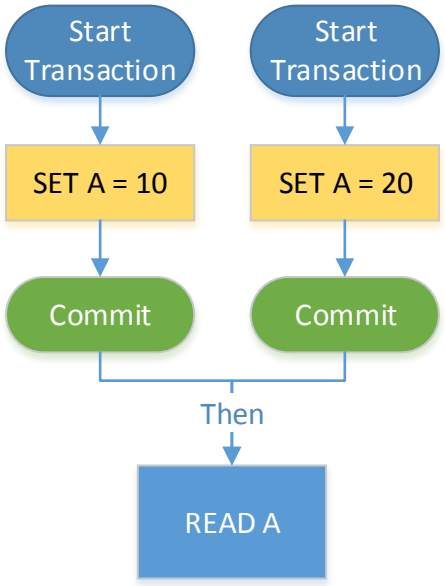

Figure 2: Concurrent transactions have to be serialized

This ordering cannot always be established simply using the time of day. In a distributed system, we cannot rely on all the components having the same notion of the current time as clocks will drift.

The ordering must also be decided upon before we establish whether two transactions conflict with each other. If one transaction reads a value that another writes they may

[1] ACID: Atomicity, Consistency, Isolation and Durability

conflict. They can only conflict if they overlap (i.e. the later one starts before the earlier one commits). If the first transaction writes to a key that the second transaction reads, and the second transaction did not see the first transaction's write, then there is a conflict. This means that the ordering of transactions must be decided upon before deciding whether they are committed.

2.1.2 One Master

In most traditional distributed database systems, a single master approves or denies changes and decides upon their ordering. That master then distributes its decisions to the other nodes in the database. The transactions are either fully serialized, or some kind of roll-back or locking mechanism is used. This creates problems when the master fails. I started my project by looking for a way to automatically and transparently handle the failure of a master.

One option is to have the nodes automatically "elect" a master. This is the approach I took initially. The master is elected for a *lease period* and must be re-elected before its lease expires. If the lease expires and the master has not been re-elected, another node can attempt to become elected master and take over. The election of a master generalises well to the idea of acquiring locks: if we take a lock server, all a master election system represents is a system where everyone attempts to acquire the same lock, and whoever gets it is elected master. The exact mechanisms for the election/locking are described in detail in Appendix A since, although I did implement master election, it was not ultimately used in the project.

There are two key problems with just doing master election and then treating the system as a traditional database: the first problem is that of choosing a *lease period*. In the worst case, we would have to wait for twice the length of the lease period with no master (which means the database would be read-only). This is because, in order to ensure a consistent master under normal operation, we must allow the current master to attempt to be re-elected for the next lease period *during* the current lease period. This means the master could acquire two lease periods and then immediately die. If the lease periods are long, this means that the system will be read-only for a long time. If we choose a short lease period, we can end up with a very large amount of network traffic due to frequent elections. We also need to allow for clock drift. Without allowing for clock drift we could potentially get two masters simultaneously as their notion of their lease periods would overlap. This means that, when one master takes over from another master, there is always going to be a gap with no master. In the best case, the gap is the maximum time drift between the two nodes. This is because the outgoing master will assume it still has the lease until its clock shows that its lease has expired. If the incoming master were to start its lease immediately when its clock showed the lease as starting then the two leases could overlap if the clocks had drifted. The only way to ensure there can never be two masters is to create a period of time for the handover to allow for clock drift (or send an explicit handover message, but

that does not deal with unexpected failures). The effect of clock drift is schematically represented in Figure 3.

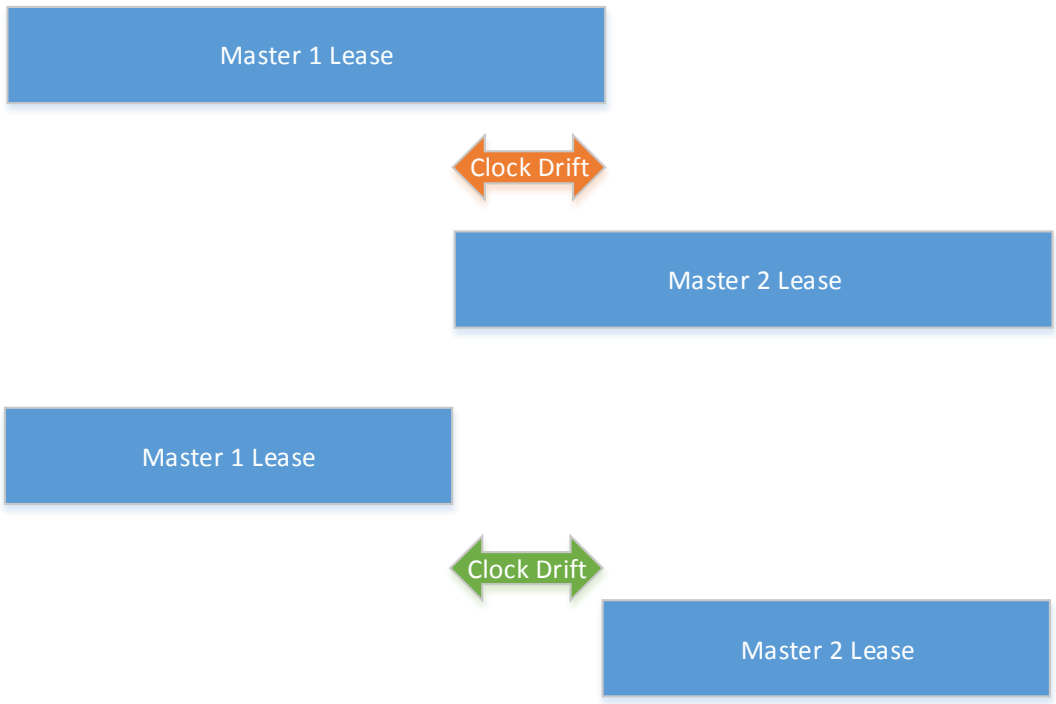

Figure 3: Master Election Handover

The second problem is that we still have one really difficult case to consider. What happens when a master dies part way through distributing the results of a transaction? It might have already distributed the results of a committed transaction to one node when it dies. That node sees the transaction as permanently committed, unless there is a roll-back mechanism.

Using roll-backs on already committed transactions is undesirable because we are unable to notify the user that their transaction has successfully completed because the transaction may be rolled back in the future.

What happens if a different node is elected master? If this new node does not find out that the transaction has been committed, it takes a worst-case view and aborts the transaction. It might not know about the commit, because the nodes that do know the transaction was committed might become temporarily unavailable, say from a network outage. Now there is a problem because some nodes view the transaction as committed and other nodes view it as aborted. Since both are considered permanent, there is nothing that can be done to rectify the problem.

In attempting to solve this second problem, I realised that I was building a solution that would maintain strong consistency even if multiple masters were elected simultaneously. Locking could still potentially be useful to increase throughput by reducing the number of aborted transactions but assuming transactions are allowed to fail when there are conflicts;

we do not need a solution that guarantees total exclusivity, just a solution that will provide exclusivity most of the time.

2.1.3 Master-Less System

Once I established that a single master was not sufficient to solve all problems, I attempted a solution where every node is an equal peer. To achieve this, I needed nodes to be able to agree on what order updates were made in.

This problem of agreeing on a value in a distributed system is commonly known as consensus. There are a number of different protocols for achieving consensus (as discussed in 2.2). I ultimately settled on the Paxos algorithm (Lamport, 2001) as it never fails to reach consensus (assuming that at least half the nodes eventually start receiving messages in a timely fashion).

There are three common ways in which a node could fail:

1. *Fail-Stop* where a node fails and never sends or receives any more messages after that,
2. *Fail-Recover* where a node temporarily fails, but then attempts to carry on with the protocol where it left off, and
3. *Byzantine failure* where a node deviates from the protocol in completely arbitrary ways, potentially sending any messages in any order.

I have chosen not to attempt to handle Byzantine failures since dealing with them is still an active research topic and they are ultimately not very common in practice for this type of application. I do handle both Fail-Stop and Fail-Recover. For these failures, I ensure *safety* (i.e. if a value is agreed upon, every node agrees on it) regardless of how many nodes experience failure. I also ensure *termination* providing strictly fewer than half the nodes have failed and the network between live nodes is fast (messages take less than 1 second to be transmitted and received). The requirement for a fast network is important because Paxos relies on being able to detect failed messages by timing out. My implementation therefore ignores messages that are delayed by more than the timeout.

2.2 Consensus Protocols

In this section, I discuss a number of consensus protocols considered for the project in detail, and elaborate on my decision to implement the Paxos algorithm.

2.2.1 Two-Phase Commit

The first consensus protocol I considered is two-phase commit. For this protocol, we call the node that initiates the round a *coordinator* and all the other nodes *participants*. There is no need to explicitly elect a coordinator and any node can act as the coordinator for any round. This means any node can attempt to gain consensus at any time.

In normal operation (see Figure 4), the *coordinator* proposes a value. Then the *participants* vote either *yes* to accept or *no* to reject the value. If every participant votes *yes*, the coordinator commits to that value and notifies every participant; otherwise it aborts and notifies every participant.

