Charlie Crisp

Building a Blockchain Library for OCaml

Computer Science Tripos – Part II

Pembroke College

April 10, 2018

Proforma

Name:	**Charlie Crisp**
College:	**Pembroke College**
Project Title:	**Building a Blockchain Library for OCaml**
Examination:	**Computer Science Tripos – Part II, July 2018**
Word Count:	**????**[1]
Project Originator:	KC Sivaramakrishnan
Supervisor:	KC Sivaramakrishnan

Original Aims of the Project

To build a library in OCaml, which can be used as a building block for blockchain applications. The library should allow participating nodes to own a shared copy of a blockchain data structure, agreed upon using consensus. Nodes should also be able to commit transactions to the blockchain, which should then be visible to other participating nodes.

Work Completed

All that has been completed appears in this dissertation.

Special Difficulties

None

Declaration

I, Charlie Crisp of Pembroke 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 extent for a comparable purpose.

Signed

Date

Contents

List of Figures

Acknowledgements

I would like to thank KC Sivaramakrishnan for being an extremely helpful supervisor throughout the duration of the dissertation, as well as over the past three years.
I would also like to thank Anil Madhavapeddy for allowing me to use his laptop for the duration of the dissertation, and being a very supportive DoS.
Finally I would like to thank my friends and family for supporting me through my final year.

Chapter 1

Introduction

Blockchain technology has existed for a long time, but the definition of 'blockchain' has changed drastically since its conception. Previously used just to describe a data structure, the term 'blockchain' is now widely used to also describe the accompanying consensus mechanisms. This is mainly due to the increasing popularity of cryptocurrencies such as Bitcoin [9] which use the 'Proof of Work' algorithm to solve the double spending problem. Blockchain is undoubtedly the most important technology in the field of cryptocurrencies, where no single client can be trusted, however, it also has many use cases outside this sector. For instance, it can be used in situations where clients *can* be trusted like a hospital maintaining internal medical records, or a bank wishing to record transactions from many of its own distributed clients.

In this report I will present 'Logan', a blockchain library in OCaml which allows the easy creation of blockchain applications. The blockchain is synchronised via a leader-based consensus mechanism with strict consistency. Developers using this library are also able to define custom validation of transactions being added to the blockchain. Because the application is written in OCaml, it can be compiled to bytecode, unikernels or even javascript and is therefore suitable for a wide range of destination applications and devices.

1.1 The History of the Blockchain

The blockchain, in its simplest form, is a series of blocks of data, where each block contains the hash of the content of the previous block. Figure 1.1 is a graphical representation of a typical blockchain data structure.

The blockchain, as a cryptographically secure chain of blocks, was first conceptualised by Stuart Haber and W. Scott Stornetta in 1990 [7]. However, until the creation of Git [12] in 2005, the blockchain was still a relatively niche concept. The invention of Bitcoin in 2008 is seen by many as the most pivotal moment in the history of blockchain technologies. Bitcoin uses the Proof of Work consensus algorithm to create a decentralised, trust-less, peer to peer network which is used to make transactions between virtual wallets.

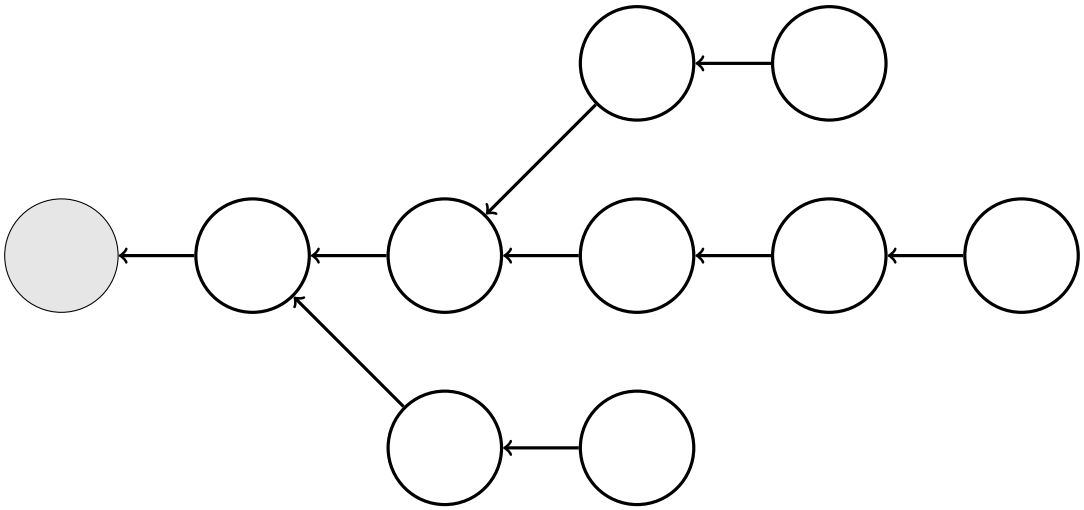


Figure 1.1: A typical blockchain structure

1.2 Blockchain Today

At the time of writing, cryptocurrencies are generating both a huge amount of excitement and cynicism in popular media. Aside from Bitcoin, cryptocurrencies like Ethereum [2] have introduced the concept of Smart Contracts which allow the execution of code on the blockchain.

Whilst it is possible to think of applications for blockchain technology in almost every sector, the development of applications outside the scope of crytocurrencies has been limited. If one considers the example of OCaml, there are currently no libraries which allow a user to easily get started with building blockchain applications.

1.3 Work Completed

I have created Logan, a library that allows developers to create blockchain applications with the ease of importing a library. Logan was designed to exist outside the realm of cryptocurrencies and therefore assumes that all Participants are trustworthy. Consensus is achieved by using a simple leader-based mechanism where the Leader will periodically pull updates from Participants and add them to a central blockchain. This blockchain can viewed by all Participants, with the guarantee of strict consistency. All that needs to be done to set up a network is specify the location of the Leader, Participants, Replicas and to specify any validation required.

I have evaluated Logan's performance by measuring latency, throughput and how the latency of adding transactions changes as the blockchain size increases. My results showed that Logan is capable of dealing with throughputs of much beyond my success criteria. I have also shown that for a network with remote Participants, the latency of

adding transactions increases linearly with blockchain size. However, I have been able to attribute this to the implementation of Git that Irmin/Ezirmin uses to pull updates over the network. I have also shown that pure Git achieves a much better efficiency and that a better OCaml implementation of this protocol would allow Logan to achieve constant transaction latency with blockchain size. Finally, I have shown that nodes can be added to a Logan network with only a modest increase in transaction latencies.

Chapter 2

Preparation

Before starting work on the project code, I completed a lot of preparation in order to aid the development process. I spent some time learning to use OCaml and familiarising myself with its features. This was important because it allowed me to write idiomatic code which was not only powerful, but could also provide an intuitive API for other developers to use. I also spent some time investigating a few key libraries, such as Irmin and Lwt. Understanding these libraries, the data structures they provide, and the technologies they present, allowed me to focus on the technological challenges in the project and not waste time reinventing the wheel. Setting up a good development environment allowed me to run automated builds and, therefore, catch any errors in the code early. This was particularly useful during the evaluation as it involved installing the project on many remote machines. Being able to catch issues like dependency resolution or out of date build commands made the evaluation process much easier. Lastly, I spent some time developing a requirements analysis for the final product. This analysis helped drive the design and development of the blockchain library whilst not restricting the work that I was able to complete.

2.1 Starting Point

The project built upon functionality provided by Irmin [1] which is a distributed data store system. Irmin is fast, durable and has all the necessary capabilities required to build a blockchain. The project also made use of Ezirmin [11] which provides a simplified API to Irmin, and defines a log data structure which was used to build the blockchain and mempool structures for this project.

2.2 Using OCaml

At the start of the project, I had never used OCaml for any project of significance. Whilst the first year Foundations of Computer Science course had given me some background into functional programming, there were still many key OCaml features which I had to learn. In the first few weeks of my project, I spent time studying the book Real World OCaml [13] which proved a great introduction to many of OCaml's features.

Pattern Matching

OCaml provides a very powerful syntax for matching patterns which allow you to write functions as shown by Listing 2.1.

Listing 2.1: OCaml Pattern Matching

```
let pattern_matcher_1 input = match input with
  | Card("spades", 1) -> Printf.printf "It's the ace of
     spades!"
  | _ -> Printf.printf "Unlucky"
let pattern_matcher_2 = function
  | Card(suit, 1) -> Printf.printf "It's the ace of %s!"
     suit
  | _ -> Printf.printf "Unlucky"
```

Listing 2.1 shows a function that will print a special string if it is passed the Ace of Spades. Here, the pattern matching checks the that the tuple associated with the data type contains the string `spades` and the number 1. The second example assigns the variable `pattern_matcher_2` to an anonymous `function` which matches the first argument of the tuple to the variable `suit`.

Options

An `Option` is a built in data type that provides a powerful way of using OCaml's pattern matching. By using the `Some(...)` and `None` constructors, one can create something of the type `'a option`. This is comparable to the `null` type in languages such as Java, however as it is part of the type system, it forces the programmer to handle any cases where `null` could be returned.

Error Handling

OCaml provides multiple different ways of dealing with errors and exceptions. A simple way of signifying an error in your return type, is to return an `option` which will be `None` if there is an error. Whilst this can be inflexible for larger solutions, it also provides a quick and simple way of signifying that something has gone wrong.

Another way of dealing with options in return types is to use the `bind` function:

```
val bind: 'a option -> ('a -> 'b option) -> 'b option
```

As the type signature demonstrates, `bind` takes an `option` and applies a function to its contents if it exists, or returns `None` otherwise.

OCaml provides a built in type `Result.t` which is an extension of optional return types, where the programmer is able to define arbitrary data to accompany the error type.

The following demonstrates a successful return type of `int` (with an unspecified `Error` type), and a `string` error type (with an unspecified `Ok` type).

```
# Ok 3;;
- : ('a, int) result = Ok 3
# Error "Something went wrong";;
- : (string, 'a) result = Error "Something went wrong";;
```

Polymorphic Variants

OCaml allows the programmer to define variant types such as the `Card` type that was defined earlier. This makes it very easy to make use of pattern matching with custom defined types.

OCaml also introduces the notion of polymorphic variants which are more flexible and do not require an explicit type declaration.

Listing 2.2: Polymorphic Variants

```
# let card = `Card ("spades", 1);;
- : val card : [> `Card of string * int ] = `Card ("spades",
    1)
```

Listing 2.2 shows how a back-tick can be used to define a polymorphic type, and OCaml has automatically inferred a type. The > symbol acts as a lower limit on the tags that the variant `card` can take, i.e. it can have the tag `Card` or any other unspecified tag. When dealing with variant types as *parameters*, the < symbol often appears in the type signature to show that the parameter can belong to only a subset of given set of tags. T The absence of both of these symbols indicates that a variant can take any value from the given set of tags, rather than a superset or subset.

Modules and Functors

Modules provide a way of grouping together related code in OCaml. They can be thought of as similar to traditional namespaces, although there a few key differences. OCaml also lets you define module type signatures which modules have to conform to.

Listing 2.3: OCaml Modules and Functors

```
module type Math = sig
  type t
  val add: t -> t -> t
  val subtract: t -> t -> t
end

module IntegerMath : Math with type t = int = struct
```

```
  type t = int
  let add int1 int2 = int1 + int2
  let subtract int1 int2 = int1 - int2
end
```

Listing 2.3 is a simple example that defines a module signature `Math` for adding and subtracting a custom type. The module `IntegerMath` is a module which adheres to this signature. The syntax `with type t = int` serves the purpose of letting the compiler know that the type `t` is externally visible. Whilst this example uses the trivial example of integer maths, it is easy to imagine this extending to, for example, matrices or sets where these functions are not part of the language.

Modules are really useful for allowing the effective division of code into isolated units, but on their own, they are slightly inflexible. Maybe a developer would want to abstract some lower level details of code used by a module. In this case, she would have to create a whole new module for each possible implementation of this abstraction. An example of this could be a datastore that uses either an in-memory or on-disc format for storing data. Functors allow modules to be created from other modules.

Listing 2.4: Ezirmin Log Module

```
module Log(AO : Irmin.AO_MAKER)(V: Tc.S0) = struct
  ...
end
```

Listing 2.4 is an example from the Ezirmin codebase and shows how to define a functor. The parameter `AO` is a module used for creating append only stores, and the parameter module `V` defines a data type. The result of this is a functor, `Log`, which can be used to create a Log module with either an in-memory or on-disc backend.

Development Environment

When developing a large scale system with OCaml, there are a couple of build systems available to use. 'jbuilder' [4] is one of these systems which is becoming increasingly popular and is used daily by hundreds of developers. jbuilder allows the developer to specify arbitrary directory structures containing executables, libraries and more. I set up my project to compile Logan, which includes the interface for running both Leader and Participant nodes. I also defined multiple executables for running both the Leader and Participants in example and test cases. I used GNU Make [3] to invoke jbuilder which allowed me to easily build and run any executables from the root of the directory.

