**Charlie Crisp**

# Building a Blockchain Library for OCaml

Computer Science Tripos – Part II

Pembroke College

March 19, 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'd 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 []. However, whilst blockchain is undoubtedly the most important technology in the field of cryptocurrencies, where no single client can be trusted, it also has many uses outside this application. It can be used in other situations where clients can be trusted, for instance, a hospital maintaining medical records, or a bank wishing to record transactions from many of its own distributed clients.

I have implemented a blockchain library in OCaml which allows the easy creation of blockchain applications. The blockchain is synchronised via a leader-based consensus mechanism with eventual consistency. 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 cryptographic hash of the previous block in the chain. 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.

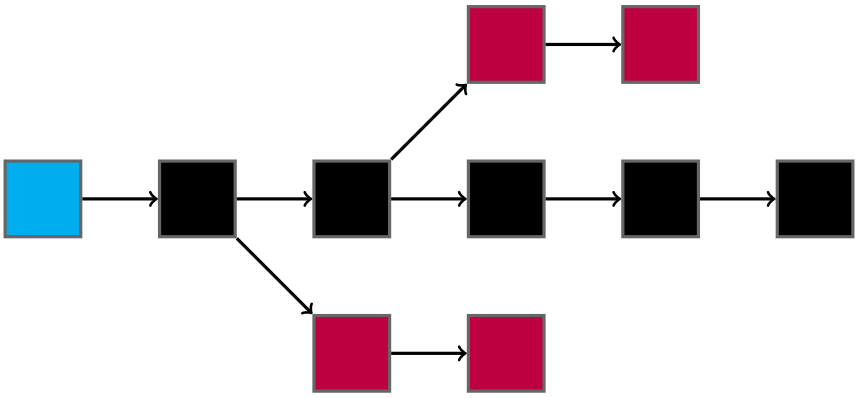


Figure 1.1: A typical blockchain structure

## 1.2 Blockchain Today

At the time of writing, cryptocurrencies are generating both a huge amount of excitement and cynicism in popular media. Cryptocurrencies aside from Bitcoin are paving the way to smarter uses of the blockchain. For example, Ethereum [2] introduces 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 a library which allows developers to create blockchain applications with the ease of importing a library. The project was designed to exist outside the realm of cryptocurrencies and therefore assumes that all participating nodes are trustworthy. Consensus is achieved by using a simple leader-based model where the leader node will periodically pull updates from all participating nodes and merge them into a central blockchain. This can then be viewed by all participating nodes, with the guarantee of eventual consistency. Setting up a network is as easy as specifying the location of the leader on all the participating nodes, and specifying the location of all participating nodes on the leader.

I have evaluated the project by... FILL IN EVALUATION DETAILS

# Chapter 2

# Preparation

Before starting work on the codebase for the project, I completed a lot of preparation in order to aid the development process later on. I spent some time learning to use OCaml and all of the different language features it provides. This was important because it allowed me to write idiomatic code which was not only powerful, but also easy to use by other developers. 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. 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 database system. Irmin is fast, durable and has the branching capabilities which are required to build a blockchain. The project also made use of Ezirmin [11] which provides a simplified API to Irmin.

## 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.

### 2.2.1 Pattern Matching

OCaml provides a very powerful syntax for matching patterns which allow you to write functions like the following...

Listing 2.1: OCaml Pattern Matching

```
type card = Card of string * int
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(class, 1) -> Printf.printf "It's the ace of %s!
     class"
  | _ -> Printf.printf "Unlucky"
```

In Listing 2.1, we have 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 uses the anonymous `function` keyword and matches the first argument of the tuple to the variable `class`.

### 2.2.2 Optionals

A built in data type that allows us to use the power of OCaml's pattern matching, is the `option`. 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.

### 2.2.3 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 = <fun
   >
```

As the type signature demonstrates, `bind` will take an `option` and apply a function to its contents if it exists, or return `None` otherwise.

OCaml provides a built in type `Result.t` which is effectively an extention 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";;
```

### 2.2.4 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.

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

Here, we have used a backtick 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 indeed other unspecified tags. When dealing with variant types as parameters, we may see the $<$ symbol in the type signature to denote that the parameter can only belong to given set of tags. The absence of both of these symbols indicates that a variant has exactly the given type signature.