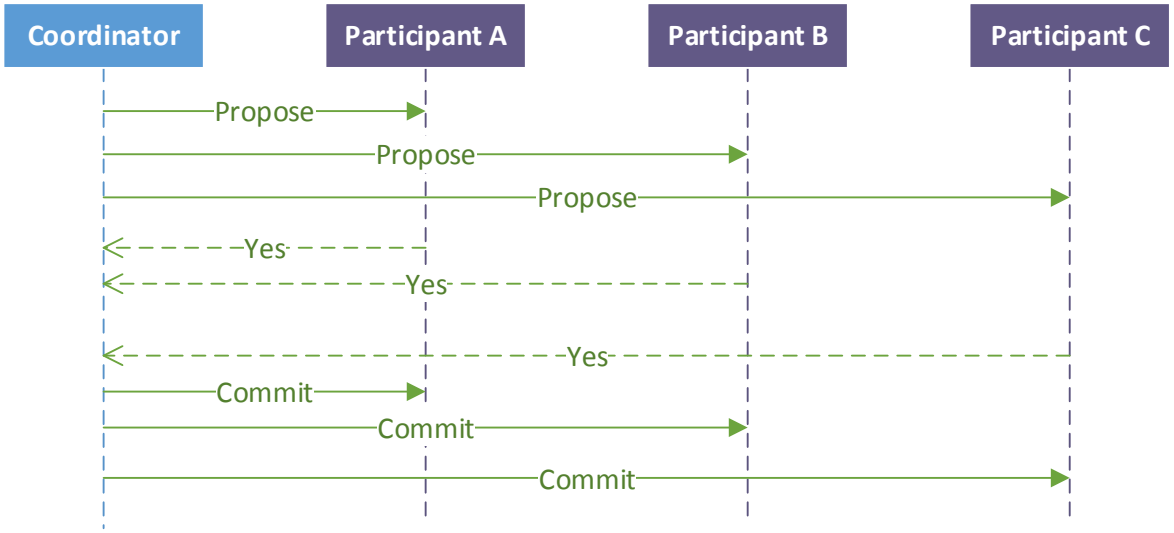

Figure 4: Two-Phase Commit: Normal operation

The next aspect to consider is the possible failure modes. In order to show that the protocol fails, we just need to establish that there is one case in which it either fails to terminate or in which two different outcomes can both be "agreed" to. Consider the case where the coordinator fails after receiving the votes: in the straightforward protocol as described above, this leads to non-termination, and we are stuck.

We can fix this by allowing a second node to take over when the coordinator fails (see Figure 5). It can ask all the participants how they voted. It can then complete the protocol by telling all the participants to commit or abort.

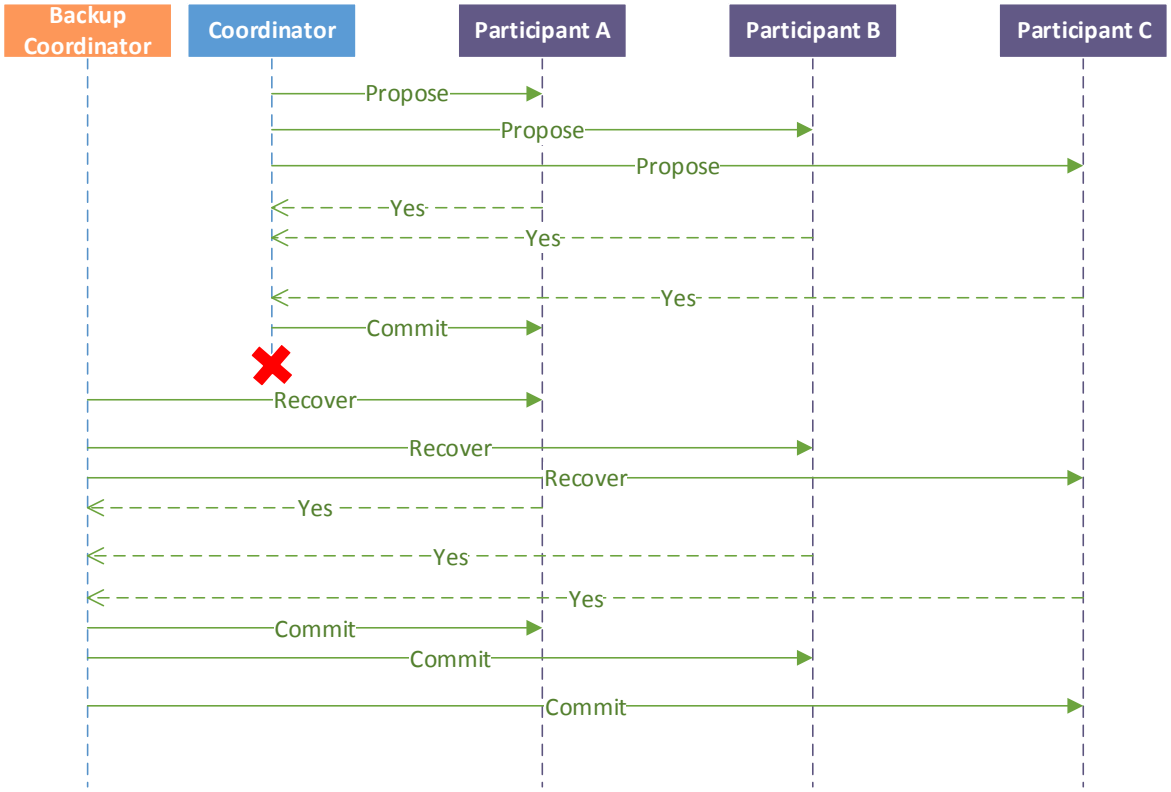

Figure 5: Two-Phase Commit: Recovery

What happens if one of the participants fails too (see Figure 6)? Now the coordinator cannot tell whether (a) all the participants voted to commit and then that participant received a commit message from the original coordinator, or whether (b) that node voted to abort. In this instance neither path would be safe, so the protocol cannot possibly terminate.

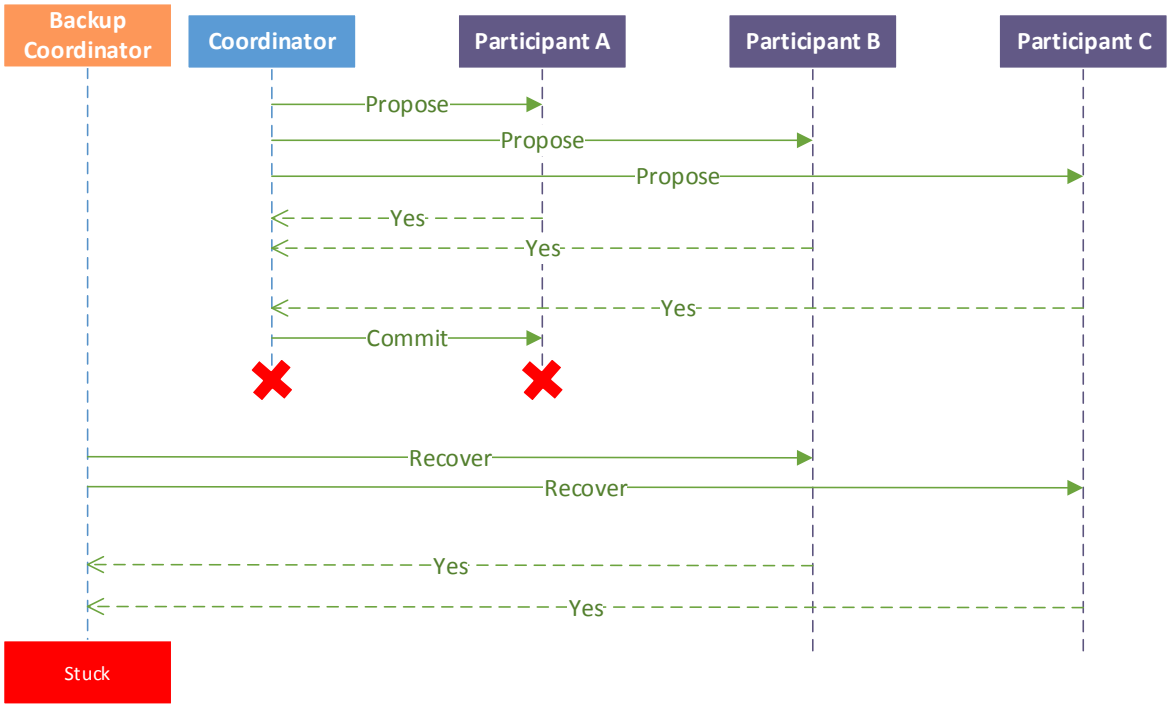

Figure 6: Two Phase Commit Correlated Failure

This means that if two nodes fail and one of them is the master, a two-phase commit can never safely terminate. It is also worth noting that if one of the participants was on the same node as the coordinator, a single failure would cause the system to never terminate. The system would be of no use if failure of two nodes could take the entire system offline, so I ruled out two-phase commit.

2.2.2 Three-Phase Commit

Three-phase commit attempts to solve the liveness problem of two-phase commit by splitting the final commit phase in two, calling the first (additional) part of the commit phase "prepare to commit".

This works by distributing the results of the transaction to one more than half (the number we allow to fail) of the nodes in the "prepare to commit" phase. That way a node which takes over can always query for the resulting state of the protocol by contacting one of those nodes. The issue with this occurs when there is a network partition. The two partitions of the network can arrive at different results: if all the nodes that received a "prepare to commit" message were in the same half of the network, then one half will commit while the other half aborts. Attempts to fix this would result in the protocol not terminating under similar conditions.

2.2.3 Paxos

The Paxos consensus protocol solves all these problems. The algorithm is described in detail in the implementation chapter. The first key insight is that Paxos does not view consensus as a binary decision to commit or abort, but rather views it as choosing a value. A value will

always be decided upon once the protocol terminates; the question is just what that value will be. The second key insight is that it is fine for multiple masters/coordinators (*proposers* in Paxos terminology) to commit values, as long as they commit the same value. By making sure the proposers see any previously committed values, we can make sure they commit the same values.

Paxos will handle any failure except Byzantine failure for up to strictly fewer than half of the nodes. This means that once we have three nodes, we have a system that tolerates the failure of one node, and for twenty nodes we could tolerate up to nine failures. It is the way that the tolerance to failure scales that makes Paxos an ideal choice for my system.

2.3 Requirements Analysis

The essential goals of the system had to be established early on in order to direct development. It was also important to establish how the system could be broken up into modules before proceeding to write the code.