In order to ensure that the project would always build, I set up a continuous integration workflow using Travis-CI. This ensured that whenever I pushed any updates to my GitHub repository, Travis would attempt to build the system and would notify me whenever there were any errors during the build. This was particularly useful during the Evaluation

stage when I was installing my project on multiple remote machines for testing. Over the development period, Travis flagged issues such as dependency resolution failures or out of date build scripts and I was able to fix these at the point when they first occurred, rather than weeks later when they would have caused many issues during remote installation.

Lwt

A library which I spent some time familiarising myself with was Lwt [5]. Lwt provides a way of interacting with threads in OCaml, although in Lwt they are known as 'Promises'.

Listing 2.5: Lwt Promises

```
val Lwt.return : 'a -> 'a Lwt.t
val Lwt_main.run : 'a Lwt.t -> 'a
val Lwt.bind : 'a Lwt.t -> ('a -> 'b Lwt.t) -> 'b Lwt.t
```

Listing 2.5 shows the basic functions for creating, running and combining threads. The above type `'a Lwt.t` refers to a thread which will eventually terminate with a value of type `'a`, and follows the well established Monad design pattern. `Lwt.return` is a function that takes a value and will create a promise that immediately returns with this value. This is useful when inserting static or precomputed variables into Lwt pipelines. `Lwt_main.run` is the dual of `Lst.return` and will run a Lwt promise until completion and then return its value. This is useful when retrieving values from an Lwt pipeline. Finally, `Lwt.bind` (or the infix notation >>=) will pass the result of a Lwt promise to a function returning another Lwt promise. This is useful for chaining together Lwt promises in a pipeline.

2.3 Requirements Analysis

During the preparation stage of my project, I spent some time analysing the requirements that would be suitable for Logan. This proved a good way of guiding the progress of the project and making sure that I solved all the problems that I set out to. Here, I will set out the criteria that I decided upon before starting development on the project.

2.3.1 Data Structure

It is required that any node in a Logan network should be able to add items to and view items in a blockchain. The data in the blockchain will be referred to as either *blocks* or *transactions*[1]. From a Participant, it should be possible to transactions of arbitrary type. It should also be possible to view an ordered list of all transactions which currently exist in the blockchain. The latency of adding transactions should not increase as the size of the blockchain increases because it is important the library will cope well for systems

[1]Whilst blocks in this project shall be referred to as transactions, most blockchain systems use blocks which contain multiple transactions

with indefinitely large blockchains.

Listing 2.6: Blockchain Specification

```
module type I_LogStringCoder = sig
  type t
  val encode_string: t -> string
  val decode_string: string -> t option
end

module type I_Validator = sig
  type t
  val init: t list -> unit Lwt.t
  val filter: t list -> t list Lwt.t
end

module type I_Config = sig
  type t
  module LogCoder: I_LogStringCoder with type t = t
  module Validator: I_Validator with type t = t
  val remotes: string list
end

module type I_Blockchain = sig
  type t
  val add_transaction_to_blockchain: t -> [> `Error | `Ok]
     Lwt.t
  val get_all_transactions: unit -> [> `Error | `Ok of t
     list] Lwt.t
  val get_transactions: int -> [> `Error | `Ok of t list]
     Lwt.t
end

module Make(Config: I_Config): I_Blockchain with type t =
   Config.t = struct
  ...
end
```

Listing 2.6 is the initial technical specification that I used to define Logan, and the functions complete the following operations:

- `I_Config` contains information which is required to run the blockchain, including `remotes` which is a list of other participating nodes.
- `I_LogStringCoder` is a module that allows the user to specify arbitrary types to be stored on the blockchain, so long as they can be encoded to (and decoded from)

a string.

- `I_Validator` is an interface which is defined by the library user. Validation is performed by initialising a `Validator` module with the transactions already committed to the blockchain. From then on, `Validator.filter` will take a list of requested transactions and will return a subset of them which are the valid transactions. The `Validator` can then assume that these transactions have been committed for future calls to `filter`.

- `Make` is a functor which accepts a configuration module and will return a blockchain module.

- `add_transaction_to_blockchain` adds a user defined type to the blockchain and then returns a polymorphic variant type containing information about whether the operation was successful. This will return an error in the case that the transaction is not validated.

This specification differs slightly from the final specification for Logan because it does not take into account the difference between Leader and Participant nodes. The specification was deliberately designed without focus on consensus so that the focus was on the external functionality rather than the internal implementation. The final interface, while different, provides exactly this functionality.

2.3.2 Consensus

Ensuring that Logan maintains consensus was a very crucial part of this project. The actual design and development of the consensus algorithm was completed throughout the duration of the project and involved a lot of research into other consensus mechanisms. However, during the requirements analysis phase of the project I set out some goals for the final implementation. These goals were laid out in order to help drive the design and development of the consensus mechanism. I decided on the following requirements:

- The consensus mechanism must guarantee strict consistency in the blockchain state. Although strict consistency could add large overhead costs, it is necessary for some applications and so is an important part of the final project.

- The consensus mechanism must be scalable. Within the scope of this project, it should be possible for the blockchain to be shared by 3 or more nodes in a network. Latencies will inevitably grow at least linearly in the number of nodes in the network. Logan's performance should demonstrate only modest descreases in performance for increases in nodes in a network.

Chapter 3

Implementation

Over the course of this project, I have successfully built Logan, a blockchain library which can perform custom transaction validation. Logan uses a leader-based consensus protocol inspired by Raft also uses the concept of a 'transaction waiting room', or *mempool*, which is used by Bitcoin. Section 3.1 gives a general overview of Logan's system architecture. Section 3.2 describes the blockchain data structure that I have used, and justifies why it can be considered a blockchain. I will also describe two bugs that I encountered in these codebases and the fixes that I implemented for each. Finally, Section 3.3 gives a description of the research I completed when designing Logan's consensus mechanism. I will first present a naïve approach to consensus and explain why this is flawed and will cause transactions to be committed out of order. I will then present an improved mechanism, which Logan uses, that guarantees correct ordering of requested transactions.

3.1 System Architecture

Logan's goal is to allow multiple nodes in a distributed system to share a blockchain data structure. Nodes in this system should be able to add transactions to the blockchain and read the entire history of previous transactions. I shall refer to blocks in the blockchain as 'transactions' as if they contain only a single transaction, however they easily can be made to contain many transactions because the datatype of transactions is arbitrary and specified by the developer. Logan aims to achieve consensus by using a leader-based approach where a *Leader* node can accept transactions from *Participant* nodes and commit them to the blockchain for everyone to see. Participants view the blockchain by retrieving the latest copy over an SSH connection with the Leader. Participants request for transactions to be committed by writing to their local mempool. The Leader periodically polls the Participant mempools, again over an SSH connection, to find new transactions to add to the blockchain. When a poll finds new transactions, it will use the Validation module to filter all the valid transactions which are then added to the blockchain. It is also possible for a Participant node to exist on the same physical machine as the Leader, in which case the Leader sources updates directly from the mempool, rather than over an SSH connection. Figure 3.1 shows how the architecture for this system is organised.

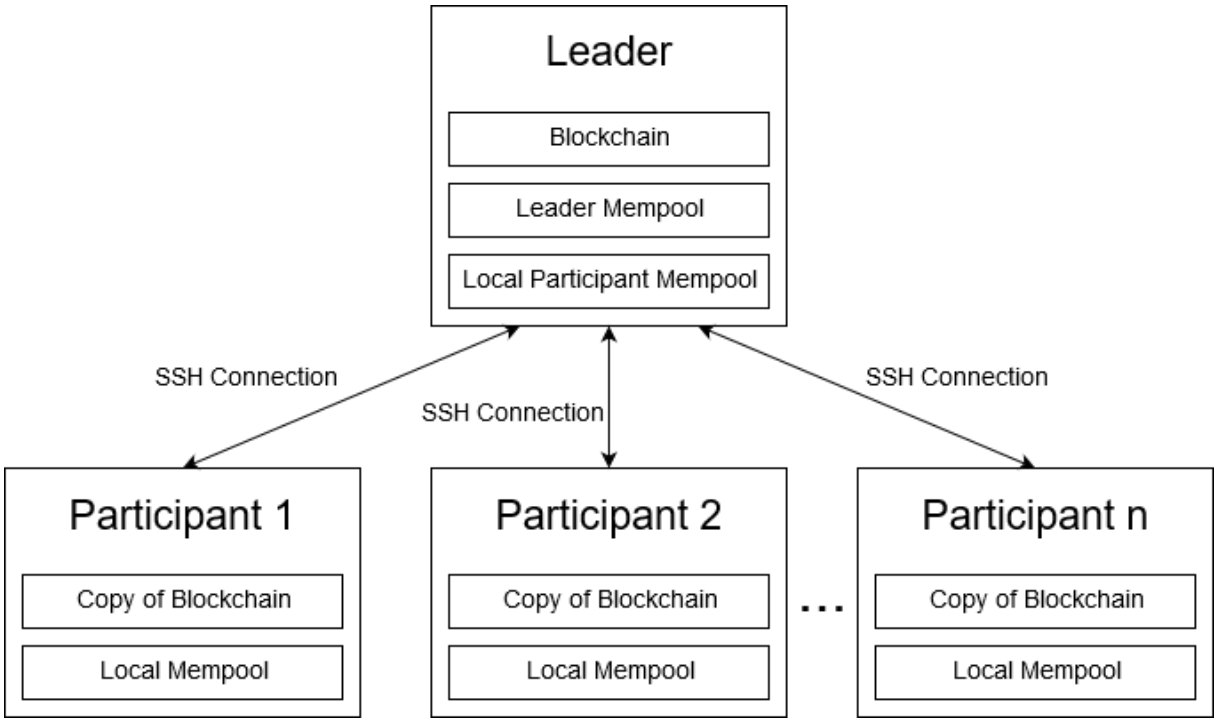


Figure 3.1: Logan system architecture overview

3.2 The Blockchain Data Structure

Irmin [6] is a library for OCaml providing datastore functionality using a Git backend. Ezirmin is a wrapper around Irmin that provides a simple log data structure which I have used as a blockchain. In order to justify that this data structure can be considered a blockchain, I took some time to investigate its semantic properties. Whilst there is no universally agreed-upon definition of a blockchain, I have used the following two criteria to define the blockchain:

1. Data is stored in 'blocks'.
2. Blocks are ordered in a tree structure where each block contains the hash its parent block.

This definition deliberately makes no mention of consensus, and exists purely to justify the validity of the blockchain data structure.

3.2.1 Irmin

Irmin exposes three main structures as shown in Figure 3.2: The Block Store, the Tag Store and the Irmin Store. For the reader familiar with Git, these can intuitively be thought of as objects/commits, branches and repositories respectively.

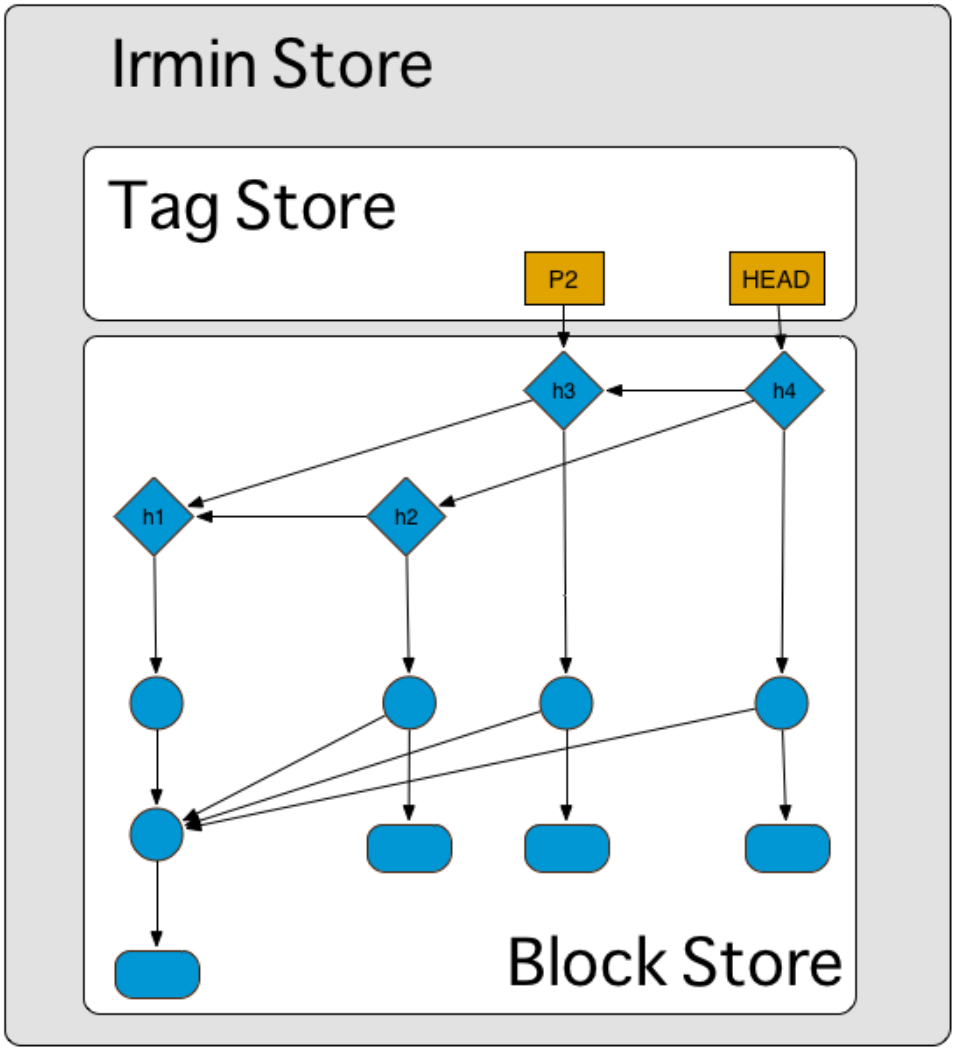


Figure 3.2: Architecture of an Irmin Store

The Block Store

The Irmin Block Store is a heap of immutable blocks. Rather than being addressed by a physical address, these blocks are addressed by the hash of their content. Because the Block Store is content-addressable, once blocks are added, their content can never be updated. Instead, updates to Irmin data structures are always made by adding new blocks to the store.