### 2.2.5 Modules and Functors

Modules provide a good 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 2.2: 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 2.2 is a simple example where we have defined a module signature `Math` for adding and subtracting a custom type. The module `IntegerMath` is a module which adheres to this signature. The slightly odd looking `with type t = int` serves the purpose of letting the compiler know that the type t is externally visible. Whilst in this example, we have used the trivial example of integer maths, it is easy to imagine this extending to, for example, matrices or sets where these functions are not built in.

Modules are really useful for allowing the effective division of code into isolated units, however they are slightly inflexible. Maybe we want to extract lower level details of the code in a module? In this case, we would have to create a whole new module for each possible implementation of this abstraction. An example of this could be a database that could use an in-memory or on-disc format for storing data. Functors allow us to create modules from other modules.

Listing 2.3: Ezirmin Log Module

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

Listing 2.3 is an example from the Ezirmin codebase (which we shall see later) where we define a functor which takes the module AO which is used for creating append only stores, and the module V which defines a data type. The result of this is a functor which can be used to create a Log module with either an in-memory or on-disc backend.

### 2.2.6 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 build a blockchain library which included the interface for running both a Leader and Participant node. I also defined two executables for running both the Leader and Participants in an example case. I also 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 was particularly useful as it 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.

## 2.3 Existing Libraries

This project is built on top of Irmin and Ezirmin. Irmin is a library that allow the creation of different types of data store, such as Read-Only, Append-Only and Read-Write. Ezirmin provides a simplified API for interacting with Irmin as well as providing the implementation of a mergeable log. Both Irmin and Ezirmin allow the use of different backends including an in-memory, and on-disc format. The on-disc format uses the git protocol to store data, although an in depth exploration of this is left to the Implementation section. During the preparation stage of project, I spend some time familiarising myself with the API and codebases of these projects. Being able to create dummy applications, and get a deeper understanding of the lower-level code was extremely useful later on when I encountered a few bugs which I had to work around.

Another 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 2.4: 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 2.4 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. To elaborate on what these functions do, if we wish to obtain a thread which has already terminated with a value of type 'a, we can pass this value to `Lwt.return`, and it will happily oblige. If we wish to run a thread to completion and eventually return its value, then we can use `Lwt_main.run`. Finally, if we wish to chain together two threads, we can use use `Lwt.bind` (or the infix notation >>=) to pass the result of the first thread onto a function which will return a second thread.

## 2.4 Requirements Analysis

During the preparation stage of my project, I spent some time analysing the requirements that would be suitable for my projects. 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.

### 2.4.1 Data Structure

A key component of this project was to build a blockchain data structure that would allow transactions to be added to a ledger. From a participating node, it should be possible to add a transaction, of a given type, between two identities. It should also be possible to view an ordered list of all transactions which currently exist in the blockchain.

Listing 2.5: Blockchain Specification

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

module type I_Config = sig
  type t
  module LogCoder: I_LogStringCoder with type t = t

  val validator: (t list -> t -> bool) option
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 2.5 is the technical specification that I used to define my project, and the functions complete the following operations:

- `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_Config` contains information which is required to run the Blockchain. In particular, should the `validator` option contain the value `Some(f)`, then `f` will be a

function that accepts a history of committed transactions, and validates whether a further transaction is valid.

- `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 return 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.

### 2.4.2 Consensus

Building consensus into the blockchain module was a key part of the 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 eventual consistency. Strict consistency, although beneficial in some scenarios, is not required as it could add large overhead costs and is not necessary for all applications of the blockchain.
- The consensus mechanism must be scalable. Within the scope of this project, it should be possible for the blockchain to be shared by 4 or more nodes in a network. This should not hinder the performance of the system, and it should still be able to handle multiple successful transactions per second.