The planned system is a distributed key value store with strong consistency. The user creates the store by simply specifying how many nodes there are, then creating each node and connecting them together in a simple broadcast network. That is, every message sent by a storage node should be sent to every other storage node.

A client should then be created that has methods to create a read transaction and methods to create a write transaction. It should just need to be able to send requests to and receive the responses from a storage node. It should not need to be consistently talking to the same node. The read transaction should just have a `get` method to retrieve a value from a key. The write transaction should have `get`, `set` and `commit` as methods.

As an extension, some method of locking would be useful as it may provide better performance by ensuring that transactions rarely conflict even under high contention. If purely optimistic concurrency is used, then any two concurrent transactions which conflict will always result in one transaction failing. Locking can improve this by just forcing one transaction to wait for the other one to finish.

2.3.1 Modules

The system breaks down roughly into the following modules:

- **Paxos** – provides a consensus algorithm via a multi-round Paxos implementation.
- **Storage Node** – Actually stores the data, sequencing transactions and committing them.
- **Storage Connection** – Classes to allow a Storage Client to connect to a Storage Node via a network.
- **Storage Client** – Provides a simple API to interact with the store.

2.4 Implementation Approach

I have implemented mock networking transports for testing purposes. These allow me to arbitrarily delay or drop packets to help simulate different failure conditions.

However, in order to limit the scope of the project, I decided to leave the implementation of a production transport for broadcast network up to the user. They must also handle the following constraints when constructing such a network:

1. *All-or-nothing delivery* – Every message is either delivered in its entirety, or dropped.
2. *At-most-once delivery* – Messages are never duplicated by the network. In practice this would require some sort of de-duplication by the receiver by storing say a hash of all the messages that have been received, or some other de-duplication mechanism such as sequence numbers.

It is acceptable for the broadcast network to arbitrarily delay or drop packets and no problems should arise as a result. Notably, the above assumptions map well to widely used network protocols such as TCP, which provides reliable and de-duplicated delivery.

The transport for the network between Storage Client and Storage Node is also left largely up to the user to create. It should just be a simple request-response network, e.g. an HTTP connection over TCP/IP. It may be useful to include some retry logic for this network.

2.5 Choice of Tools

2.5.1 Programming Languages

I chose C# as the primary language for implementing the project. This choice was made for two main reasons:

1. C# is a strongly typed, high level language. I felt strong typing and a good system for writing structured code via the use of classes and object orientated programming would add to the maintainability of the code.
2. C# has a good system for writing asynchronous code in an easily readable way. Using tasks to represent all asynchronous operations, along with the new `await` keyword allowed me to write the algorithms in code with similar structure to their pseudo-code even with the added asynchrony.

I also chose to use JavaScript Object Notation (JSON) strings for all networking operations since these provide the desired flexibility to support many network transports. Strings are easy to send over any networking link, and JSON would make it easy to write a client for the system in a language other than C# in the future.

2.5.2 Libraries

I initially began implementing the project using NAct (an “actors” library) to help handle threading and asynchrony. The idea of NAct is to remove the vast majority of threading issues by giving each object its own thread and making all operations within that object run

on the same thread. This would have freed me from worrying about locking. In the end this approach had to be abandoned because NAct has several serious bugs around its support for error handling using exceptions. Since many of the algorithms I have used require appropriate handling of errors and retries to continue in the worst cases, they would have been very difficult to code in an environment that did not work properly with exceptions.

I considered a number of different libraries for JSON serialization, but ultimately settled on `JavaScriptSerializer` from `System.Web.Extensions` in the .NET framework. I chose this library class since it seemed to be the simplest and best maintained option.

2.5.3 Other Tools

The development was carried out on my personal desktop, running Windows 7 and later Windows 8. I had a laptop available that I could use instead in the event that the desktop failed, but its use was not required in the end. Additional cloud resources were used, primarily for backup purposes (see section 2.5.3.2 and 2.5.3.3).

2.5.3.1 Integrated Development Environments

I used Visual Studio 2012 for the majority of my development work. It provides a good built-in text editor and the addition of autocomplete significantly speeds up development. Visual Studio 2012 also provides an integrated unit test runner, which made it simple to run tests frequently.

Although Visual Studio does provide integration with version control systems, I found this feature to be bug-ridden, so I did not make use of it. I opted instead to use a separate application to interface with my version control system.

I also used LINQ Pad[2] to try out small snippets of C# to establish how they would work. This was especially useful when testing out the JSON serializers.

2.5.3.2 Version Control

I used the git version control system. This was crucially important to allow me to view previous versions of the code. It also allowed me to use a Modify-Test-Commit development methodology. I chose git over other version control systems partly because it makes it easy to continue working even when offline (or on a slow internet connection), but largely because of the existing infrastructure around GitHub[3]. GitHub's repositories provided backup, as well as additional convenient tools for viewing past versions.

This dissertation was stored in Dropbox[4] whilst being worked on, and was kept separate from the source code.

[2] http://www.linqpad.net/
[3] https://github.com/
[4] https://www.dropbox.com/

2.5.3.3 Backup Strategy

The backup strategy for the code consisted primarily of regularly committing and pushing to the GitHub repository. GitHub itself has a sound backup strategy, such that complete failure there was unlikely. However, in addition to this I took manual monthly backups, which I stored in both Dropbox, and separately in an Amazon S3 cloud storage bucket.

The backup for the dissertation was provided by continuously synchronising to Dropbox. Dropbox provides primitive versioning and allowed me to go back to any specific day if I needed to. In addition to this, I manually copied the files into an Amazon S3[5] bucket to create a weekly backup should Dropbox fail.

2.6 Summary

In this chapter, I have described the results of the work undertaken prior to implementing the project. I have introduced distributed databases, and the difficulties in constructing successful consensus systems. I have also given some insight into the languages and tools used and why they were chosen.

The next chapter will provide detail of the implementation.

[5] http://aws.amazon.com/s3/

3 Implementation

3.1 Consensus Protocol – Paxos

I implemented the Paxos consensus protocol. This provides a guarantee that if more than half the nodes are online then consensus will eventually be achieved, and there will never be two different nodes which both believe they have reached consensus on different values.

The Paxos protocol differentiates between three different components:

1. The **Proposer** provides the key public API of allowing us to propose a value and see if it gets accepted by the distributed system.
2. The **Acceptor** is an internal tool to help proposers arrive at agreement. The acceptor never has to find out what actual value was ultimately agreed to.
3. The **Learner** provides an API to discover the results of rounds of Paxos that were never initiated by the current node.

The learner is not strictly necessary for a Paxos implementation, but it becomes useful for performance reasons. Having a learner makes it much easier for nodes of the system to stay up to date. The proposer is responsible for proposing an acceptable value for consensus.

In Wrench, Paxos-mediated consensus is used for the following:

- Generating unique sequence numbers. Each write transaction is given a unique sequence number to provide ordering.
- Agreeing whether a transaction was committed or aborted. It is fine to abort a transaction if and only if every node agrees that it was aborted. The consistency here is important.

3.1.1 Proposer

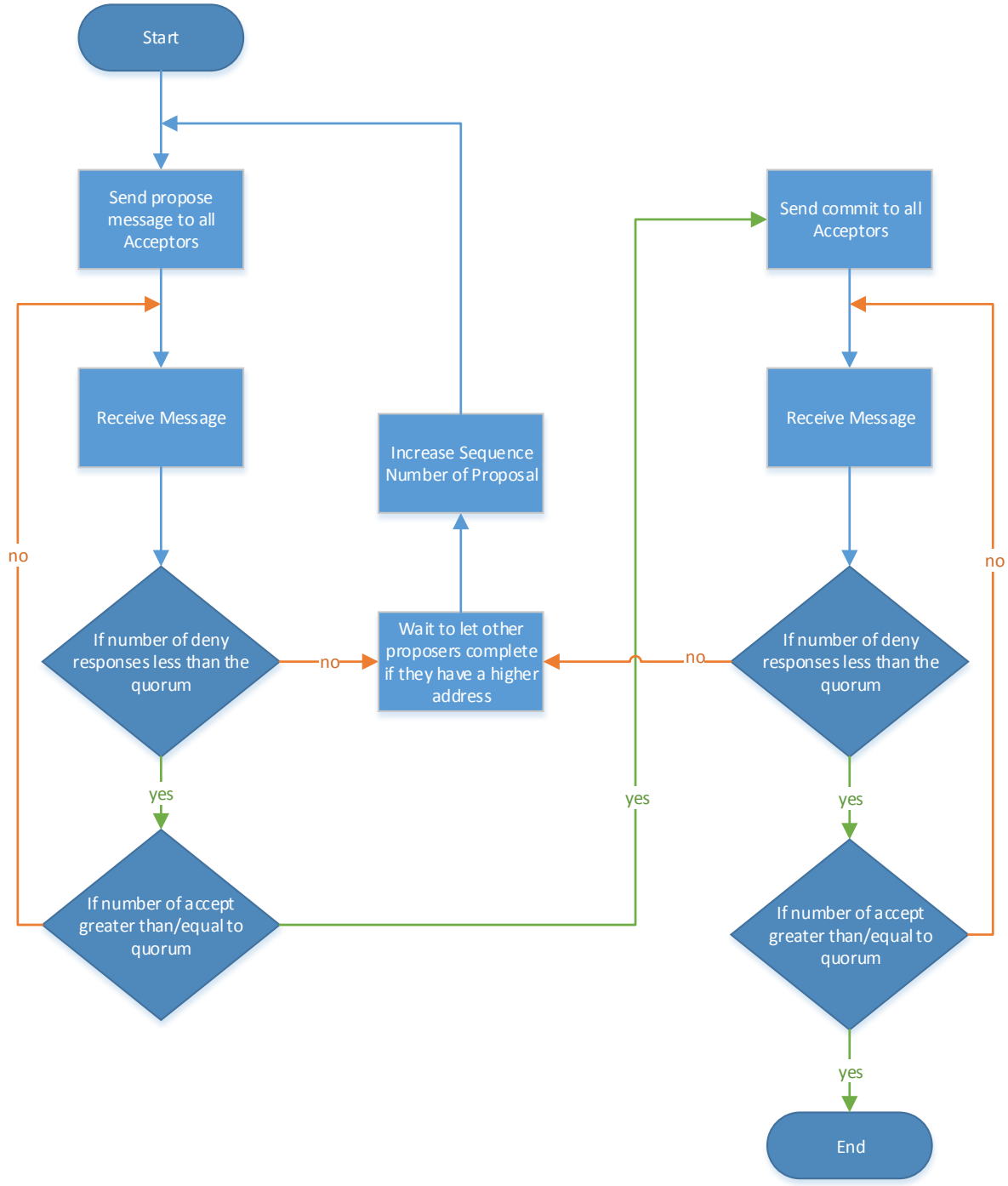

Figure 7: Paxos Proposer Algorithm

The proposer (see Figure 7) starts by choosing a sequence number. This must be unique across all nodes, which is achieved by picking a number and then appending the address of this node (each node must therefore be given a unique address). It then attempts to complete two phases, starting again with a higher sequence number if it fails.