The Tag Store

Any immutability in Irmin data structures derive from *tags*. Tags provide a way of indexing into any block in the Block Store. This concept is similar to that of branches or references in Git, which provide a way of indexing into a particular commit in the Git history. Because blocks are immutable, any changes to an Irmin data structure can only be visible if a tag is updated to point to a different block. In this report, I will refer to the HEAD tag, which indexes into the latest recognised log item. When Irmin merges data structures, it creates new blocks and updates the relevant tags to point to these, however, the old blocks remain in the block store.

Irmin Stores

An Irmin Store is simply the combination of a Block Store and a Tag Store. Considering the criteria set out earlier for a data structure to be considered a blockchain, Irmin Stores satisfy the first criteria, i.e. that data is stored in Blocks. Indeed, they also come close to satisfying the second criteria, as all blocks are indexed by the hash of their content. However, this is not quite enough, as blocks are not required to contain hashes of parents in a chain. In order to satisfy these criteria, I looked to Ezirmin which provides a log data structure.

3.2.2 Ezirmin

Ezirmin is a library that provides a simplified interface to the Irmin library. It is designed to provide a interface to Irmin without functors, but useful defaults instead. Importantly, it has a built in log data structure which uses Irmin's append-only store, saved on disk in the Git format.

Listing 3.1: Ezirmin Log

```
module type FS_Log = sig
  type elt
  type cursor
  val append : ?message:string -> branch -> path:string list
      -> elt -> unit Lwt.t
  val get_cursor : branch -> path:string list -> cursor
     option Lwt.t
  val read : cursor -> num_items:int -> (elt list * cursor
```

```
      option) Lwt.t
  val read_all : branch -> path:string list -> elt list Lwt.
      t
  ...
end
```

Listing 3.1 gives some of the interface for an Ezirmin log which uses a file system backend. The log allows for items to be appended to and read from a log, and the concept of cursors is simply an abstraction for Irmin tags. In particular, the function `read` will read from the item indexed by a cursor, and will return a new cursor for the next unread log item, alongside the result of the read.

Ezirmin Log as a Blockchain

Logan uses an Ezirmin Log as an implementation of a blockchain, and to verify that this is a valid data structure to use, I investigated the implementation of Ezirmin Logs. Ezirmin Logs use Irmin blocks to store log entries, and therefore satisfy the first criteria for being considered a blockchain. To verify that they also satisfy the second criteria, i.e. that blocks are ordered and contain the hash of their parent, I looked at the implementation of log items.

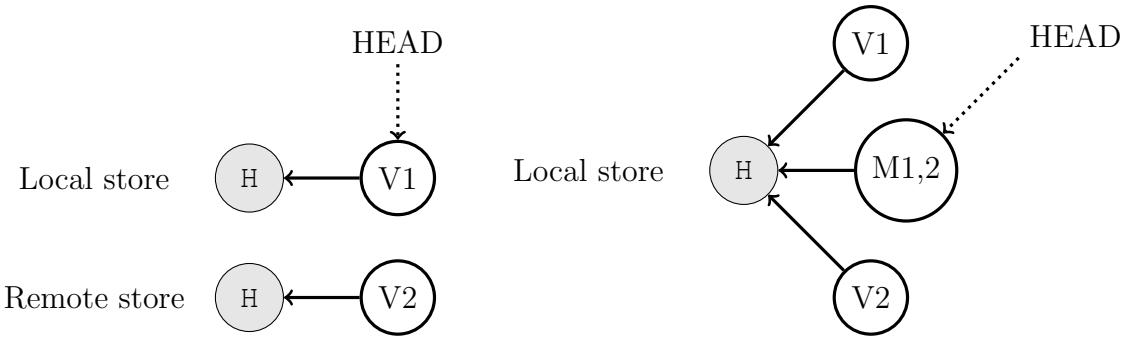
Listing 3.2: Ezirmin Log Item

```
type log_item =
{ time    : Time.t;
  message : V.t;
  prev    : K.t option}
```

Listing 3.2 is taken from the Ezirmin Log implementation and shows how each log item has a `prev` value which is a key pointing to its parent log item. This imposes an ordering of log items, and means that any changes to previous log items will not be observed, as they will simply create a new block with a new address. Merging Ezirmin Logs causes new `Merge` blocks (as opposed to `Value` blocks) to be created that collate multiple log entries from divergent log histories. These entries are ordered by their timestamps and this means that after a merge, items may appear to have been inserted into the log history. This is not desirable behaviour for a blockchain, so I have ensured that the blockchain data structure maintained by Logan is never merged with other structures, only appended to. Consequently, each blockchain transaction will exist in its own `Value` block and will never removed or reordered in the blockchain. Therefore an Ezirmin Log satisfies both of my conditions to be considered a blockchain.

Merging Ezirmin Logs

Aside from using an Ezirmin Log as the blockchain data structure for this project, I also use a *mempool* log to retrieve updates/transactions from remote mempools. This retrieval process uses the `Ezirmin.Repo.Sync` module to merge new log items into



(a) Two Ezirmin logs diverging from a shared history, `H`.

(b) After the local log has merged changes from the remote log, the HEAD tag now point to a merge item, `M1,2`, which is generated by the custom merge function.

Figure 3.3: Merging Ezirmin Logs

the Leader's mempool. Under the hood, this module uses the git protocol to pull updates over an SSH connection.

Irmin allows for the developer to specify custom merge strategies. Whenever a Irmin Store merges changes from a remote Store, Irmin will do one of two things:

1. If the newly discovered blocks do not have a divergent history to the current store, they will simply be added to the Block Store. In other words, if only new blocks have been added in the remote store, these will be added to the local store.

2. If the remote and local stores have divergent histories, then a custom three way merge is made using the heads of each store and their latest common ancestor. This merge is defined by the developer.

Figure 3.3 shows how a local Ezirmin Log performs a merge from a remote log with a diverging history. Before the merge, the local store has a `Value` block, `V1`, but the remote store has a different `Value` block, `V2`. After the local store has merged changes from the remote store, a new `Merge` block, `M1,2`, has been created which contains both histories as defined by the custom merge function. Blocks `V1` and `V2` still exist in the block store, but the HEAD tag has been updated to point to `M1,2`.

Ezirmin Bugs

Whilst Ezirmin provides a set of very desirable semantic properties, it is not a widely used library. Consequently, during the course of the project, I encountered bugs in the Ezirmin codebase which I had to debug and fix. In this section I shall outline the two bugs that I encountered the changes I made to the Ezirmin codebase to fix them.

The first bug occurs when updates are merged into a log over a network.[1] Irmin uses Git as a backend and in order to merge updates over a network, the `git pull`

[1]The issue on GitHub can be found at https://github.com/kayceesrk/ezirmin/pull/7

command is used. In Git, objects are stored as blobs on the leaves of a tree and `git pull` works by retrieving all blobs at the leaf nodes of the tree for a given branch. However in Ezirmin, blobs can contain pointers to other blobs, and `git pull` will not know to retrieve these blob too. This will cause an error when Ezirmin tries to read these objects which do not actually exist locally. The solution that I implemented to this problem is to use an internal branch which uses an Irmin Store to explicitly track every blob in the log. Items are added to this Irmin Store on the internal branch every time they are added to the log. This operation inserts these blobs into the git tree such that if `git pull` is performed on the internal branch, all the blocks required by the master branch will be retrieved.

The second bug that I encountered derived from the merge behaviour of Ezirmin logs when there are more than two remote machines merging updates from each others' log.[2] The issue is that when a `Merge` block is created, it may contain log items which, on other machines, are stored as `Value` blocks or as different `Merge` blocks. The result is that Irmin sees these blocks as different, and Ezirmin does not perform any checks to detect or remove duplicate log items occuring in different contexts. This behaviour resulted from complex sequences of merge operations and so was difficult to reproduce, but in the worst case caused the size of the blockchain to grow exponentially with the number of merge operations performed. In one case, a log with 40 unique log items grew to be greater than 120,000 log items in size due to duplicates. I implemented a fix by removing duplicate log items when a merge operation takes place. Duplicates are found by comparing the log item timestamps, and whilst this does not guarantee their individuality, the timestamps have such significant precision to make this highly unlikely.

3.3 Achieving Consensus

Building consensus was by far the most important part of work completed for this project. In order to guide the design of the consensus mechanism, I completed an extensive amount of research on existing algorithms. This section will briefly summarise this research and my conclusions about their suitability for Logan. The resulting design for consensus is a leader based approach, where Participants can commit transactions to a mempool. The Leader polls these mempools for updates, which are validated and then committed to the blockchain. This blockchain can be read by all Participants, and provides a definitive source of ordered, committed transactions.

3.3.1 Existing Consensus Algorithms

Proof of Work

Proof of Work (PoW) is a deceptively simple consensus mechanism, used by most cryptocurrencies to avoid the double spending problem. Transactions are contained within blocks which can be broadcast out to the network of participating nodes. Whenever a

[2]The issue on GitHub can be found at https://github.com/kayceesrk/ezirmin/pull/8

block is received by a participating node, the node checks that the block contains a proof of computational work done. This proof usually takes the form of a random sequence of data (this is known as a nonce) appended to the end of the block, causing the block's hash to be prefixed with a set number of 0s. This acts as a proof of computational work, because the data appended to the end of a block can only be found by a brute force method called mining, but can also be verified easily by simply computing the block's hash.

Why is this useful? Well, this makes a guarantee on the validity of the blockchain based on the simple assumption that more that 50% of the workforce is genuine. That is, ifm we assume that the longest chain of blocks is the correct one, then in order to create a sequence of biased transactions, we would have to create a chain longer than the correct one. This would require an equal number of 'Proof of Work's, which, due to the random nature of block mining, would require more than 50% of the workforce. Whilst it may be possible to maintain an equally sized chain with less that 50% of the workforce for a short period of time, the chances of this decrease rapidly as time passes. All in all this means that the longer a block has been in the chain, the more likely it is that the block is valid.

Whilst this forms a very effective mechanism for achieving consensus, there are also some considerable downsides to using a PoW approach to consensus. Firstly, there is a huge amount of computational work wasted in the process of mining. The effect of this is energy consumption [1] and wastage to a level which can cause serious environmental harm. PoW also does not generalise well outside of the scope of cryptocurrencies. It assumes no trust in any participants which may not be a suitable model for an application. Additionally, it also assumes that miners can be rewarded, usually with cryptocurrency, but this incentive is ad hoc and may not exist in other applications.

Mempools

Mempools are an important part of the design of Bitcoin, and whilst they are not inherently linked to the Proof of Work consensus algorithm, they are worth investigating. When a Bitcoin transaction is made, it is first written into what is known as a Mempool. This transaction can then be seen by participating miners, who can then choose to put this in the next block that they mine. This is significant, as it provides a 'waiting room' for any transactions that have not yet been validated.

Proof of Stake

The Proof of Stake (PoS) algorithm is used by some cryptocurrencies and works by randomly allowing participants to create (or 'forge') a single block. However, the probability that a participant is chosen to 'forge' a block, is weighted by its stake, such that participants with higher stakes in the blockchain are more likely to be chosen to forge a block.

So, why is PoS desirable? By far the most convincing reason for using PoS over PoW, is that there is no need to waste lots of energy in the process of mining. This hugely

reduces the environmental impacts of scaling a PoS network. Using PoS also allows trust to be distributed according to an arbitrary heuristic which can be desirable property in some applications.