# Chapter 3

# Implementation

## 3.1 The Blockchain Data Structure

Irmin is a library for OCaml providing data store functionality using a Git backend. Ezirmin is a wrapper around Irmin that provides a simple log data structure which I used to create a blockchain data structure. In order to validate that these libraries fulfill the necessary criteria to be used as a blockchain, I took some time to investigate the semantic properties of the primitives they expose. Whilst there is no universally agreed-upon definition of a blockchain, I have used the following 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 makes no mention of consensus, and in this project I have made a clear separation between data structures and the consensus used to synchronise them.

### 3.1.1 Irmin

Irmin is a library for OCaml which provides Git-like, distributed, branchable storage [6]. Irmin exposes three main structures as shown in Figure 3.1: The Block Store, the Tag Store and the Irmin Store. For the reader familiar with Git, these can intuitively be though of as object/commits, branches and repositories respectively, however I will now explore each of these structures.

#### The Block Store

The Irmin Block Store is a virtual 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 will add new blocks which try to utilise as much shared history with the existing data in the store, in order to minimise storage space.

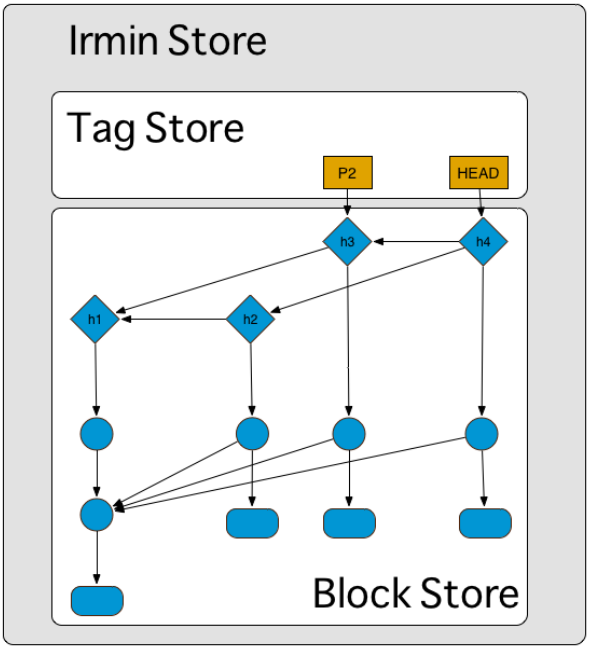



Figure 3.1: An Irmin Store composed of a mutable Tag Store and an immutable Block Store

#### 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. In particular, I will refer to the HEAD tag or pointer, which will index into the latest recognised block. In the case of logs, this is the latest recognised log item.

#### 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, but not the second criteria. In order to satisfy these criteria, I looked to Ezirmin which provides a higher level log data structure.

### 3.1.2 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 with some useful defaults. 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. In particular, the function `read` will read from the position of a cursor into the log, and will return a new cursor for the next unread log item, alongside the result.

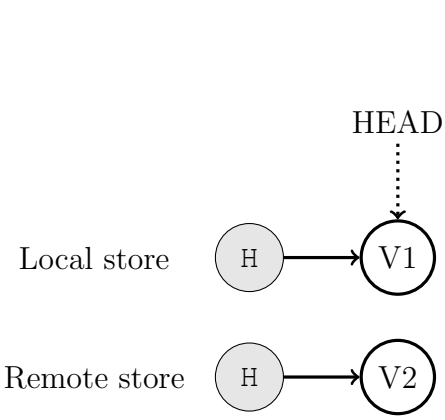
### Ezirmin Log as a Blockchain

Ezirmin uses Irmin blocks as an underlying data structure, and therefore satisfies the first criteria for being considered a blockchain. In order to see that the second criteria, i.e. that blocks are ordered with each containing the hash of its parent, I looked at the implementation of log items.