The first phase is to get the proposal *accepted*; the second phase is to *commit* the accepted proposal. Each acceptor will accept the proposal if it has the highest sequence number seen by this acceptor so far. Once a majority of nodes have accepted the proposal, the proposer can attempt to commit the proposal. If a majority of acceptors accept the commit, the algorithm terminates with the value that was committed.

It is possible for two (or more) proposers to commit to a majority of nodes. The secret to the success of Paxos is ensuring that they commit to the same values. To do this, each response to a proposal includes the previously committed value if it has already committed. The acceptors can only commit to one value, and never change their minds, so they simply reply with that value (for the current round). This means that all proposers in the commit phase know about at least one of the values that have already started to be committed (if a majority of acceptors are in the committed state). By choosing one of those values in preference to its own value the proposer ensures that dulling proposers will end up committing the same value. In this way, only one value can ever be committed to a majority of acceptors.

3.1.2 Acceptor

The acceptor (see Figure 8) has two internal methods: *Propose* and *Commit*. These methods are to be called by the *Proposer* over the network, not to be used directly. Each one is largely independent of the other except for some shared state.

The propose method of the Proposer manages the overall algorithm, while the propose method of the acceptor is only there to ensure that any proposer attempt in the commit phase will have learnt about any successful commits with lower sequence numbers. This is what prevents three duelling proposers from being able to end up with a third of the votes each. In that instance, at least two of the proposers will have attempted to propose the same value, thereby giving them a majority.

The commit phase is the final part of the algorithm. It should be noted that different acceptors are likely to be committed to different values. It is the proposers and learners that all arrive at the same result. This is because the result is not arrived at until a majority of acceptors are committed to it. Once a proposer has received accept responses from the majority of acceptors to its commit proposal, the proposal is considered successful.

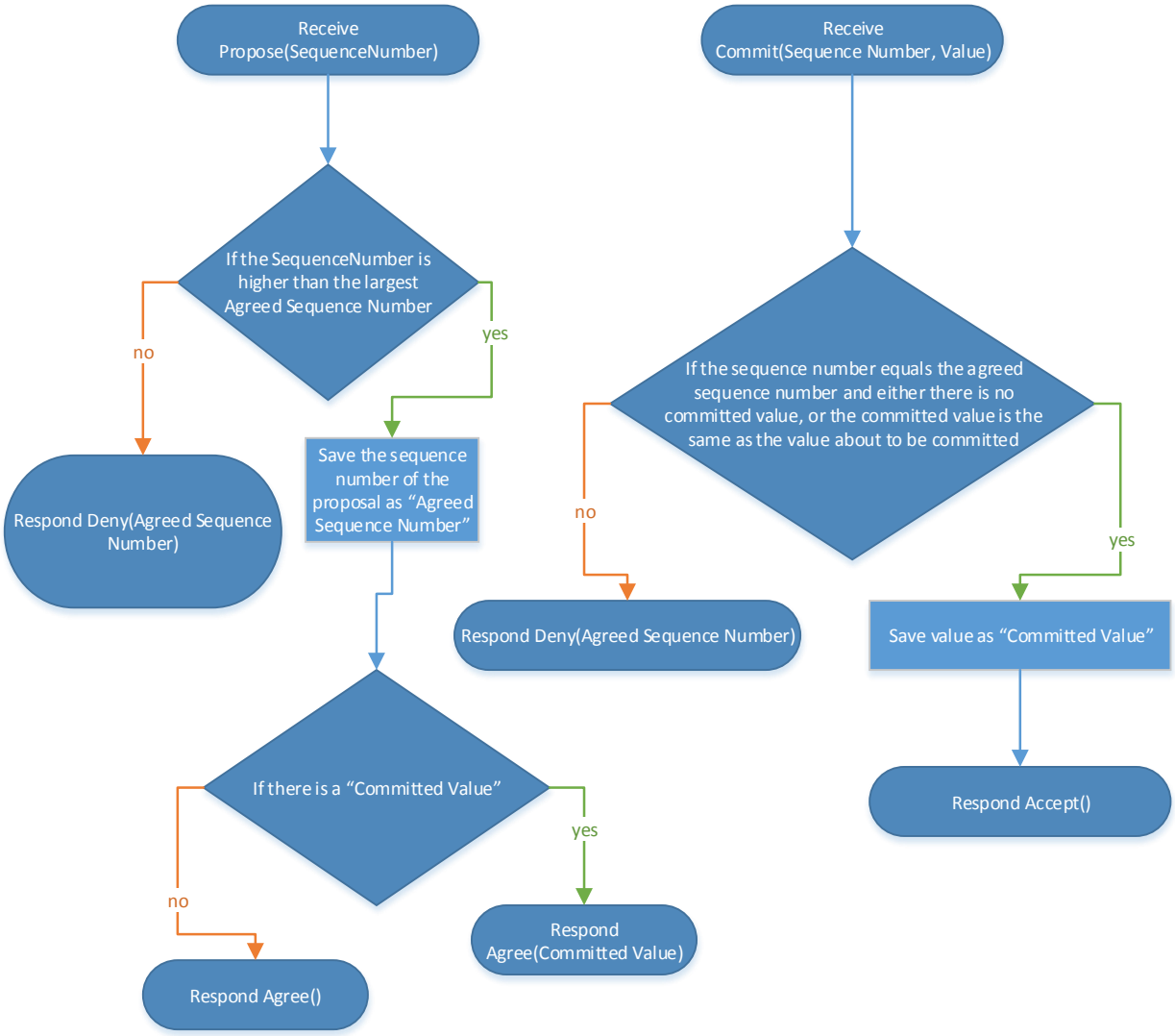

Figure 8: Acceptor Algorithm - note the internal state that must be persisted

Figure 8 shows the messages sent in response to Propose and Commit messages. When the response is `Deny` it always includes the highest agreed to sequence number. This is a performance optimisation so that the next time the proposer attempts to propose or commit it can ensure it sends a high enough sequence number.

3.1.3 Learner

The learner allows us to discover the result of a round of Paxos that happened, even if this node did not trigger it. It is always possible to find out the result of a Paxos round by proposing a value. However, if the round has not finished yet, the node attempting to discover the result might thus end up proposing the final value. In Wrench we ensure that this method is an option by having a special `SKIP` value that represents the empty transaction. The learner simply counts the number of unique accept responses for each value attempted to be committed. The fact that all messages are broadcast to all nodes means that the learners can be completely passive.

3.1.4 State

The proposer is completely stateless. It can be safely replaced at any time with no loss of consistency. The learner does store state, but if it is replaced, it will learn everything again over time with no problems. The acceptor stores state which is essential to the running of the algorithm. Replacing an acceptor at runtime requires somehow recovering that state. The only state that must be recovered for safety is which values the acceptor has committed to. It is safe (and in fact preferable) for the acceptor to forget what it thinks it committed to once the round has completed, and instead commit to the value ultimately agreed to by the round of Paxos. We can do this by using the learner (see Figure 9).

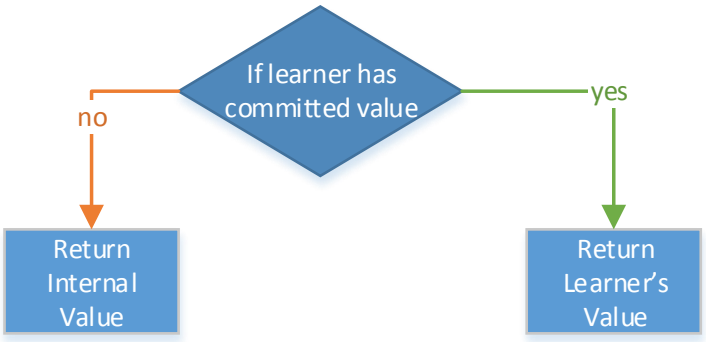

Figure 9: Learner Based Acceptor Recovery

If the learner is up to date, we do not need to worry about recovering the state of the acceptor anymore, since it will just use the learner's value.

I first ensure that we wait until the majority of learners are up to date before considering a Paxos round terminated. This just involves an extra loop at the end of the proposer's algorithm to ensure results are all distributed.

The second challenge is to recover the state of the learner. The reason this is easier than recovering the entire state of the acceptor is that every learner has the same set of values. All that is needed is to enter a recovery phase before accepting further proposals and wait until the learner has received the state from half of the other nodes. This way the learner will become up to date.

3.1.5 Combined

Together, the proposer, acceptor and learner make up one Paxos node. The interface for that Paxos node consists of two parts. It has a networking part which has an event that emits strings which must be broadcast to all other Paxos nodes, and a method to pass the received strings to. Handling of this broadcasting is carried out by a separate component. In the case of Wrench, the broadcast messages from Paxos are transported along with the other broadcast messages that distribute the results of transactions.