One of the flaws of PoS is that it does not have such a strong deterrent against attacks. With PoW, attacks require huge amounts of computational power and it is likely that to create an attack, you would have to spend more on hardware than you would gain. PoS does not have this same built in mechanism, and so there have been many suggested schemes for increasing the safety of PoS networks. For example, it is possible that participants should need to pay some form of deposit before forging blocks, which can be slashed if they break any rules. PoS also suffers from the same problem as PoW in that it does not generalise well. It is another example of a consensus mechanism designed for networks with a strong notion of Stake and with minimal trust in any individual participant.

Paxos

Paxos is a family of consensus protocols which can be used to guarantee consistency in distributed systems. It was first proposed in a paper by Leslie Lamport in 1998 [8], although the paper was first submitted in 1990. Named after a fictitious civilisation living on the island of Paxos, the algorithm puts forward a way for any number of nodes to propose and agree on a value. Participating nodes belong to various roles, one of which is known as a 'Proposer' or Leader.

The main part of the algorithm is split into two sections, propose and accept. In the first stage, a Proposer decides that it wants to propose a value and then broadcasts out a 'Prepare' message to a quorum of 'Acceptors'. Acceptors will then decide if they want to make a 'Promise' which is a commitment to accepting that proposal in the future. If a quorum of promises is received by the Proposer, then it will assign a value to its proposal and will again send an 'Accept Request' out to a quorum of Acceptors. Finally, if enough 'Accept' messages are received then the Proposer can be certain that the value has been agreed upon by consensus.

This algorithm has been proved to be consistent but it also has a lot of complexity and a lot of different variants. The combination of different roles and states makes it easy to implement incorrectly. It is also important to consider that Paxos describes a 'family' of algorithms, with some parts left deliberately unspecified, and choosing how to implement these is not a trivial decision. The final issue with Paxos is that it cannot guarantee progress. Whilst it enforces conditions which make it unlikely that progress will not be made, it is still theoretically possible for the mechanism to stall indefinitely.

Raft

Raft [10] is an algorithm that was designed to be equivalent and as efficient as Paxos, however, it also places a much greater emphasis on comprehensibility. It uses the notion

of a *strong leader*, which is an elected server that has total control over which log entries are accepted. There are two other types of server, a *follower* and a *candidate*. Followers are completely passive, and only respond to requests from leaders and candidates. A candidate is a server that has put itself forward for election.

So how does the algorithm operate? Time in Raft is split into *terms* which are labelled by a monotonically increasing integer. Each term effectively signals a time period where a particular server is the leader. A term starts with a leadership election when a follower transitions into the candidate state, increments its term number and requests votes from other servers. It will wait for a majority of votes and then elect itself the leader, unless it times out or receives a message from another leader with a greater term number. Typically, these leader election processes are triggered when a follower does not receive a heartbeat message from the leader for longer than a given time. In the pathological case, the vote can be split between leaders, triggering a new election which is also split and so on. However, Raft uses randomised election timeouts to avoid this problem. Additionally, Raft also prescribes some restrictions on the servers which can be elected leader so as to avoid newly elected leaders overwriting previously appended log items.

Raft is an simple algorithm that is easy to understand, but it also guarantees the Log Matching Property that if two logs contain a log entry with the same index and term, then that entry, and all preceding entries will be identical.

3.3.2 Roles in a Logan's Approach

Logan uses a mempool and a simple leader-based algorithm to maintain consensus, and in this section, I will describe this approach. Because I have implemented a centralised algorithm, my specification has differed slightly from that presented in the Requirements Analysis to take into account differing functional requirements for a Leader and for a Participant. As I am assuming that all Participants can be trusted, it is possible to use a leader-based approach without introducing security issues such as the ones tackled by the Proof of Work mechanism. This approach also reduces the potential complexity an overheads of implementing completely decentralised consensus. I have introduced the notion of validation which allows the Leader to accept or reject transactions depending on arbitrary conditions defined by the developer. Finally, Logan uses the notion of a *Replica* which is a machine that performs some of the Leader's functionality to improve fault tolerance. Fault tolerance will not be described in detail in this section but instead in Section 3.4.

Leaders

Logan uses the notion of a *Strong Leader* similar to that used by the Raft protocol. The Leader is chosen statically in order to reduce the complexity of implementation, and it also has a statically defined list of `remotes` which specifies the location of all Participants. The Leader is a node which will never actually request transactions to

be added to the blockchain, its role is simply to periodically read requests from the mempools of Participants, validate them, and then add them to the blockchain. This blockchain can be read by any Participant, and is treated as the empirical source of which transactions have been committed and in which order. That is, any two nodes that read a copy of the blockchain, will always agree on content and ordering of transactions in the blockchain, up until the end of the shortest copy.

Listing 3.3: Leader Specification

```
module type I_LeaderConfig = sig
  type t
  module LogCoder: Participant.I_LogStringCoder with type t
     = t
  module Validator: I_Validator with type t = t
  val remotes: string list
end
module type I_Leader = sig
  val init_leader: unit -> (unit -> unit Lwt.t) Lwt.t
end
module MakeLeader (Config: I_LeaderConfig) : I_Leader =
   struct
  ...
end
```

Listing 3.3 is the specification of the Leader module, which differs slightly from the specification presented in the Requirements Analysis. In particular, `init_leader` performs an initialisation step, and then returns a function which, when executed, will actually start the consensus process.

Participants

Participants, in contrast to Leaders, can request transactions to be added to the blockchain. This is done by writing a transaction to a local mempool, which is then read by the Leader.

Listing 3.4: Participant Specification

```
module type I_LogStringCoder = sig
  type t
  val encode_string: t -> string
  val decode_string: string -> t option
end
module type I_ParticipantConfig = sig
  type t
```

```
  module LogCoder: I_LogStringCoder with type t = t
  val leader_uri: string option
end
module type I_Participant = sig
  type t
  val add_transaction_to_mempool: t -> [> `
     Could_Not_Pull_From_Remote | `Validation_Failure | `Ok]
      Lwt.t
  val get_transactions_from_blockchain: int -> [> `Error | `
     Ok of t list] Lwt.t
  val get_all_transactions_from_blockchain: unit -> [> `
     Error | `Ok of t list] Lwt.t
end
module Make(Config: I_ParticipantConfig): I_Participant with
    type t = Config.t = struct
  ...
end
```

Listing 3.4 is a specification that shines a light on the role of Participants and the functionality they provide. The module includes the ability to define custom data types that can be stored on the blockchain, to read from the blockchain, and to attempt to write to the blockchain by writing to a mempool. It is also important to note that Participants can exist on the same machine as a Leader, but will exist in an entirely separate process.

Replicas

Replicas are nodes which contain the latest copy of the blockchain. I shall explain later how fault tolerance can be introduced by ensuring that transactions only exist on the visible blockchain once they have been recognised by a quorum of Replicas. Replicas are useful because the Leader is a single point of failure, and if it fails, it is important to have another machine which can recover the state of the Leader just before the point of failure.

3.3.3 Retrieving Updates from Mempools

In order to see requested transactions, the Leader has to observe the mempools of the Participants. The Leader does this by pulling updates from Participant mempools and then adding them to the blockchain. However, it is important to make sure that the Leader does not miss transactions or commit transactions out of order. This is tricky because after pulling updates from many mempools the local copies will reflect each mempool at a different moment in time. In this section, I will give a naïve method for retrieving updates and I will show how this method is flawed. This will then motivate the design of a more complex approach that ensures all transactions are committed in order. In both approaches, the Leader will continuously poll mempools for updates and commit transactions that it believes to be valid.

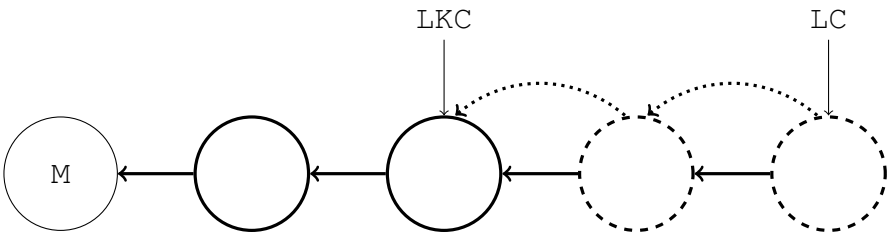


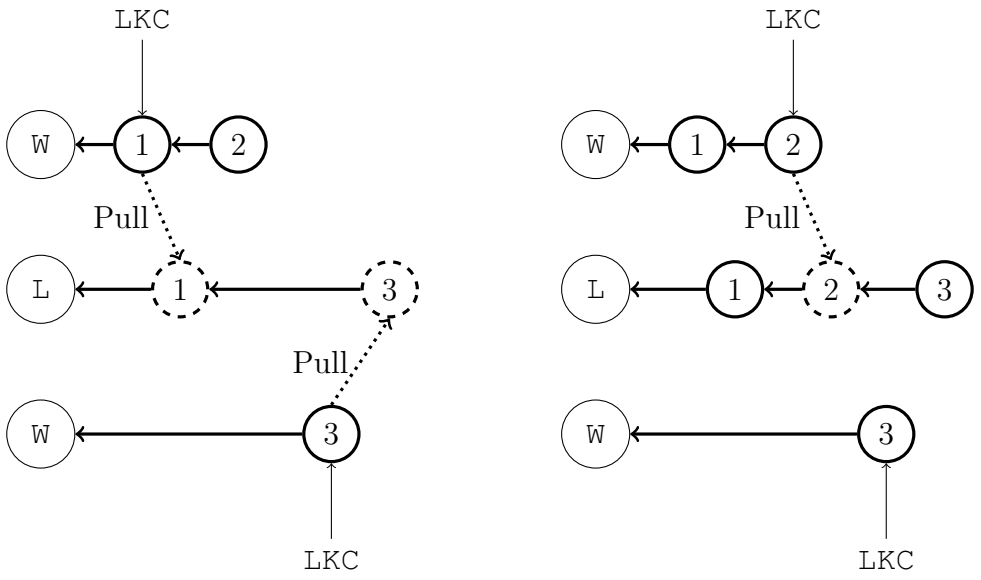
Figure 3.4: Retrieving Updates from a Single Participant Mempool

A Naïve Approach

The first approach to retrieving updates from Participants is to pull updates from all the Participants into separate branches of the Leader's mempool. It is possible view newly retrieved updates for each mempool by maintaining a *Latest Known Cursor* that points to the latest transaction seen by the Leader, i.e. in its last poll. Figure 3.4 shows how the *Latest Cursor*, `LC`, to the latest item in the mempool, is used to retrieve updates. In this diagram the dotted nodes represent transactions that have been pulled in the latest poll. The Leader can traverse back through the log from this cursor until it reaches the *Latest Known Cursor*, `LKC`, and then it can return the newly found updates.

Once the Leader has retrieved updates from the mempools of all Participants, it can order these transactions by their Ezirmin log item timestamp. After pulling updates from all Participants, the Leader can now see a list of newly retrieved transactions which are ordered by a timestamp. These updates can be validated and added to the blockchain, and the *Latest Known Cursor* updated to the latest message in each of the mempool.

However, this approach is fundamentally flawed and causes many updates to be seen out of order. This is because of the delays that occur when mempool updates are retrieved over the network from multiple Participants. Figure 3.5 demonstrates a typical situation where this occurs. The chains labelled `W` represent the mempools on the Workers' machines and the middle chain represents the stream of updates as seen by the Leader. These updates are shown as if they are part of a single merged mempool, although in reality they exist on isolated branches in the Leader's mempool, and are only merged by the Leader in memory. In the figure, updates from the first worker are pulled before updates from the second. Whilst the operations may take place in parallel, network latencies can cause these operations to pull updates at different times. After the pull from the first Participant has completed, the Participant may have added additional transactions. These transactions will be timestamped before the latest transaction in the second Participant's mempool, such that when the leader next polls for updates, transactions will be found which have a timestamp before the latest item committed to the blockchain. These items should have been committed first as they have an earlier timestamp. This transaction cannot be inserted into an earlier position of the blockchain because it would mean that different Participants might observe the blockchain in diverging states. It is not clear whether or not these transactions should be committed late or skipped, but ideally all transactions should be committed to the blockchain in the order that they were committed



(a) After first Leader poll. Transactions 1 and 3 are recognised by the Leader and added to the blockchain.

(b) After second Leader poll. Transaction 2 is retrieved but should be ordered before transaction 3 in the blockchain.

Figure 3.5: Missing mempool updates with a naïve consensus mechanism

at Participants. I shall refer to these transactions as *missed*.

A Complex Approach

In order to solve this problem, I first examined the nature of these *missed* transactions and noted the following properties:

1. Missed transactions must have been added whilst the Leader is pulling updates. If they had been added before the poll had begun, then they would have been retrieved in previous poll. Therefore, *missed* transactions must occur after the latest timestamped transaction from the previous pull.

2. Missed transactions must have been added at a point in time earlier than the latest update from the subsequent round of retrieved updates. If they occurred at a later point in time, then they would be correctly received in that subsequent Leader poll.