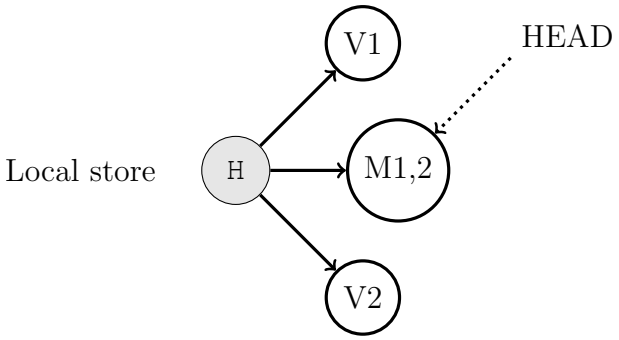
Listing 3.2: Ezirmin Log Item

```
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, which is stored as a block, has a key value which points to the parent log item. This imposes an ordering of log items, and means that any changes to previous log items, which will create a new block with a new address, will not be seen as part of the log. As the following section details, merging logs causes new `Merge` blocks to be created with more than one log item. Whilst this can change the semantics of the chain of pointers between the above log items, the blockchain data structure used in this project is never merged into, so only ever contains `Value` blocks containing singular log items. Therefore it can be concluded that an Ezirmin Log satisfies both of my conditions to be considered a blockchain, with the definition of a 'Block' in this case being a timestamped Ezirmin log item.




(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 3.2: Merging changes from a remote Ezirmin log into a local Ezirmin Log

### 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 `EzirminLog.Sync` module to merge new log items into the leader's mempool. In this section I will investigate the semantics of Ezirmin merges as it is critical to understanding how consensus is achieved in this project.

Irmin allows for the developer to program custom merge strategies. Whenever a synchronisation happens over the network, 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.

This merge behaviour is displayed in Figure 3.2 where a local log performs a merge from a remote log with a diverging history. A custom block, `M1,2`, is 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 logs make use of this functionality by storing log entries as either `Values` or `Merges`. `Values` contain single log entries whereas `Merges` store a list of `Values` in the order that they were created. For an Ezirmin log, the custom merge takes all log entries from `Values` and `Merges`, orders them by their timestamps, and returns a `Merge` object with the resulting list of values. This behaviour can be seen in Figure 3.2 by considering `Vn` to be a `Value` of the log entry `n` and `Mn,m` to be a `Merge` containing log entries `n` and `m`.

## 3.2 Consensus Algorithms

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 the project. The resulting design for consensus is a leader based approach, where participant nodes can commit transactions to a mempool. This mempool is then polled for updates by the leader, and these updates are then validated and committed to the main blockchain. This blockchain can be read by all participants, and provides a definitive source of ordered, committed transactions.

### 3.2.1 Building Consensus

#### 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 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 allows us to make guarantees on the validity of the blockchain based on the simple assumption that more that 50% of the workforce is genuine. If 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 doesn't 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 monotomically 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 doesn't 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.

### 3.2.2 A New Approach

I have build a consensus mechanism that uses a mempool and a simple leader based algorithm. 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 participating nodes 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 of implementing completely decentralised consensus. My approach makes use of the mergeable log data structure provided by Ezirmin, and the synchronisation module that allows updates from a log to be pulled into another. Finally, I have introduced the notion of validation which allows both participants and leader nodes to accept or reject transactions depending on arbitrary conditions.

### Leaders

My consensus mechanism 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 log items in the blockchain, up until the end of the shortest copy (or both copies).

Listing 3.3: Leader Specification

```
module type I_LeaderConfig = sig
  type t
  module LogCoder: Participant.I_LogStringCoder with type t
     = t
  val remotes: string list
  val validator: (t list -> t list -> t list) option
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 centralised specification presented in the Requirements Analysis. In particular, the `init_leader` function will perform an initialisation step, and then return 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. Validation can of this transaction can also happen at this stage, but as we'll see later, it cannot filter out all invalid transactions, and only exists to ease load on 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
  val validator: (t list -> t -> bool) 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 define how to validate transactions, to read from the blockchain, and to attempt to write to the blockchain by writing to a mempool.