The other part of the interface consists of an event which is fired with the results of any rounds of Paxos that the learner discovers (regardless of whether or not they were initiated by this node) and a method that allows the caller to propose a value for a round.

3.2 Transaction Sequencing

Each transaction must be given a start sequence number which represents the point at which the transaction started. All reads are performed as snapshot reads at that sequence number.

In addition, each write transaction must be given a unique sequencing number when it is committed. This is an integer that is incremented by one for each transaction. Paxos is used to associate the Transaction ID with the Sequence Number. This ensures that no two transactions will ever have the same Sequence Number.

The Storage Node keeps track of the highest numbered Sequenced Transaction that it has seen. Using this, it attempts to propose the next value to Paxos. If no other nodes have committed a transaction in the meantime, this succeeds. If another transaction has already been associated with that sequencing number, it attempts to propose that transaction for the next sequencing number.

3.3 Backing Store

Many databases store most of their data in persistent mediums like SSDs or HDDs. This is essential if one needs to store more data than one can afford to keep in memory. Many simple websites have small databases though for storing data like session information, which would fit in main memory. It is now relatively inexpensive to build a machine that has 4 or even 8 GB of main memory. This allows for substantial databases to be built, which do not require complex indexing structures and yet remain extremely fast.

I have taken this approach by storing all data in main memory, in the same process as the rest of the system. This means that the store itself is just a simple C# object. That C# object is just a concurrent dictionary (a base class that implements a thread safe dictionary in C#) which maps strings to Value Stores. These Value Stores keep track of the actual data over time.

A value store keeps a second concurrent dictionary mapping sequence numbers to values. When a value is read, it looks for the highest sequence numbered write to that key that was before the sequence number of the current transaction and returns that value (or `null` if there is no such value). When a transaction is committed, it's results are `SET` by associating the value that is set with the Sequence ID of the transaction.

3.4 Read Transactions

The storage keeps the complete history of the system after every write transaction. When a read transaction occurs, the reads is done at a point in time. This provides *snapshot isolation*. A read-only transaction never has any conflicts, so it always commits successfully.

This works by getting the highest committed sequence number at the start of the read transaction at a given node and doing all other reads at that point in time. This means that two concurrent read transactions that begin on different nodes may read at different points in time.

The C# client library I created also provides the additional guarantee of "read your own writes". That is to say if we commit a write transaction before beginning a read transaction, the read transaction will always see the results of the write transaction. This is done by keeping track of the highest sequence number the client has seen. It sees the sequence number of any committed transactions as well. It then treats this as the minimum sequence number it will use in any future transactions.

3.5 Write Transactions

A write transaction starts out just like a read transaction. It receives the Sequence Number of the last completed transaction and treats that as its "start" number. All reads in the transaction are done at that point in time. During the transaction the client keeps track of all writes and reads. None of the writes are sent to the server during the transaction. It is up to the client to provide "read your own writes" within the transaction.

To commit, the client sends the server its starting Sequence Number, a collection of all the key/value pairs that were set, and a list of all the keys that were read.

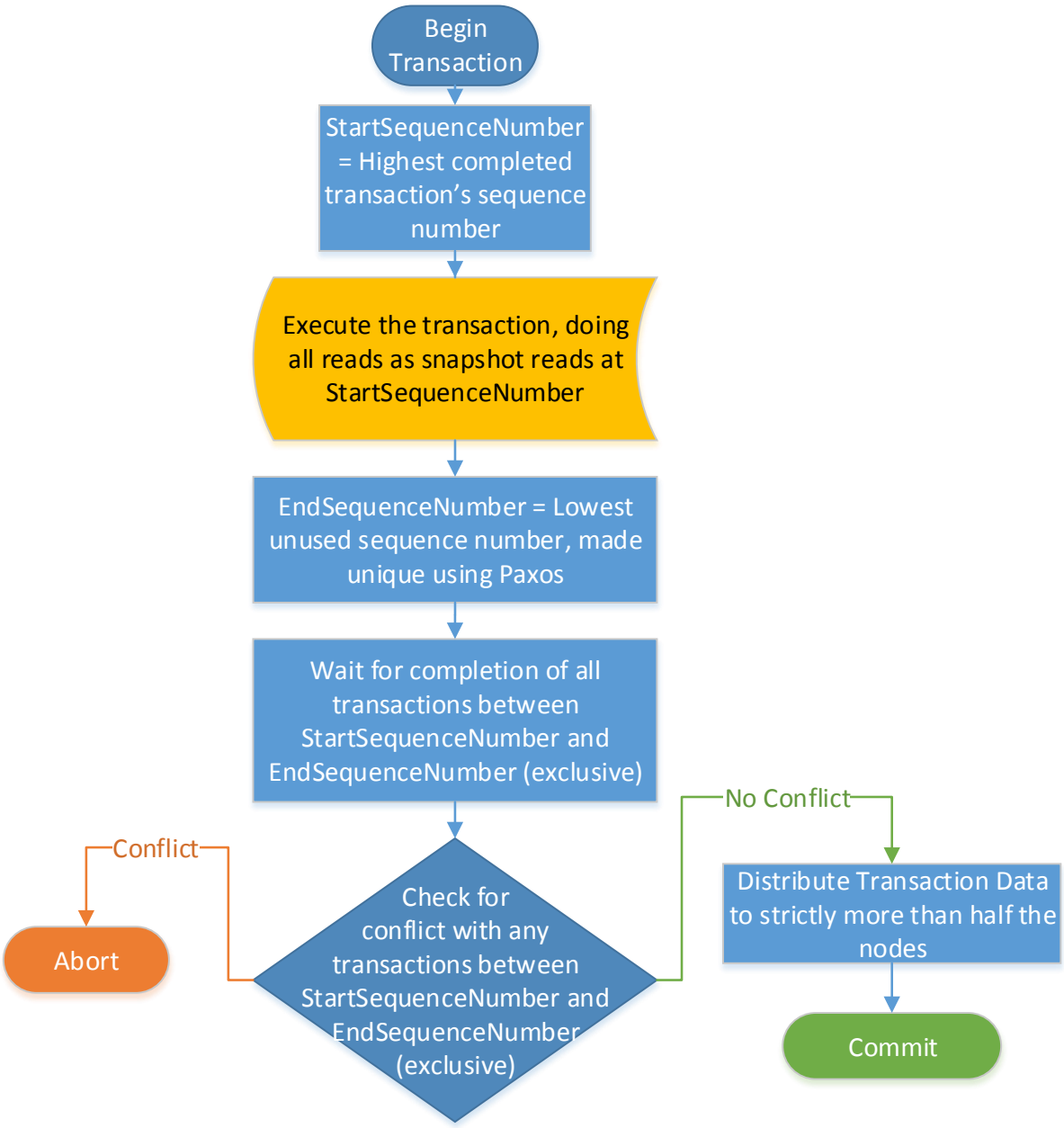

Figure 10: committing of a typical write transaction

3.5.1 Checking For Conflicts

Once the transaction has been sequenced and we know what order it goes in, it is a simple matter to check for conflicts using optimistic concurrency. The node waits for all previous transactions to complete (either *commit* or *abort*). It then checks against each transaction that was *committed* after the current transaction started, but before the current transaction's sequence number. If the current transaction read a value that was set by a committed transaction in this time frame, there is a conflict and the current transaction must be aborted. If there are no conflicts, the transaction can be committed.

3.5.2 Committing

To commit, first the data of the transaction is distributed. That is the set of keys that were set during the transaction and the values they were set to. Once this information has been received by a majority of nodes, we attempt to commit it by running a round of Paxos with the key set to the transaction ID and the value set to `COMMIT`. If that round succeeds then the transaction was committed.

3.5.3 Aborting

If the transaction is aborted, the same Paxos round must be run with the value set to `ABORT`. It is safe for any other node to time out the transaction and attempt to `ABORT` it in the case that the node trying to commit it fails while attempting to commit. The Paxos system guarantees that the transaction will either be *committed* or *aborted* everywhere and no nodes will disagree about the final status of the transaction.

3.6 Changing Nodes

The Paxos recovery, combined with simple systems to re-distribute the contents of past transactions, mean that any node can safely be replaced with an identical (but blank) node at any time. This is useful for replacing broken nodes.

The system also continues to function completely as normal for failure or removal of strictly fewer than half of the nodes.

These features also allow us to change the number of nodes in the system by replacing each node one at a time with a new node that has a different setting for the number of nodes in the set. If we are decreasing the number of nodes in the set then we must first remove a node and then replace each of the existing nodes. If we are increasing the number of nodes we must start by replacing each node and then add the new node.

4 Evaluation

4.1 Overall Results

The project proposal (Appendix B) established nine success criteria (condensed into eight here). All success criteria were met, with the exception of a number of small changes resulting from my decision to implement a masterless system for better failure tolerance.

1. A plan for how multiple concurrent transactions will be handled must be established.

Multiple concurrent read-write transactions are handled by using optimistic concurrency. Read-only transactions are handled using snapshot isolation, so they always see a consistent view of the world and never conflict with anything else.

2. An implementation of Paxos should be built to support election of a unique leader.

An implementation of Paxos was constructed. A system to use this to elect a master was also implemented; see Appendix A. Ultimately, the system I built did not require the election of a master so this functionality was not needed in the finished project.

3. A mechanism for synchronising the time across multiple nodes must be built

This would have been used for the master election. The system I built serializes transactions using unique sequence numbers (logical timestamps) rather than real time stamps. This meant that there was no need for fine grained synchronization of time across the nodes of the system.

4. An in-memory store must be built, which must attach time-stamps to each write and maintain multiple versions of each value.