These properties naturally suggest a less naïve algorithm for retrieving updates which only adds updates that have existed for more than one poll cycle. Once the Leader has retrieved a set of updates, it will add them to a *retrieved* buffer. In the next poll cycle, the Leader will transfer the retrieved buffer to a new *to-be-committed* buffer. The Leader now retrieves new updates from mempools, as before, and will do one of two things for each newly retrieved transaction:

- If a transaction is earlier that the latest transaction in the to-be-committed buffer, the Leader will insert it into the correct position in the to-be-committed buffer based on its timestamp.

- If a transaction is later than the latest transaction in the to-be-committed buffer, the Leader will add it to the retrieved buffer.

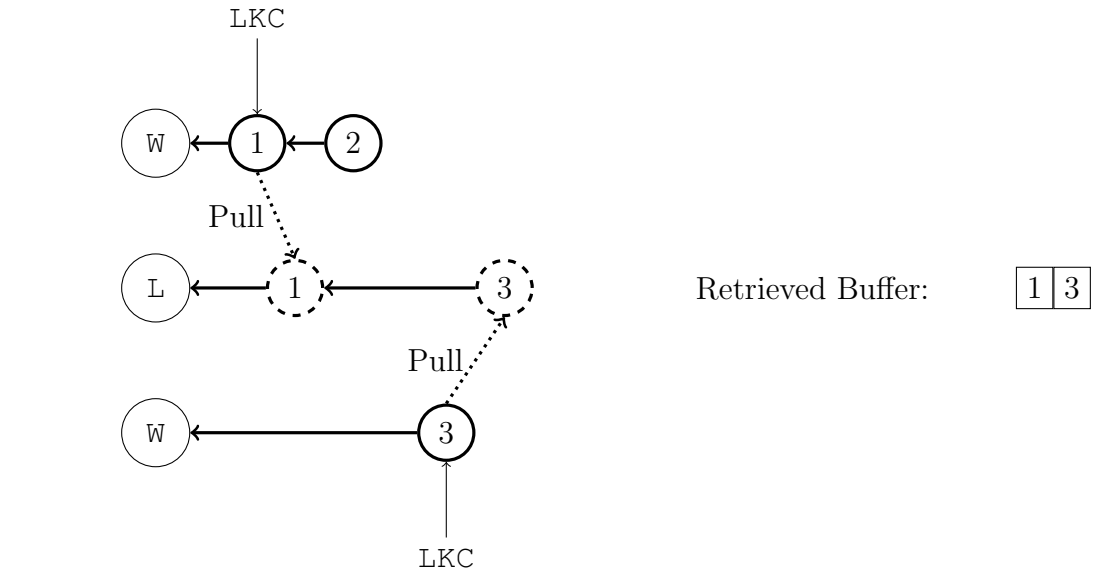
Figure 3.6 demonstrates how this approach would solve the problem presented in Figure 3.5. In a), transaction 2 has been missed whilst transactions 1 and 3 have been added to the retrieved buffer. In b), transactions 1, 2 and 3 have all been inserted into the to-be-committed buffer and will be added to the blockchain. Transaction 4 has occurred after the latest item in the to-be-committed buffer, i.e. transaction 3, and is therefore added to the retrieved buffer instead.

3.4 Tolerating Leader Failure

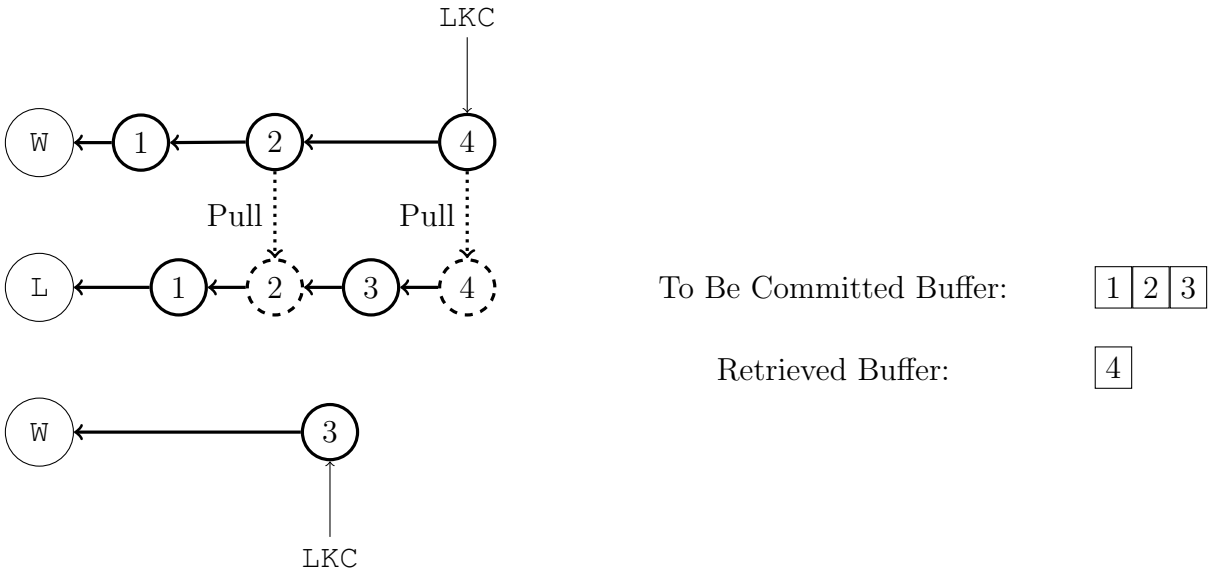
Now that I have detailed Logan's data strutures and how it achieves consensus, the only aspect of the implementation left to consider is Fault Tolerance. Because Logan implements a leader-based consensus mechanism, this means that the Leader is a single point of failure. In the event of a failure, different Participants may have views of the blockchain at different times and the Leader may have committed transactions to the blockchain which were never viewed by any Participants. Building full fault tolerance and recovery into Logan would be enough work to justify a whole new Part II Project. Instead, in this section I will show the simple mechanism that Logan uses which allows manual recovery of the blockchain state in the case of a Leader failure. This will justify that a more complex and automated approach to recovering from Leader failure is possible, and will demonstrate a good starting point for implementing such a mechanism.

To introduce fault tolerance, I will have used *Replicas*. Replicas, in computer science, are nodes which duplicate some part of another node's behaviour in order to allow recovery from failure. They can be either *active*, if they fully duplicate the actions of another replica, or *passive*, if they complete some operation and then transfer the results to another replica. The replication I will demonstrate will be passive.

In order to encorporate the notion of replica, the process of adding commits to the blockchain needs to be altered slightly. Now, before a commit is made, it needs to be recognised by a quorum of replicas. Irmin's notion of branches exposes a perfect primitive for this application. In a poll, the Leader can commit transactions to an uncommitted *cache* branch before pushing this to all replicas. If a quorum of these push operations return successfully, then the Leader can merge these transactions into the main blockchain branch to be viewed by all Participants. If a failure were to occur at any point after this merge, it is now guaranteed that any transactions in the Leader's blockchain will also exist on a quorum of replicas. This means that the system has access to sufficient information to reconstruct the contents of the blockchain before failure.



(a) After first poll



(b) After second Leader poll

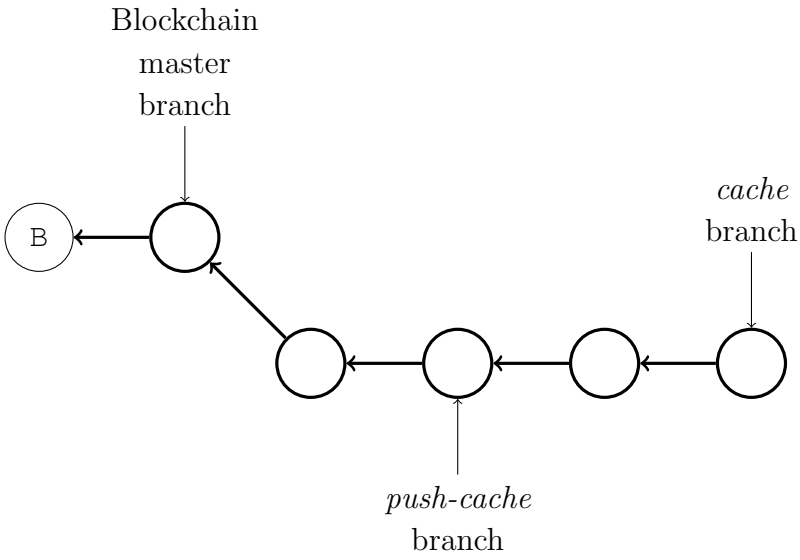
Figure 3.6: Buffering Updates in a Complex Consensus Mechanism

Listing 3.5: Naive Leader Loop

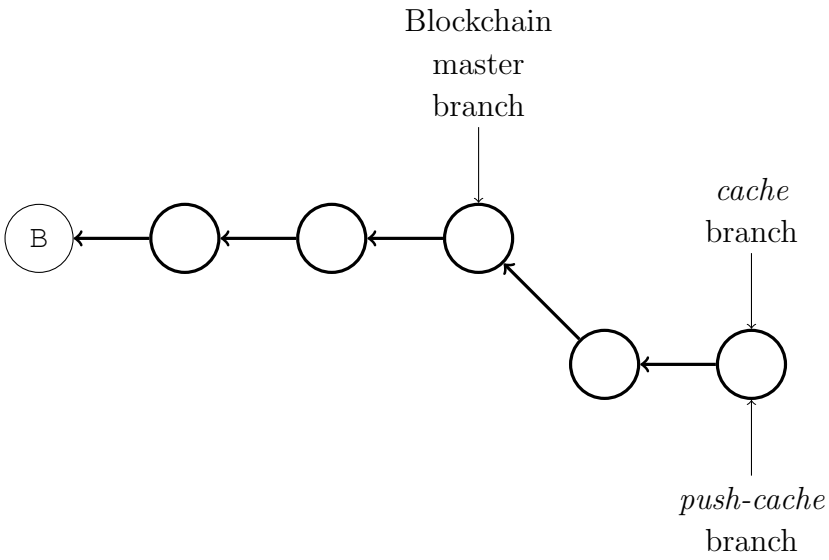
```
let rec start_leader () =
        retrieve_and_process_updates() >>=
        commit_to_temporary_blockchain_branch >>=
        push_to_replica_quorum >>=
        merge_temp_branch_into_blockchain_master >>=
        start_leader
```

Listing 3.5 shows pseudocode for the Leader's poll cycle, now encorporating replicas. One of the major issues with this approach is that is presents a bottleneck in the Leader's execution which is likely to cause significant delays.

I have implemented a better approach which starts an asychronous thread, that is in charge of pushing to replicas and merging the transactions which have been pushed. I shall refer to this as the *push thread*, and the *main thread* will now only commit items to the *cache* branch. This thread takes the blockchain cache branch and use this to create a new *push-cache* branch which it pushes to replicas. The difference between these two branches is that the cache branch will constantly be updated as new items are added, whereas the push-cache branch will remain fixed pointing to a particular transaction whilst a series of transactions are being pushed to replicas. When this push returns successfully, the thread will then merge this branch into the blockchain master branch, and loop. Figure 3.7a shows a blockchain with four transactions in the cache branch. The first two of these transactions have been recognised by the push thread and are in the process of being pushed to a quorum of Replicas. The final two transactions have been added after the push thread started its latest loop, and are therefore currently not being pushed to replicas. Figure 3.7b shows the state of the blockchain after the push thread has successfully pushed its updates to a quorum. Now the master branch includes the two new transactions, and the *push-cache* branch has been updated such that the two newest cache transactions will be pushed to the blockchains of all the replicants.



(a) During push



(b) After push

Figure 3.7: Blockchain data structure with cache for pushing to replicas

Chapter 4

Evaluation

Metrics Used

In order to evaluate the performance of Logan, I have looked at two key metrics: latency and throughput. In a blockchain system, it is important that the latency is as low as possible. Latency dictates how long a user must wait before they can be sure that their transaction has been added. It is easy to see why this is important by thinking in terms of a cryptocurrency application. In this context, long latencies will cause users to become irritated and potentially stop using your currency. Throughput is also a useful measure because it shows how well a system can cope with large loads. Again in the context of a cryptocurrency, it is important that lots of users should be able to make transactions at the same time. As with all distributed systems, these two metrics are inevitably linked, and higher system throughputs will lead to larger latencies. In the worst case, a large throughput will cause the system to become overloaded, latencies will rise and throughput will even descrease! I have looked at how the throughput applied to the system affects the latency of Logan transactions, in order to try to determine the optimal use case conditions for a Logan system.

I have also looked at how the latency of adding transactions varies with the size of the blockchain at a constant throughput. It is important that the system should be able to work well even when the size of the blockchain is large so that performance will not degrade over time.