### Retrieving local updates

The mechanism used by the leader to read mempool updates is different for the situations when the participant is on the same machine, and when it is on a remote machine. Here, I will detail the process of reading updates from a participant on the same machine as

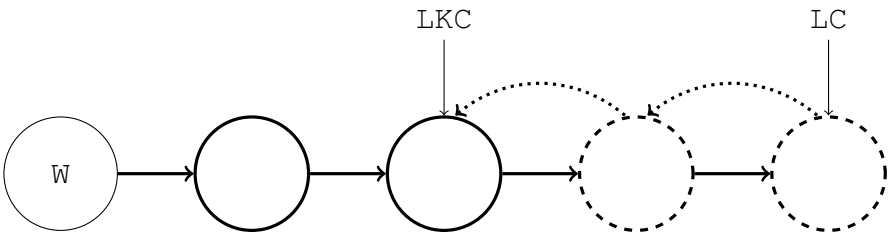



Figure 3.3: A mempool on a local participant. `W` indicates the history of a *Worker's* mempool. Blocks with dashed outlines represent transactions that have not yet been added to the blockchain. `LKC` and `LC` are the *Latest Known Cursor* and the *Latest Cursor* respectively.

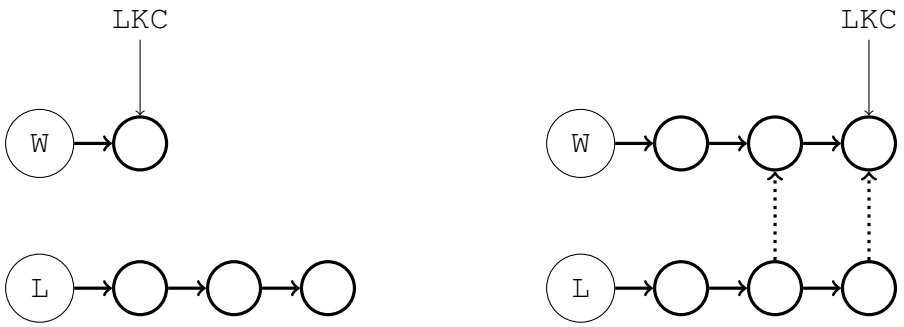
the leader.

The leader will always maintain a cursor to the latest entry it has read from the participant mempool. When it attempts to find new updates, it will get a new cursor to the latest element of the mempool. The leader can then compare these two cursors, and if the *latest known* cursor points to an entry with an earlier timestamp than the *latest* cursor, then it will add the *latest* item to an accumulator, and then repeat the process with a cursor pointing to the parent of the *latest* node. Finally, when the cursors match, the items in the accumulator are returned as new updates which can be added to the leader mempool and the *latest known* cursor is updated accordingly. It is important to note that no validation is performed at this stage as no item has been added to the blockchain yet. Figure 3.3 shows how unseen blocks are read sequentially from a mempool until the seen blocks are reached.

### Retrieving remote updates

The previous section demonstrates how updates can be retrieved from a mempool on the same machine as the leader. In this section, I will highlight how updates are retrieved from mempools on remote machines.

Irmin provides a `Sync` module which allows for the histories of a local and a remote Store to be combined according to a custom merge function. Ezirmin logs build on this functionality by providing a merge strategy which uses the log entry timestamps to order log entries in a merge. Figure 3.4 demonstrates how this merge works in a situation where a number of blocks have been added to a single remote participant mempool. The first (and most naïve) approach I took to pulling updates from a number of participants, was to sequentially merge updates from all the participants into the leaders mempool. Participants also merge updates from the leader's mempool before adding to their own mempool to increase the shared history and decrease the work required for the leader to perform the merge. After pulling all updates, the leader will traverse its own mempool, finding new updates until it reaches the *latest known* cursor. Now, the updates can be validated and added to the blockchain, and the *latest known* cursor updated to the latest




(a) Before merge (b) After merge

Figure 3.4: Merging mempool updates and adding to the blockchain from a single remote participant (below) to leader (above). `W` and `L` respectively signify the histories of the *Worker's* and *Leader's* mempools.

message in the mempool.