This was completed in an earlier implementation of the project. However, during the course of the project the timestamps were replaced with a simple integer to sequence the transactions. It was not necessary to keep track of the actual time when writes occurred, only the ordering of them relative to other transactions. As such, a simple sequencing number was simpler and smaller to store. The final system does not allow reading at a point in time at the past, but it does allow reading at an old sequencing number, which allows reading at any state in the store's history.

5. Updates must be distributed from the master node to all slave nodes in a replica set.

Updates are distributed from the node where writes occur to all other nodes. The fact that there is no single master means that the writes are actually just distributed from wherever they occur, i.e. from whichever node is contacted by the client to perform the commit operation.

6. It must support strong consistency for read-only and read-write transactions.

Both types of transactions support consistency in the sense that transactions are isolated and serializable. It is not really strong consistency in that it does snapshot isolation so two separate clients may have transactions out of order. Strong consistency along with availability and partition tolerance is not really possible though due to CAP theorem. Under normal operation it will appear to be strong consistency.

7. The system must continue to function when some nodes fail (except Byzantine failures).

The system continues to function completely as normal in the failure of up to one fewer than half the nodes. It will even continue to function completely as normal if nodes fail in the middle of a transaction.

8. Tests should be performed to determine the speed, scalability and resistance to failure of the resulting system.

Section 4.2 and 4.3 describe how such tests were written and their results. I only test node failure by simulating it with network loss. This should not matter though as given a network that provides the all or nothing delivery the two are in fact identical.

4.2 Testing

The system consists of a number of separate components, each of which has highly testable functionality. As such, unit tests were written to test each component separately as well as in combination with other parts of the system. This was very useful in development for discovering a few edge cases that were not immediately obvious from reading the code.

4.3 Performance & Reliability

As a distributed storage system my project will be evaluated largely on performance and reliability. As a fully distributed key-value store, reliability means that it should carry on as normal even when some of the nodes have failed.

4.3.1 Paxos

I will start by evaluating the performance of my Paxos implementation on its own before going on to look at the resulting performance of the entire system. The performance of the Paxos implementation is central to the system performance as every write transaction includes at least two rounds of Paxos (one to sequence the transaction and one to commit it).

The Paxos implementation slows down linearly as nodes are added to the system (Figure 11). This means that adding more nodes will be unlikely to increase write performance significantly in its current form. Read performance is not limited by the speed of Paxos though, so should scale more effectively. There are also ways of reducing the number of network messages required for each round of Paxos by acquiring the propose phase early.

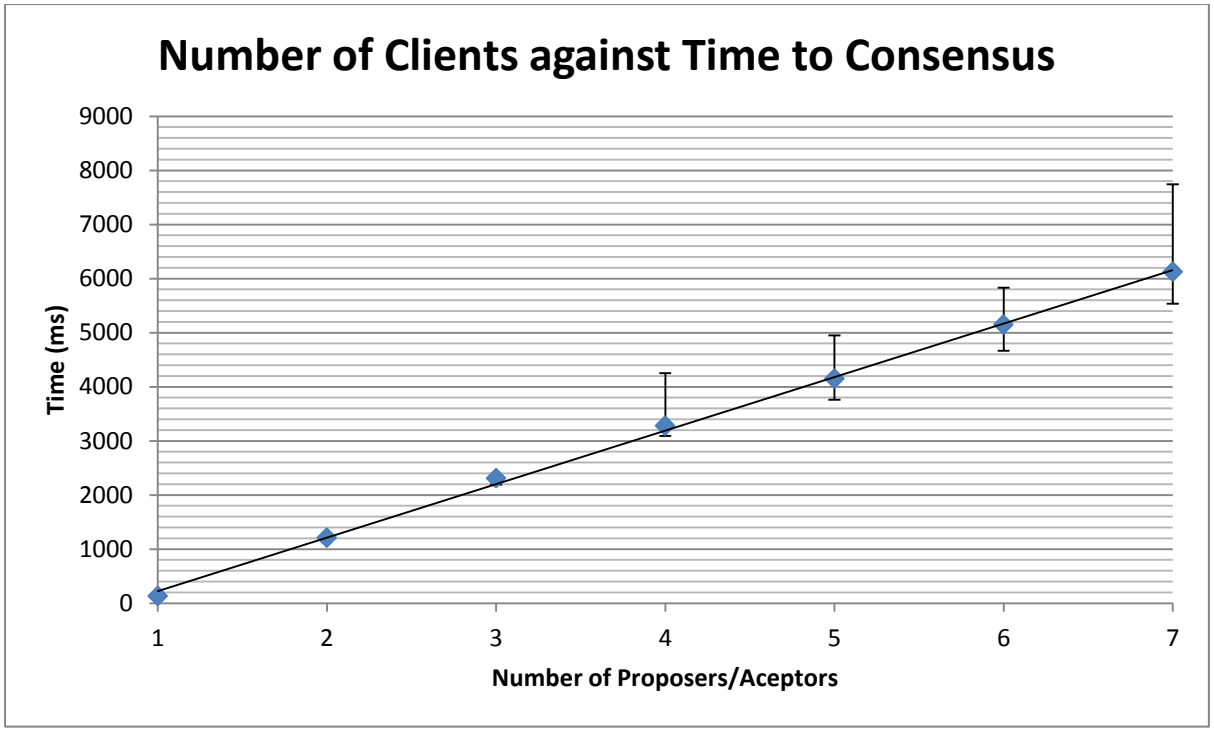

Figure 11: Number of Clients against Average Time to Consensus with network delays of 20ms, over 100 runs. Error bars show maximum and minimum times recorded.

Paxos also slows with networking delay. Figure 12 shows behaviour with network delays that are randomised to be between 0 and the value on the x axis. Interestingly, it does not get linearly slower as we add acceptors when the network delay is uniformly random. This means that in a typical real world system we could add more nodes without sub-linear slowdown of write transactions. This is because in the case of three acceptors, only the first two messages must be waited for in order to achieve a majority. This means that although more messages are sent, the proposer does not need to wait for the responses to be received from those additional messages.

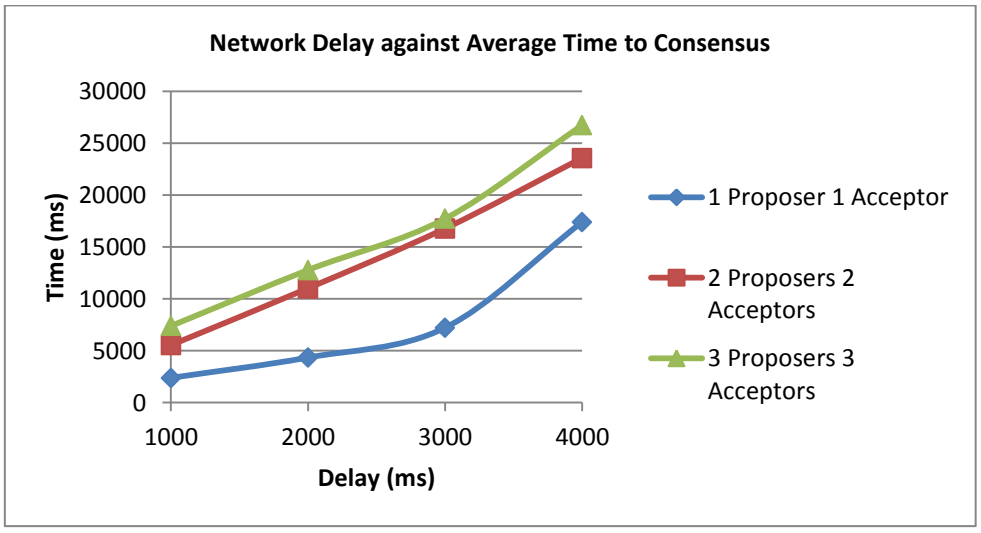

Figure 12: Maximum Network Delay (uniformly random between 0 and t) against Average Time to Consensus (over 150 runs)

We get a similar story with network loss (Figure 13). The greater the network loss the longer it takes to reach consensus. Here the advantage of having at least three nodes is even more pronounced, to the extent that it is actually faster to have more than three nodes than two once the network is lossy. The third node allows for the round to complete even when some network messages are lost. With fewer than three nodes, all messages must succeed. One proposer and acceptor is still faster because there are fewer messages to be dropped and no duelling proposers so no requirement to wait for back offs.

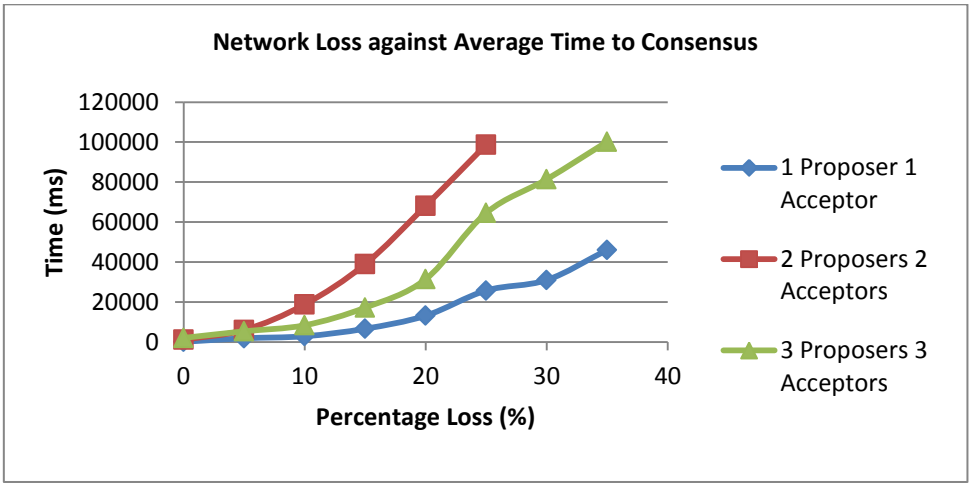

Figure 13: Percentage Network Loss (Uniformly Random) against Average Time to Consensus (over 150 runs)

4.3.2 Overall

I ran overall tests assuming the network delay between nodes was 1ms each way, which is reasonable if they are located in the same data centre. I also assumed a delay of 100ms each way between clients and the storage nodes, modelling the worst-case delay on the wide-area network between client and data centre. I ran all the tests against a cluster of five nodes.