4.1 Performance on a Local Machine

Figure 4.1 shows how the latency of transactions varies with throughput for a Participant node on the same machine as the Leader node. Latency stays at 10-15ms up until the system reaches about 70 transactions per second. The set of data points which lie significantly above the straight line are points where the system has become overloaded. The rate of transactions has become so high that latencies have increased and the throughput has actually decreased again. This is a really interesting result because it shows that Logan's logic functions well at throughputs of between 50 and 60 transactions per second, whilst maintaining low transaction latency. It is worth noting that this result is using

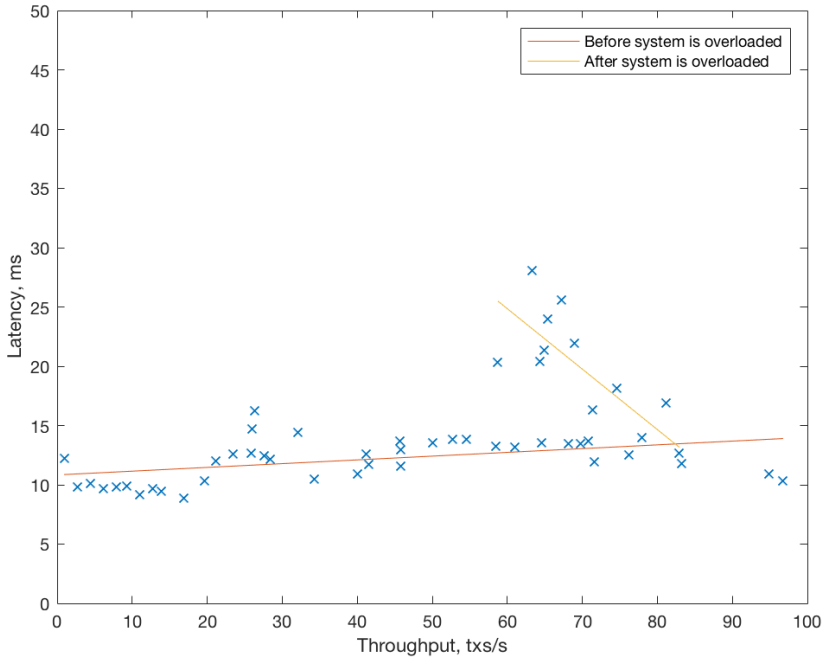


Figure 4.1: Latency variation with throughput on a single local Participant

a low performance virtual machine hosted in the cloud, with limited disk write performance. It is reasonable to expect that a more powerful machine would be able to achieve significantly better results. However, despite these machine limitations, these figures are much better than my success criteria which suggested the system should be able to cope with around 2-3 transactions per second.

Figure 4.2 shows how the latency of Logan transactions varies as the size of the blockchain increases. The Figure shows clearly that the latency of adding a transaction is constant in the size of the blockchain. This is a very significant result because it proves that the logic which Logan uses is capable of achieving constant time blockchain appends in the size of the blockchain. This is a very desirable trait as it means that Logan can be used for systems that run for indefinite periods of time, with arbitrary long blockchains. In the Requirements Analysis, I set out the criteria that Logan should be able to add transactions at constant latency and this result shows that I have been successful in achieving this criteria.

4.2 Performance on Remote Machines

4.2.1 Single Remote Participant

In order to see how the system performs when network latencies are involved, I first looked at Logan's performance when there is just one Leader and one remote Participant, i.e. a Participant on a different machine to the Leader. Figure 4.3 compares the latency

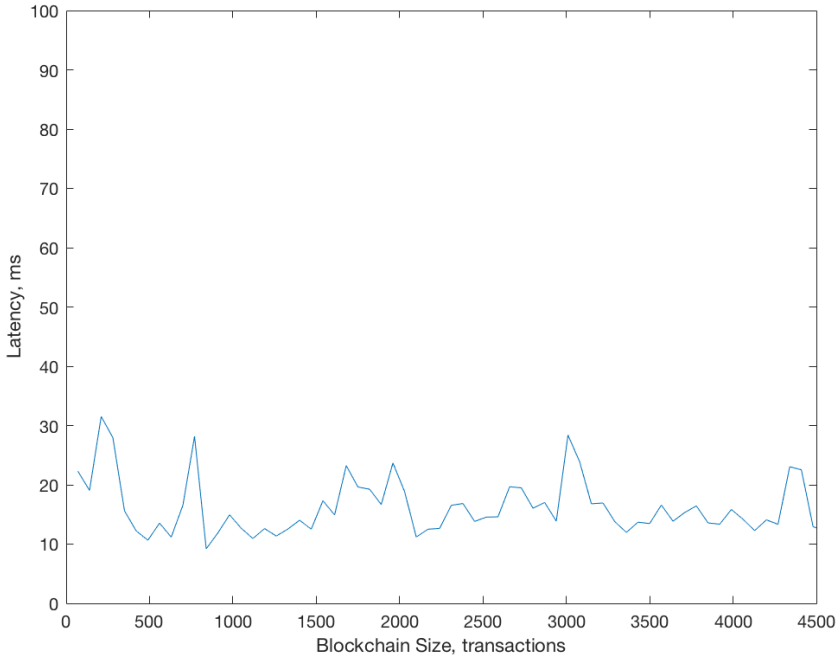


Figure 4.2: Latency variation with blockchain size on a local Participant

of making a transaction and the size of the mempool, for different throughputs. The mempool size can also be thought of as a measure of time elapsed as the thoughput is constant for each line. In this situation, mempool size is equivalent to blockchain size but has been used instead because the mempool is the structure being pulled over the network, not the blockchain. The first noticable feature of this graph is the saw-tooth shape of the latencies at high throughputs. In these cases, the flat or downward sloping segments represent transactions which were committed in the same poll cycle after being collectively retrieved over the network. Importantly, the system is perforing well up until around 20 to 40 transactions per second. There is also an obvious linear increase in latency for all of the throughputs and this is undesirable behaviour as it indicates that large mempools will accompany poor performance.

In order to examine what was causing the linear increase in latencies with mempool size, I logged the time that the Leader spent completing different tasks for each poll cycle. Figure 4.4 shows how these delays evolve over time and compares them with the number of updates that are pulled in each poll cycle. In this figure, the biggest bottleneck is obviously the time taken to pull mempool updates over the network, which increases linearly with the size of the mempool. Consequently, the number of updates that are retrieved during this period also increases and therefore the time taken adding these updates also increases. This increase in the latency of pulling mempool updates prevents the system from being able to add transactions to the blockchain in constant time. Not only this, but the latencies shown in Figure 4.4 are particularly high, so to investigate this further, I examined the latencies of using Ezirmin's `Sync` module against the native Git command line interface.

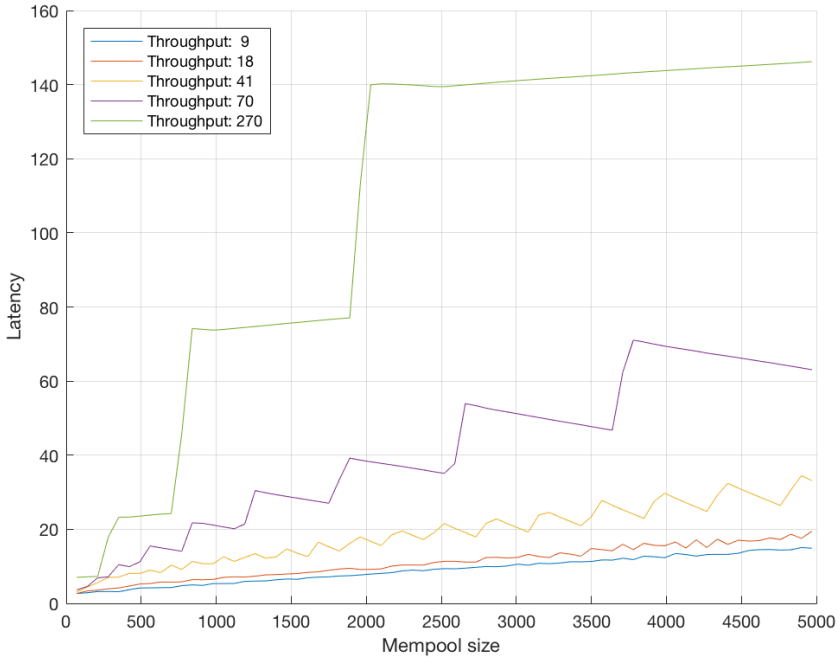


Figure 4.3: Logan latencies with increasing blockchain size

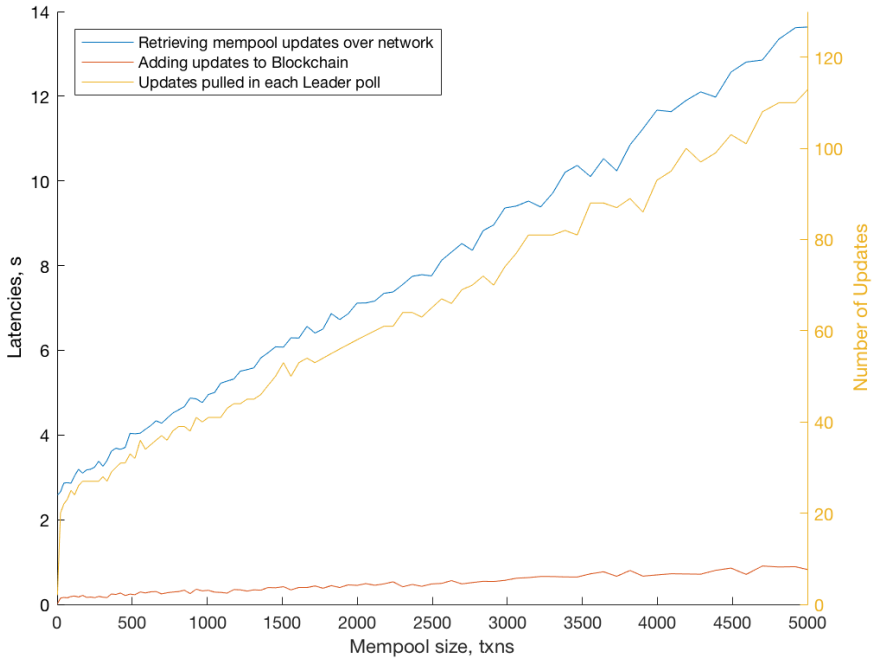


Figure 4.4: Delays incurred by Leader for every poll cycle. Throughput is constant at 10 transactions per second

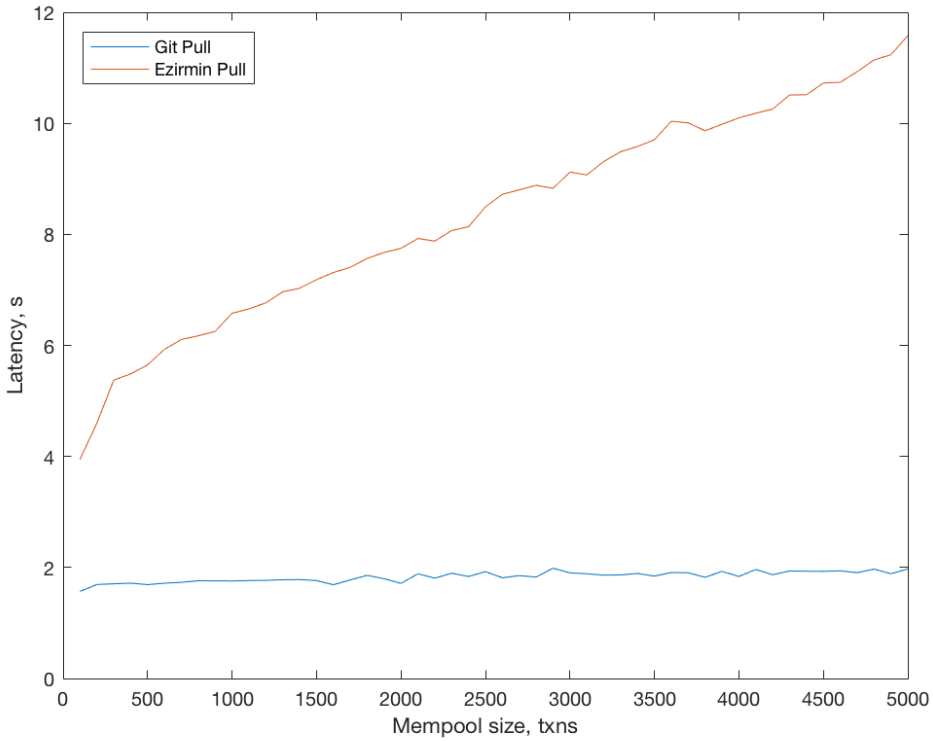


Figure 4.5: Latencies of pulling 100 transactions against mempool size

Figure 4.5 compares the latencies of the `git pull` command with the latencies of using the `Sync.pull` command provided by Ezirmin. Whilst the latency of `git pull` is linear in the size of the blockchain, the latency of using Ezirmin to pull updates increases at a much faster rate. NOTE: UPDATED GRAPH NEEDED AS CURRENT DOESN'T SHOW LATENCY FOR PULLING INTERNAL AND MASTER BRANCH.