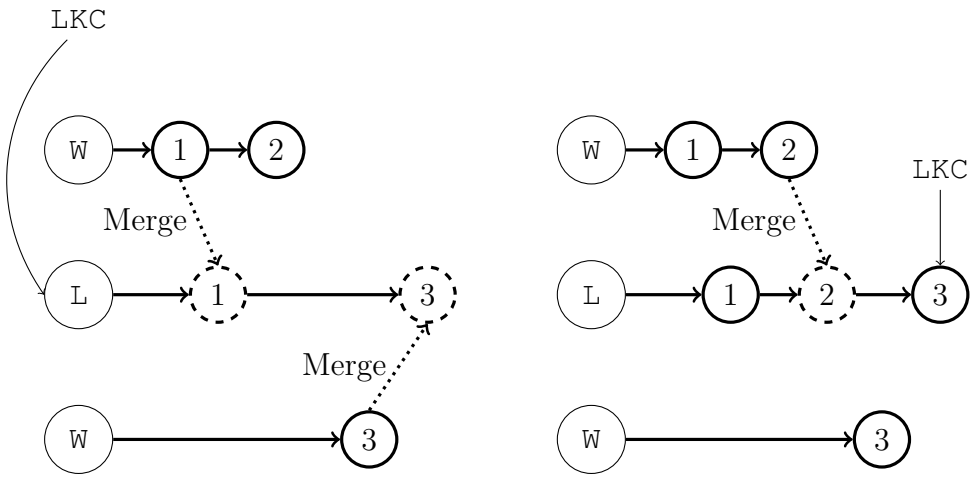
However, this approach is fundamentally flawed and causes many updates to be overlooked by the leader. This is because of the delays that occur when mempool updates are retrieved over the network from multiple workers. For example, Figure 3.5 demonstrates a situation where there are just two workers. In this case, updates from the first worker will be merged before the second, however, after the first merge has taken place, the first worker may have added additional transactions. If these transactions are timestamped before the latest transaction in the second worker's mempool, then when the leader next polls for updates, transactions may be merged to a position in the mempool before the latest known cursor. This means that they will not be seen by the leader and therefore they will not be added to the blockchain. I shall refer to these transactions as *missed* transactions, and all others at *tracked* transactions.

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

1. Missed transactions must have been added during a merge. If they occured before the merge had begun, then they would have been tracked. Alternatively, missed transactions must occur after the latest known cursor from the previous merge.

2. Missed transactions must have been added at a point in time earlier than the transaction pointed to by the latest_known cursor. If they occured at a later point in time, then they would be tracked by future leader polls.

These lead very naturally onto a less naïve algorithm for retrieving updates which only adds updates that have existed for more than one 'poll cycle'. Instead of maintaining a single cursor to the latest-known item, another cursor is now maintained to the previous latest-known item. I will refer to these as the `latest_known` and `previous_latest_known` items. After merging the newest set of updates, a leader can be sure that no more missed transactions will be added before the `latest_known` item, and after the `previous_latest_known` item.




(a) After first sync from leader. Transactions 1 and 3 are merged after the latest known transaction and are therefore *tracked*

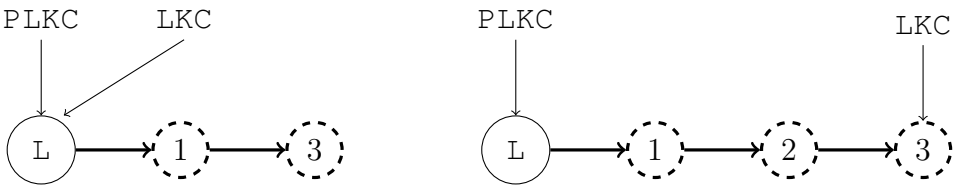
(b) After second round of leader synchronisation, transaction 2 is merged into the mempool before the latest known transaction and is therefore *missed*

Figure 3.5: In this diagram, time flows from left to right. The sequential nature of mempool merges causes transaction 2 to be merged into the history of the mempool before the transaction pointed to by the latest known cursor. 2 will therefore not be added to the blockchain.