4.3.2.1 Write Transactions

I tested write transactions by running a transaction that read two keys and wrote two keys sequentially on a single client and measured how many times it could be executed per second over 100 runs. This only achieved one write per second. This performance is dominated by the slow connection between client and server. To give a better measure of scalability, I decided to re-run the tests with that delay set to one millisecond. This gave 3.5 transactions per second.

4.3.2.2 Read Only Transactions

I tested transactions that each read four keys sequentially on a single client and saw the same results as for write transactions of one transaction per second. I then repeated the experiment with delays between server and client set to one millisecond and achieved throughput of 6 transactions per second. This is an expected result as reads have less internal traffic than write

4.3.2.3 Conflicting Transactions

I also tested the performance of transactions that conflicted by running two transactions in parallel threads. Each transaction attempted to read increment and set two integers, one that was a shared integer and the other that was private. Over 50 runs of this experiment it took an average of 130 seconds for the shared integer to reach 50 and in every case the private integers summed to 50, which helps back up the claim of strong consistency. Both integers were also non-zero at the end of every test run, which demonstrates that one process does not totally starve the others.

4.3.3 Usability

It is important to review the usability of the API. Unfortunately this cannot usefully be done empirically so I will describe the API and consider its usability using an exploratory approach.

The set up handles temporary node failures completely transparently (provided fewer than half of the nodes fail). Providing fewer than half of the nodes are swapped out at any time, and new nodes are allowed time to become up to date, all the nodes can be replaced completely transparently. This presents a possible method for dynamically changing the size of the set of nodes at runtime. To increase the number of nodes, simply replace each node one at a time with a node that is set to a higher number of nodes, and then add the additional node.

The client API provides a very simple API for read transactions (an example of which can be seen in Figure 14) that simply uses the `BeginReadTransaction` method to return a `ReadTransaction` object and then the `Read(key)` method of the `ReadTransaction` which returns the value of that key.

```
using (var transaction = client.BeginReadTransaction())
{
    var foo = await transaction.Read("foo");
    Assert.IsTrue(foo == "bing" || foo != "boo");
}
```

Figure 14: A Typical Read Transaction

The API for write transactions (an example of which can be seen in Figure 15) provides the additional methods `Write(key, value)` and `Commit`. Since nothing is actually written until we call `commit`, it is acceptable to abort simply by not committing. If there are any conflicts, the call to `commit` will result in an exception. As such, it is recommended that clients support retrying the transaction in that case. An API was also provided to automatically retry a transaction until it succeeds.

```
using (var transaction = client.BeginWriteTransaction())
{
    if ("boo" == await transaction.Read("foo"))
    {
        await transaction.Write("foo", "bing");
    }
    await transaction.Commit();
}
```

Figure 15: A Typical Write Transaction

5 Conclusion

The project was a success. It provides a key-value store with strong consistency and high availability. It will continue functioning for reads even if only one node remains and will be fully functional for writes provided more than half the nodes continue working. This could be extremely useful in applications such as banking where high availability and strong consistency are paramount.

Since the current system requires every node to have the entire store in memory at all times, there are many ways this could be improved. It would be useful to support backup to a persistent store. In order to scale beyond very small, fully replicated, data stores, the system will need to implement sharding. The idea here would be to partition the data based on the keys, so that each node only stores some portion of the keys. Work would then be required to make the same guarantees about consistency with transactions that span multiple partitions.

The system currently stores a complete history of all data for all time. It would be possible to implement garbage collection. Once all nodes are aware of the results of a transaction and there are no reads still attempting to read on or before that transaction, the details of the transaction are no longer needed and can be disposed of. This would be important if the store was to be used over a long period of time.

6 Bibliography

Brewer, E. (2000). Towards Robust Distributed Systems. *Symposium on Principles of Distributed Computing (PODC).*

Lamport, L. (2001, 11 01). *Paxos Made Simple.* Retrieved 05 15, 2013, from Microsoft Research: http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf

Appendices

A Master Election

One possible use case for a Paxos implementation is to build a master election system. This is one of two ways in which my project makes use of Paxos. It should be noted that master election is actually just a special case of locking. If we take a key as a parameter to a master election, it becomes a lock system. I have chosen to do master election and then have that master hand out locks, instead of running Paxos each time I want a lock for something. For the most part, given a reliable master, this will prove significantly faster since just one single round trip time is required to the single master to obtain a lock. A complete round of Paxos, by contrast will likely generate at least two times the number of nodes worth of round trip times, and every message is broadcast to every node.

To construct master election out of the consensus system it is just a case of running repeated rounds of Paxos for each time period. The time period for each election is passed in as a parameter, and must divide exactly into an hour (e.g. 5 minutes, 1 minute, 10 minutes but not 7 minutes) so that the system can easily chose a unique key for each time slot. The start of each time period is then used as the key for the round of Paxos. Longer time periods reduce the overheads of frequent Paxos elections, but increase the time the system will be offline for in the event that the master fails without warning.

This is sufficient to ensure that there is at most one master, and that a new master will efficiently take over one round later if the first master fails. It has a problem though: because the master is likely to change with every round, and, due to potential clock drift, there has to be a delay during the handover in which neither node is the. The new master needs to wait until the old master will definitely know its round has finished, which means that it must wait for the total clock drift possible between the two nodes. This clock drift is minimised using NTP (discussed later), but in normal operation we want to avoid this delay.

We have to achieve two properties.

1. Keep a consistent master in normal operation (only switch over if it fails).
2. Avoid the delay when not switching masters.

To achieve the first, we just have to ensure that the current master gets priority when attempting to win the next round of election. If a master is taking over from itself, it does not need to wait for the timeout for clock drift to expire, so that seems like the second problem solved. The reality is that we still have the problem of the time spend performing the round of Paxos. To get around this, the current master is allowed to propose itself early. This serves to give it priority in the election (satisfying one) and ensures there is no delay waiting for an election (satisfying two).

This means the master always knows it is the master, and has a consistent tenure on the position. The remaining nodes still need to know that it has a consistent tenure, though. If they were just using the proposer to find out who became master, they would not realise that the master was continuing in their capacity until it was too late, which would result in momentary gaps. To fix this, I use the learner, which learns of the results as soon as the current master does, thus allowing everyone to be aware that the master is not changing and thus has a consistent tenure.

A.1 Read-Only Transactions with a Master

The read transactions would be done in the same way with the addition of a master as without. They would have snapshot isolation and would happen at any node at any time.

A.2 Read/Write Transactions

Transactions that write, as well as read, values must take out a lock on all values they either read or write. To simplify things, the master server orders transactions. The master server gives each transaction a sequential ID upon arrival. It is also given a randomly generated GUID (Globally Unique Identifier). A transaction can only start once all transactions that have a lower ID number are committed, rejected or do not touch any of the same values.

The write transactions only read from the master, so they can be sure to see the latest values. All their "writes" are considered tentative initially. A list of write operations is stored for the transaction. The transaction can be rolled back simply by deleting that list of operations. To commit it, those operations are applied to the underlying data model.

To determine whether a transaction is committed or not, a round of Paxos is executed. The full state of the transaction is first distributed to strictly more than half the nodes in a "tentative" state. That way a node that learns the transaction has been committed will be guaranteed to be able to obtain the full state of the transaction (providing it can contact strictly more than half the nodes). Once the state of the transaction is distributed to all nodes, the master attempts to run a round of Paxos with the RoundID equal to the ID of the transaction and the Proposed Value equal to the GUID of the transaction. If the master changes mid transaction it is possible that two transactions might tentatively be given the same ID, but this ensures that only one of them could be committed.

If a non-master node has a pending transaction which it does not know the result of, it can find out the result of the transaction simply by trying to abort it. This is done by running Paxos with the RoundID equal to the ID of the transaction and the value equal to "ABORT". One of 3 results will then occur:

1. The result of the round of Paxos will be the GUID of the transaction. In this case the transaction should be committed.
2. The result of the round of Paxos will be "ABORT". In this case the transaction must be aborted and no transaction will ever be allocated that ID.

3. The result of the round of Paxos will be another GUID of a different transaction. In this case, that other transaction needs to be committed with that ID, but the current transaction must be aborted.

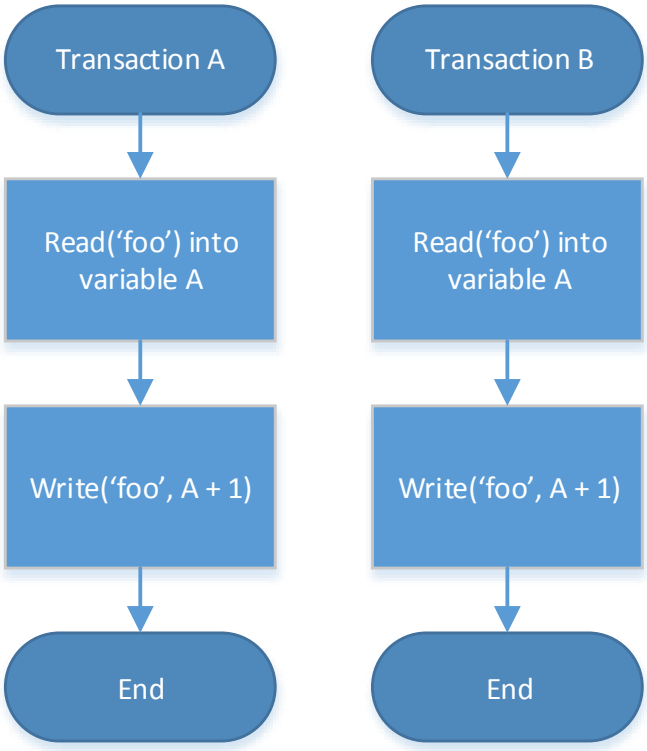

Figure 16: Two parallel write transactions may conflict

Locking and NTP

Most of what I have described thus far could make sense with optimistic concurrency. i.e. just check at the point you try and commit a transaction that there is no other transaction that interacted with the same data. The issue with this is that two transactions that interact with the same data and overlap in time span will always fail. This failure wastes a lot of time and resources if the clients have to back off and retry. It is pretty much impossible to build a system that will never force a transaction to retry, and yet is fully resilient to the failure of a master node, but greater performance can be achieved by getting close.