These results provide a very bleak outlook on the performance of the Git implementation which is used by Ezirmin. In order to show the extent to which this affects Logan, I updated the synchronisation implementation to directly call Git commands via a shell. Figure 4.6 is a parallel of Figure 4.3 but using this new implementation instead. The figure shows that the system can perform significantly better by using pure Git. Specifically, the system can achieve much better constant factors in the increase of latencies with mempool size. Figure 4.7 is a parallel of Figure 4.4 that demonstrates that this speedup is due to the greatly reduced times that the leader spends pulling mempool updates as the mempool size grows. Whilst this bottleneck is much lower than before, it still takes up much more time than any other stage of the Leader's execution. This suggests that Git using SSH is still a very inefficient way of transferring updates to the leader. This could be due to many unnecessary overheads in the implementation of Git and SSH, such as the SSH handshake and the mechanism Git uses for finding new updates to be transferred between repositories. If a better protocol was used, much better performance could be achieved. Another downside of using pure Git is that it

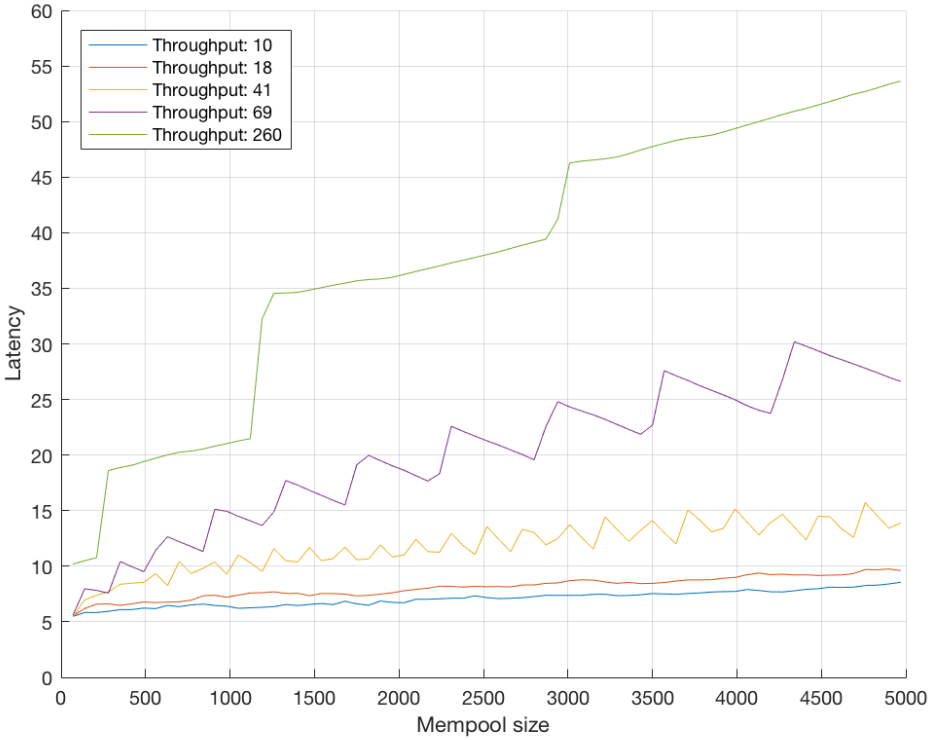


Figure 4.6: Logan latencies using pure Git instead of OCaml's implementation

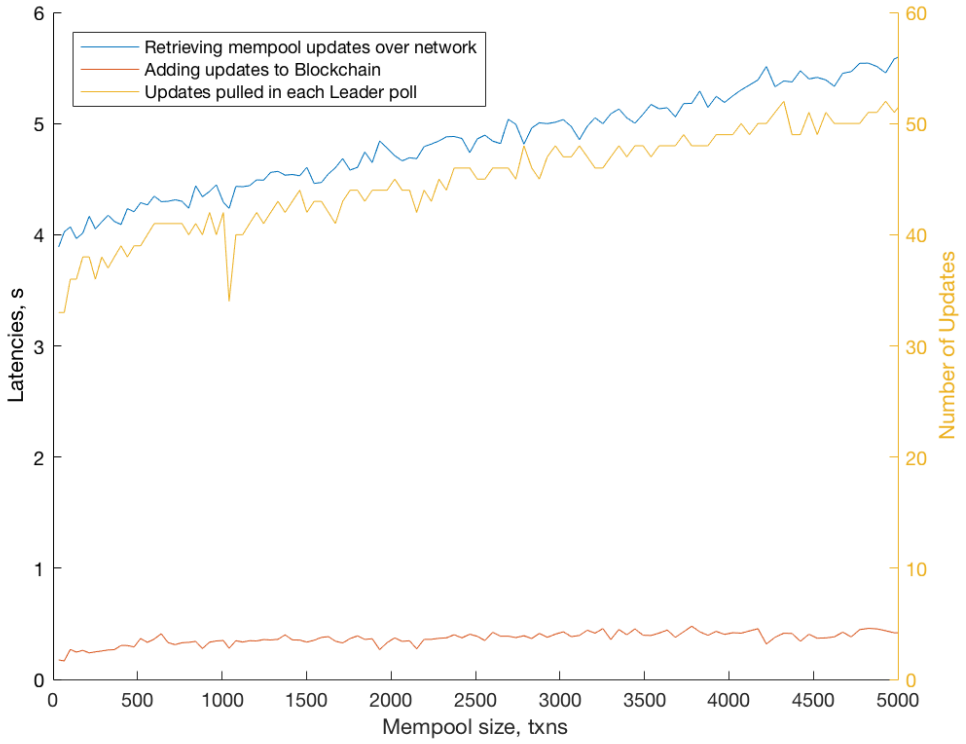


Figure 4.7: Delays incurred by Leader when using pure Git. Throughput is constant at 10 transactions per second

requires any system running a Logan Leader to also have a native installation of Git and this can be problematic when compiling to unikernel systems. This result, however, does suffice as a justification of Logan's potential performance if OCaml were to have a better implementation of Git.

**** Whilst I have now shown that Logan can perform well by using pure Git, there is still the unfortunate result that transaction latencies still increase linearly with the size of the mempool. This is due to the linear increase shown by Figure 4.5 and the fact that Logan's mempools are always growing. If the network delays were constant in the size of the mempool, then at a constant throughput the Leader should pick up the same number of updates in a cycle, and latencies should stay constant in the size of the mempool. Unfortunately this is not the case for Git's implementation which compares the contents of entire repositories on each pull to determine what updates need to be sent over the network. However, the information of what data needs to be sent can be known ahead of time and a better streaming protocol would, in theory, allow for this latency to become constant.

Additionally, the fact that mempools are always growing is not a requirement for Logan. A mempool only needs to contain the latest uncommitted transactions and could therefore take a queue data structure instead of the log data structure used by Logan. In this scenario, Participants would add transactions to their mempool queue which would then be pulled and pop by the Leader. Participants could periodically merge changes from the Leader's copy in order to reduce the size of their blockchain, and therefore the subsequent pull operations. This would also provide the desirable side effect of allowing the Participant to see when the Leader has recognised requested transactions, because they would get popped off the queue. Whilst Ezirmin provides a Queue data structure, I have not used it in this project in order to reduce implementation complexity. Using this structure would also likely decrease initial performance as it uses OCaml's implementaton of Git. Because of the bidirectional nature of merging changes (i.e. popping and pushing), it would not be trivial to use command line Git in this case.

Overall, in the case of a single remote Participant, Logan is able to comfortably reach throughputs of up to 30 transactions per second which is much above the success criteria. Latencies in this case are less than 10s which is a good result for most applications. Finally, whilst Logan does not fulfil the success criteria of having constant-time blockchain appends, I have demonstrated how this is due to the overheads from using Git, and I have justified how a better synchronisation protocol would allow Logan to achieve this goal.

4.2.2 Multiple Remote Participants

After showing that Logan can perform well with a single remote Participant, I evaluated how this performance scales to two remote Participants. Figure 4.8 shows how the latency of making transactions varies with throughput when the pure Git protocol is used. An important feature of this graph is the fact that the slope of the plots have

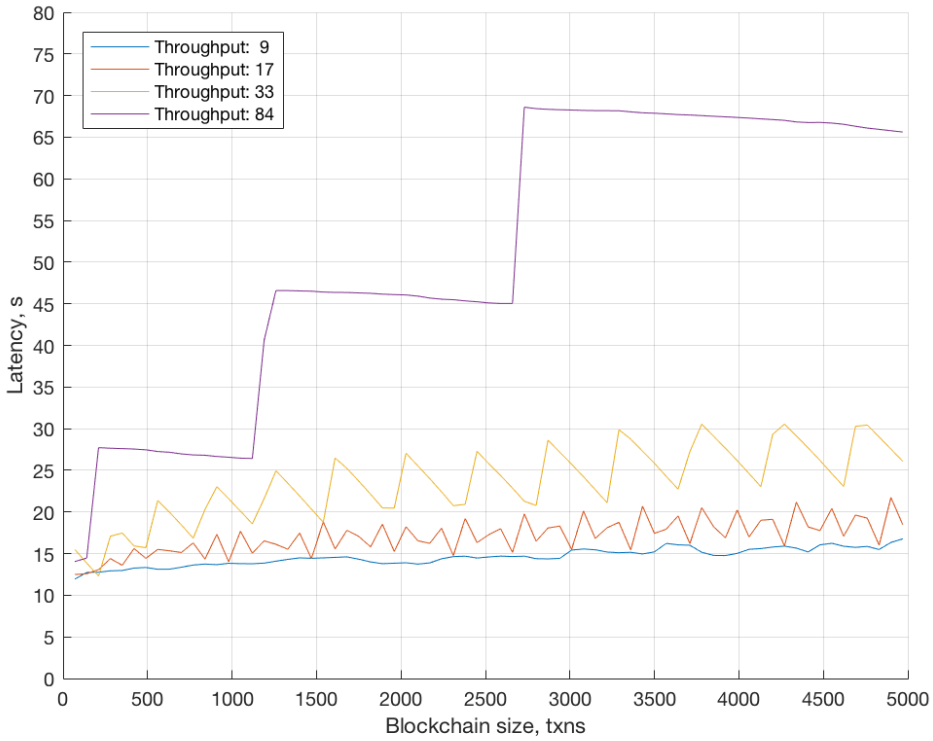


Figure 4.8: Latency variations against blockchain size with two remote Participants using pure Git

not really changed from the case with only one remote Participant. For a throughput of around 10 transactions per second, transaction latencies will still rise by around 5 seconds over 5000. The biggest difference is the fact that transaction latencies now start at around 12 seconds, whereas they were around 6 seconds before. This is because native Git does not allow for updates to be pulled from multiple different remote repositories at the same time, and therefore, all remote repositories must be pulled sequentially. Even if a protocol is used that can achieve constant time blockchain appends, this result limits latencies to be linear in the number of remote nodes in a network which is undesirable.

Figure 4.9 shows the performance of Logan when using Irmin's Git rather than pure Git. Importantly, the initial transaction latency has not increased from the case when there is just one remote Participant in the network. In contrast to pure Git, Irmin's Git provides the functionality for pulling remote updates in parallel and consequently the initial transaction latency has not doubled in this case. This indicates that the number of nodes in the network will not hinder transaction latency so long as the Leader node is computationally powerful enough to process updates from mempools in parallel.

Overall, the results from using Irmin's Git implementation and pure Git both show different trade offs in terms of initial transaction latency and increases in transaction latencies over time. However, the benefits shown by each implementation are not mutually exlusive. A better implementation of Git should be able to pull mempool updates in

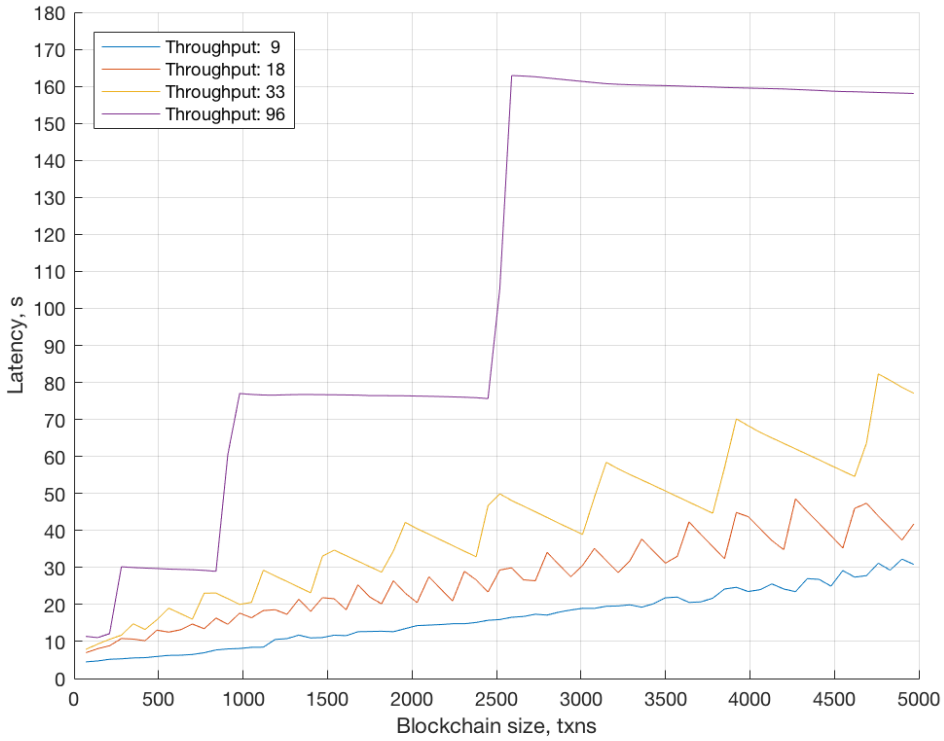


Figure 4.9: Latency variations against blockchain size with two remote Participants using Ezirmin's Sync module

parallel whilst achieving the same linear increases in latencies that pure Git acheives. If I assume, as before, a better streaming implementation for mempool synchronisation, then these results show that it is feasible for Logan to be able to achieve constant latencies in the size of the blockchain, as well as the number of nodes in the network. This would satisfy all of Logan's success criteria. Implementing this protocol would also be outside the scope of this project and I can therefore conclude that my implementation of Logan fulfils all of the success criteria that I set out.