Listing 3.5: Selecting new updates

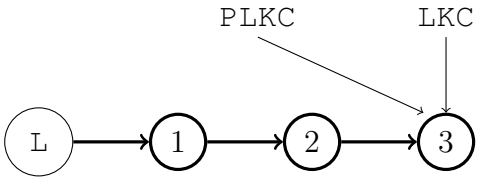
```
let is_early_enough = is_earlier_or_equal scanning_cursor ~
   than:latest_known in
let is_late_enough = is_later scanning_cursor ~than:
   previous_latest_known in
match is_early_enough, is_late_enough with
  | Some(false), Some(true) -> (*Item cannot be added to
     blockchain yet. Move scanning cursor back by one item
     and loop*)
  | Some(true), Some(true) -> (*Item can be added to
     blockchain. Add item to accumulator, move cursor back
     by one item and loop*)
  | _ ->(*Too far back in the mempool, so return item
     accumulator*)
```

Listing 3.5 shows a snippet of code from within a recursive function to get updates from the mempool, which can be added to the blockchain. This demonstrates a new approach which starts with a newly retrieved cursor, `scanning_cursor`, to the latest mempool element and then scans back through the mempool, adding valid items to an accumulator and eventually returning. On each loop, this algorithm will check if the item is valid, adding the item to the accumulator if this is the case, and then either return the accumulator or move the `scanning_cursor` to the next item back in the mempool.




(a) After first merge, items 1 and 3 have been merged but not added to the blockchain.

(b) After second merge, item 2 is now part of the mempool history, and items 1, 2, and 3 can be added to the blockchain.




(c) After second merge, items 1, 2 and 3 have been added to the blockchain.

Figure 3.6: Using two cursors, any missed items can be caught and added to the blockchain. PLKC signifies the *Previous Latest Known Cursor*.

Every item from the `latest_known` item (inclusive) to the `previous_latest_known` item (exclusive) will be added to the blockchain. Any of the further updates are not safe to add, and will be postponed until the next poll. In the case that both cursors point to the same item (i.e. no new items were retrieved in the latest poll), no items are added as it is always assumed that the `latest_known` item is added in the previous poll. The cursors can now be updated as follows, where `get_new_latest_cursor` will get a cursor to the latest item in the leader's mempool.

```
previous_latest_known_cursor := latest_known_cursor;;
latest_known_cursor := get_new_latest_cursor;;
```

This approach will not only pick up all the previously missed transactions, but all the tracked ones too. Figure 3.6 demonstrates how this approach would solve the problem presented in Figure 3.5.

At a first glance, it may seem sensible to use the `latest_known` cursor to get new updates from the mempool by just iterating back through the mempool from cursor. The problem with this approach is that the cursor is an abstraction of a tag to a block in the Irmin Block Store. This means that the history of the mempool according to that cursor is not changed by any subsequent merge operations. Consequently, any *missed* transactions will not be visible using this approach.

This now begs the question 'How can the new approach, which uses out-of-date cursors, still be valid?'. The answer to this is that, the new approach only uses the cursors to perform timestamp comparisons. The result of these comparisons is the same whether

a cursor to the items in the out-of-date mempool or a cursor to the items in the new mempool is used.

# Chapter 4

# Evaluation

# Chapter 5

# Conclusion

# 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

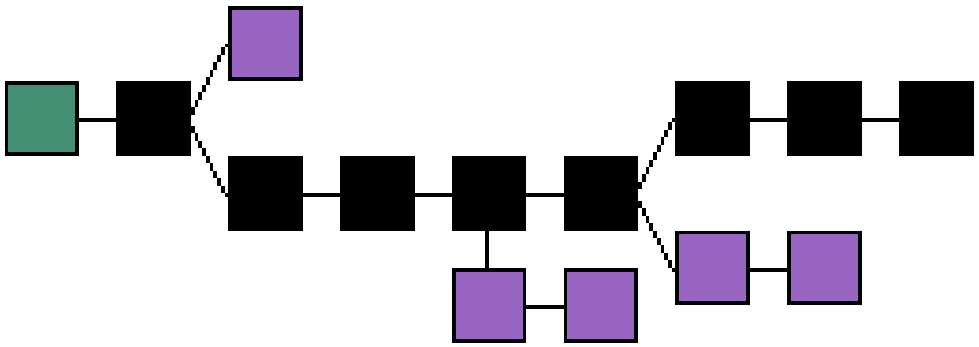


Figure 1: A typical blockchain structure [3]

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**