The system I have built attempts to ensure that no two concurrent read/write transactions will interact with the same data. It is worth noting that read transactions never suffer from this problem since they always just read at a consistent point in time. Two transactions technically only conflict if both transactions attempt to read something that the other transaction wrote. A simple example would be two process both trying to atomically increment a value (see left).

Two transactions like that will never compete successfully if run in parallel. This could ultimately lead to no progress being made. To attempt to prevent this, we try and ensure that each transaction waits for any lower numbered Transaction IDs to be committed before it can start. This is implemented by taking a lock out with the master server for all keys a transaction will touch. To prevent deadlock the locks are always acquired in alphabetical order.

The system remains safe, just less performant, if this locking is not truly fully exclusive. As such we can handle handover from one master to another by just resetting the list of locks we believe to be held by clients.

With this in mind, it turns out you can set the "drift" time to pretty much 0 in the master election system, because whilst it is desirable to have only one master as often as possible, it is not actually the end of the world if there are temporarily two masters.

B Original Project Proposal

A Distributed Key-Value Store With Strong Consistency

B.1 Introduction and Description of the Work

Key-value databases are very simple databases that support extremely limited query functionality on keys and, in exchange, return a value. One example might be a file system. You can very efficiently read all file names in a directory, but it is very slow to query the files' content. Extracting a file's contents when you have the file's name is also very efficient. In this way file systems are a basic form of key-value store.

Redis[6] is an in-memory key-value data store. It has proven very useful due to its high performance and simple interface, however the support for replication in Redis is very limited; it currently only supports master-slave replication where slaves are read-only (in order to avoid having to deal with distributed consistency issues). It also has beta support for automated failover to a slave if the master becomes unresponsive.

Redis also has no support for sharding. That is to say, all the data for one Redis database must be on one node. This means the only way to scale the system is to improve the performance of a single computer. After a certain point it is much more efficient to add additional computers than increase the performance of a single computer, so this is not ideal. Sharding resolves this by letting you store some data on one set of computers and other data on other sets of computers. Using this technique, additional computers allow you to store more data, in addition to providing more throughput.

Google recently published a paper on their spanner global distributed data-store [2], which uses Paxos[3] and atomic clocks to support both high availability and strong consistency in transactions. The code itself is not open source, but some of the ideas could be applied to a simple key-value database to provide the same properties of availability and strong consistency.

This project proposes to build a simple key-value store, with an interface similar to that of Redis, but incorporating features of high availability while maintaining strong transactional consistency. Sharding may also be added as an extension to the project if time permits. I will

[6] Redis, http://redis.io/

initially support a subset of the commands supported by Redis (such as GET key and SET key value), but this could be gradually extended.

B.2 Resources Required

I am planning to implement the system using C#. I will develop primarily on my desktop computer, my laptop will serve as a backup, and the PWF as a last resort if that fails. I will use Visual Studio for development and git for version control with GitHub[7] as a backup. I will also periodically copy the entire directory to Dropbox[8] as a further backup.

B.3 Starting Point

To complete the project I will draw from

1. **Part 1B Courses from the Computer Science Tripos**
   Specifically the Concurrent and Distributed Systems course will be relevant. Some of the additional reading material on Paxos will also be extremely beneficial.
2. **Spanner[9] Paper and Paxos Made Live[10]**
   There are a number of papers that detail the theory behind building highly available distributed systems. I will draw on these extensively and will use pseudocode provided in these to implement many of the algorithms needed.
3. **Previous Programming Experience**
   I have considerable programming experience both from working and as a hobby in *C#* and *JavaScript*.

B.4 Substance and Structure of the Project

The objective of the project is to design and implement an in memory, key-value database with strong consistency that does not rely on any manual intervention to specify a master and can continue operating effectively even in the failure (excluding Byzantine failure) of up to one fewer than half the nodes in the system.

The intention is to implement a subset of the commands found in Redis, which is a popular in-memory key-value store. I intend to extend the semantics for transactions though. I will also be providing support for replication.

In order to provide this system, there will need to be a way to elect masters. Each replica set (the set of nodes that together make one database or one shard if sharding is supported) will have precisely one master, and must be able to cope with the failure of that master. This will mean using Paxos.

[7] GitHub, https://github.com/
[8] Dropbox, http://www.dropbox.com/
[9] Spanner, http://tinyurl.com/google-spanner
[10] Paxos Made Live, http://tinyurl.com/paxos-made-live

In order to co-ordinate transactions, a notion of time must be established and agreed upon by all nodes in the database. This can be achieved using NTP, where all nodes in a replica set take their time from the master. This will allow support for reading from slaves rather than always the master node in a replica set.

Read-write transactions must both lock any data they read and ensure that writes are not seen by any other clients until the transaction has committed. This can be achieved as in Spanner by using 2 phase locking. The deadlock case will also need to be detected and handled by rolling back the transactions involved.

B.5 Success Criteria

For the project to be deemed a success the following items must be successfully completed.

1. A plan for how multiple concurrent transactions will be handled must be established
2. An implementation of Paxos should be built to support election of a leader
3. A system should be implemented that automatically maintains one leader regardless of the addition or removal of nodes.
4. A mechanism for synchronising the time across multiple nodes must be built.
5. An in-memory key-value store must be built, which must attach time-stamps to each write and maintain multiple versions of each value
6. Updates must be distributed from the master node to all slave nodes in a replica set.
7. It must be possible to execute read-only transactions and read-write transactions with strong consistency, such that the transactions are serializable.
8. The system must continue to function even when some nodes fail; here failure is defined as the failure to respond within some time-out. This does not include Byzantine failures.
9. Tests should be performed to determine the speed, scalability and resistance to failure of the resulting system.

B.6 Optional Extensions

It would be highly desirable to implement a system for sharding if time permits. This would allow scaling horizontally by adding more nodes to store more data.

A relatively simple, and worthwhile, extension would be to support persistence to disk. This would provide a much more reliable database long term.

The project may begin to provide closer to the full API for Redis and become an option as a drop in replacement.

It could also begin to automate the management of group membership. Under some situations it may be safe to automate removal of an unresponsive node or add a new node.

B.7 Timetable and Milestones

22nd October – 4th November

Study papers on *Paxos* and produce a working implementation of a single round *Paxos* state machine.

5th November – 18th November

Select some simple commands for manipulating a key-value store (excluding transactions) and build a data structure for storing these key-value pairs and history to allow querying at any point in time. Build a simple command line application to allow an easy way to interface with this key-value data store.

Milestone: Key-value pairs can be stored and read and if a time-stamp is provided they can be read as they were at some point in the past. Keys can also be written in the future such that the results of the write only appear after a period of time has passed.

19th November – 25th November

Extend the *Paxos* implementation to support multiple rounds and to elect a leader with a (configurable length) lease. Test for correctness and performance (minimise network traffic when no election is in progress, maximise speed of new master election when node fails).

26th November – 9th December

Use the *Paxos* master election and key-value store to implement a highly available key-value store that is transactionally sound for a database replicated across multiple nodes.

Milestone: Key-value pairs can be stored and read in a distributed system which can continue to operate in the case of failure of some nodes. Results can be observed to be transactionally consistent.

10th December – 23rd December

Build a system using NTP to synchronize the times throughout the system so that time-stamps can be relied upon to accurately serialize messages. Use these time stamps to support read-only transactions reading potentially stale data from slave nodes.

Milestone: The same behaviour as in stage 4 can be observed, but with the ability to have read-only transactions run on slave nodes, rather than master nodes.

24th December – 6th January

Tidy up any issues with the implementation and fix any bugs that arise. It is likely that there may be stability or performance issues with the implementation that can be resolved at this point.

If that is not the case, some work on optional extensions could take place here, such as sharding across multiple replica sets.

7th January – 20th January

Write initial chapters of dissertation.

Milestone: Preparation chapter of Dissertation complete, Implementation chapter at least half complete, code should be in a state that could be submitted with no further work in the worst case scenario.

21st January – 27th January

Prepare progress report **(Due in noon on Friday 1st February)** and progress report presentation.

28th January – 3rd February

Create and test some purposefully complex transactions such as those prone to deadlock and those prone to failure if not correctly serialized.

4th February – 24th February

Test the implementation for performance, scalability and availability when some nodes fail.

Begin the evaluation chapter and create appropriate graphs of performance/scalability.

25th February – 31st March

This time will be used as a contingency for any previous stages that over-run and to allow time to implement some optional extension features.

Milestone: Finish all coding, implementation, testing and a draft of the evaluation section of the dissertation.

1st April – 28th April

Write and submit draft dissertations iteratively for review by supervisors.

Final changes should be made before the start of May and any spare time will be used primarily for revision, although it also acts as a final buffer if previous stages overrun.

The deadline for submission of the dissertation is **noon on Friday 17th May**

Milestone: Submission of Dissertation.

B.8 Resources Declaration

I am planning to implement the system using C#. I will develop primarily on my desktop computer, my laptop will serve as a backup, and the PWF as a last resort if that fails. Since

there is no requirement for especially high performance, any of these machines should be more than adequate. I will use Visual Studio for development and git for version control with GitHub as a backup. I will also periodically copy the entire directory to Dropbox as a further backup.