4.3 The Effects of Leader Failure

TODO What should I put in this section?? Maybe compare overheads of the asynchronous thread that does the pushing? Maybe measure the extra latency?

Chapter 5

Conclusion

5.1 Successes

I have successfully designed and implemented Logan, a blockchain library that empowers developers to easily use distributed blockchain data structures in their applications. As blockchain is becoming more and more popular, more uses for it are being suggested by the day. Logan could be used in a wide range of applications where nodes in a blockchain network can be trusted. The system uses a leader based approach to consensus and is able to function well with mutliple remote nodes. It is also able to function well even at high throughputs, and can perform the necessary operations to recover from failure. Unfortunately, adding transactions to the blockchain over a network is a linear time operation with the size of the blockchain, but this is due to a poor implementation of the Git protocol used by Irmin. With a better implementation, Logan could achieve constant time transaction latencies with the blockchain size, as well as the number of Participants increases.

5.2 Possible Further Work

5.2.1 Failure Tolerance and Changing Leaders

Tolerating failure is an aspect of this project that I have only touched upon in this project. I have shown how Logan can perform replication but I have not implemented an automated approach for detecting and recovering from Failure. This is something I'd like to look into.

5.2.2 Adding Participants Dynamically

Currently, as well as only being able to define a single Leader, Participants can only be defined statically. Some consensus mechanisms introduce ways of adding and removing nodes to a network and I think it would be very feasible to build this feature into Logan.

5.2.3 Using SSH

Logan uses SSH connections to connect the Leader to Participants, but this may not be desirable in all situations and it may add unnecessary overhead. An alternative would be to use HTTP and I would like to investigate whether this would have an effect on the performance of the platform.

Bibliography

[1] Bitcoin energy consumption index. https://digiconomist.net/bitcoin-energy-consumption.

[2] Ethereum white paper. https://github.com/ethereum/wiki/wiki/White-Paper.

[3] Gnu make. https://www.gnu.org/software/make/.

[4] jbuilder. https://jbuilder.readthedocs.io/en/latest/overview.html.

[5] Lwt: A cooperative threads library for ocaml. https://ocsigen.org/lwt/.

[6] Gazagnaire, T. Introducing irmin: Git-like distributed, branchable storage. https://mirage.io/blog/introducing-irmin, 2014.

[7] Haber, S., and Stornetta, W. S. How to time-stamp a digital document.

[8] Lamport, L. The part-time parliament.

[9] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system.

[10] Ongaro, D., and Ousterhout, J. In search of an understandable consensus algorithm.

[11] Sivaramakrishnan, K. Ezirmin: An easy interface to the irmin library. http://kcsrk.info/ocaml/irmin/crdt/2017/02/15/an-easy-interface-to-irmin-library/, 2017.

[12] Torvalds, L. Git: A distributed version control system, 2005.

[13] Yaron Minsky, Anil Madhavapeddy, J. H. *Real World OCaml*. O'Reilly Media, Sebastopol, California, 2013.

Appendix A

Project Proposal

Computer Science Tripos - Part II - Project Proposal

Building a Blockchain Library for OCaml

Charlie Crisp, Pembroke College

December 29, 2017

Project Supervisor: KC Sivaramakrishnan
Director of Studies: Anil Madhavapeddy
Project Overseers: Timothy Jones & Marcelo Fiore

Introduction

The blockchain, in its simplest form, is a tree-like data structure. Chunks of data are stored in 'blocks' which contain the hash of the contents of the previous block. This creates a 'blockchain' which can exhibit branching in the same way that a tree data structure can (see Figure 1). One of the most important features of a blockchain, is that a change in a block, will alter the block's hash, thereby altering all the future blocks in the chain. This makes it very easy to validate that the data in a blockchain is trustworth, by verifying the hash in a block, is the same as the hash of it's parent's content.

Blockchain technology has generated a lot of interest in recent times, but

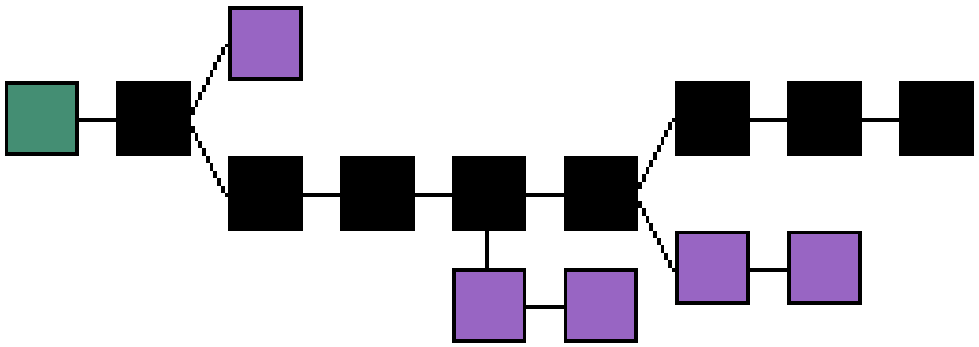


Figure 1: A typical blockchain structure [3]

mostly in the field of cryptocurrencies. With a simple Proof of Work consensus algorithm, the blockchain can be used to build a secure, distributed ledger of transactions. However, whilst the uses of the blockchain are far wider reaching than cryptocurrencies, progress outside of this field has been much slower.

I will build a pure OCaml, reusable blockchain library to allow the creation of distributed, secure ledgers, which are agreed upon by consensus. The library will allow users to create and add entries to a distributed blockchain ledger with just a few lines of code. The users will also be able to trust that entries in the blockchain are exactly replicated across all nodes in the network.

It will be built on top of Irmin [1] - a distributed database with git-like version control features. Being pure OCaml, the blockchain nodes can be compiled to unikernels or JavaScript to run in the browser. I will evaluate the blockchain by prototyping a decentralised lending library and evaluating the platform's speed and resilience.

Starting point

The project will build upon functionality provided by Irmin [1] which is a distributed database system. Irmin is fast, durable and has the branching capabilities which are required to build a blockchain.

Resources required

I will be using a Macbook provided by OCaml Labs [2] in order to develop the source code for the project. If the Macbook fails, then I will easily be able to transfer my work onto the MCS machines, as my project has no special requirements.
My work will also be backed up to a git repository hosted on GitHub and saved to a dedicated memory stick on a daily basis.
During the evaluation stage I will be running my platform on different cloud based devices and/or Raspberry Pi's. There are many possible providers for cloud computing, including Amazon Web Services and Microsoft Azure. OCaml Labs [2] will provide the necessary funds to acquire these resources.

Background

Consensus

Consensus is a group process where a network of nodes will reach a general agreement. There are different ways of achieving consensus but here are some of the most common:

1. **Proof of Work**: Trust is given to nodes which can prove that they have put in computational work. This is the consensus mechanism used by Bitcoin.

2. **Proof of Stake**: Nodes are selected to validate blocks based on their stake in the blockchain. There are few variations on this algorithm which introduce notions such as delegation or anonymity.

3. **Raft Consensus**: A leader is elected and acts as a governing authority until it fails or disconnects, whereupon a new leader is elected.

Work to be completed

The work for this project will be split into the following major parts.

1. Design and build a module to allow nodes to create and maintain a blockchain ledger. This will include allowing nodes to add blocks to the chain and to form new branches.

2. Design and build a module to allow nodes to interact over a network and to achieve consensus. As highlighted the Background section, there are many different ways to achieve consensus, and a large part of this work will be to determine which method is most suitable. This decision will take into account a method's failure tolerance in terms of nodes failing and network failure, as well as general speed and any requirements (e.g. computational work for a Proof of Work algorithm).

3. Design an application using these modules. This will take the form of a book lending platform where nodes will be able to register books and lend them to other nodes in the network. This application has been chosen, because the blockchain library should allow for typically centralised applications to be created in a decentralised way. It will also allow for testing of critical features, for example, books should never be 'doubly-spent', i.e. if one user believes they have ownership of a book, then no other user will think the same.

4. Design an evaluation program to simulate different load on the lending platform. This will be run in different configurations in order to measure the performance of the platform.

Evaluation metrics and success criteria

I will consider the project to be a success if the following criteria are achieved:

1. Nodes in the network are able to connect and communicate information.

2. Nodes are able to achieve consensus about the state of the distributed ledger.

3. Nodes are able to reconnect after being individually disconnected.

4. Nodes are able to re-converge after a network partition.

In order to evaluate the performance of the system, I will measure the *throughput* and *speed* of transactions of the book lending platform. Throughput will be measured in transactions per second, and speed will be quantified as time taken to complete a transaction. I will evaluate how these properties vary with respect to the following metrics:

1. **Number of nodes**: I will scale the number of nodes in the network between the range of 2 and 5.

2. **Rate of transactions**: I will vary the number of transactions made per second.

Should I achieve and be able to measure the above criteria within the time frame of my project, I will further test system against the following metrics:

1. **Network latency between nodes**

2. **Network bandwidth of nodes**

Timetable

1. **Michaelmas Weeks 2-4** (12/10/17 - 01/11/17):
   Set up an environment for developing OCaml and familiarise myself with the language and it's module system. This is important because the blockchain library needs to be reusable, and therefore well isolated.

2. **Michaelmas Weeks 5-6** (02/11/17 - 15/11/17):
   Familiarise myself with Irmin and it's data structures. This is important as I have never used the library before, but it will be used to build the blocks in the blockchain library. In this time I will also begin to design the API of my library.

3. **Michaelmas Weeks 7-8** (16/11/17 - 29/11/17):
   Finalise the API and start to build the module for creating and interacting with a distributed ledger. This will also involve investigating which hashing algorithms can be used to form the blockchain data structure.

4. **Christmas Vacation** (30/11/17 - 17/01/18):
   Finalise the API of the module for achieving consensus between multiple nodes. This work will also include investigating different methods of consensus and their suitability for my project.

5. **Lent Weeks 1-2** (18/01/17 - 31/01/18):
   Build the module for achieving consensus between modules. I will also start work on an lending library application which will be used to evaluate the performance of the blockchain library.

6. **Lent Weeks 3-4** (01/02/18 - 14/02/18):
   Finish work on the lending library application and install it on a number of Raspberry Pi and/or cloud based devices. I will also begin work on my dissertation and I aim to complete the Introduction and Preparation chapters.

7. **Lent Weeks 5-6** (05/02/18 - 28/02/18):
   Evaluate the performance of the platform by simulating load from each of the devices and measuring the speed of transactions. A stretch goal for this period is also to evaluate a range of further metrics. Additionally I will continue work on my dissertation and aim to complete the Implementation chapter.

8. **Lent Weeks 7-8** (01/03/18 - 14/03/18):
   Finish a first draft of my the dissertation by writing the Evaluation and Conclusion chapters. I will also send the dissertation to reviewers to get feedback.

9. **Easter Vacation** (15/03/18 - 25/04/18):
   With a first draft of the dissertation completed, I will use this time to review the draft and to make improvements. I will also incorporate feedback from reviewers, and complete the Bibliography and Appendices chapters.

10. **Easter Weeks 1-2** (26/04/18 - 09/05/18):
    Conclude work on dissertation by incorporating final feedback from reviewers.

11. **Easter Week 3-Submission Deadline** (10/05/18 - 08/05/18):
    I aim to have completed the dissertation by this point, and to be focusing on my studies. However, this time may be needed to make any final changes.

References

[1] Irmin - A pure OCaml, distributed database that follows the same design principles as Git.
https://github.com/mirage/irmin

[2] OCaml Labs - An initiative based in the Computer Laboratory to promote research, growth and collaboration within the wider OCaml community **http://ocamllabs.io/**

[3] Image of blockchain data structure from Wiki Commons. **https://commons.wikimedia.org/wiki/File:Blockchain.png**