Charlie Crisp

Building a Blockchain Library for OCaml

Computer Science Tripos – Part II

Pembroke College

February 18, 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 [6] 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 [5]. However, until the creation of Git [8] 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.

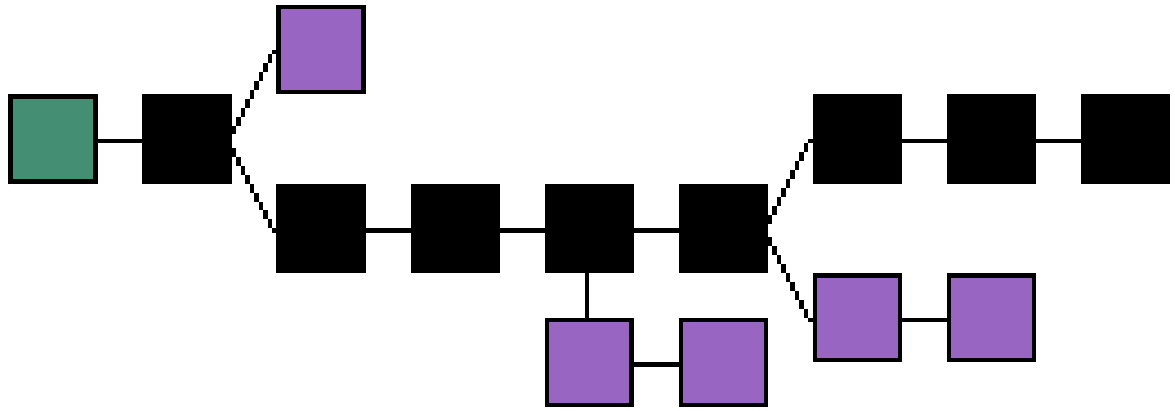

Figure 1.1: A typical blockchain structure

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 [1] 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 [7] 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 [9] 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 like the following...

```
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 the first example above, 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`.

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.

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 above 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 code 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.

```
# 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.

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.

```
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
```

The above code 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.

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

The above code 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.

Development Environment

When developing a large scale system with OCaml, there are a couple of build systems available to use. 'jbuilder' [3] 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 [2] 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

Lwt

Irmin

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.

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.

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

module type I_Blockchain = sig
  val add_transaction_to_blockchain: 't -> [> `Error | `Ok] Lwt.
     t
  val get_all_transactions: unit -> 'a list Lwt.t
end

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

The above code is the technical specification that I used to define my project. `add_transaction_to_blockchain` will add a user defined type to the blockchain and then return a polymorphic variant type containing information about whether the operation was successful. `I_LogStringCoder` is a module that can be used to allow arbitrary data types to be added to the blockchain, so long as a function is provided to encode them as a string. Finally, `Make` is a functor which will accept a module for coding the strings, and a module for any other configuration details and will return a Blockchain module.

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

Git as a blockchain

Git provides a data structure which can be interacted with via a command line API. However, it is not trivial why these data structure is classed as a blockchain.

In order to convince oneself of this, it is worth considering what features are required for a data structure to be considered a blockchain. Whilst there is no universally agreed definition of a blockchain, it is commonly accepted that it will exhibit the following features:

1. Data is stored in 'blocks'.
2. Blocks are ordered.
3. Each block contains a link to its parent block.

Now we can consider the promises that Git makes and ensure that the above conditions are satisfied.

1. Git stores data as sequences of 'commits' which can be thought of as blocks.
2. Git history is stored as a tree structure with arbitrary branching. This imposes an order on commits or blocks.
3. Each Git commit contains the hash of the previous or parent commit.

It is clear to see that Git has all the necessary properties to be used as a blockchain data structure.

Irmin

The next question that I needed to answer, was how it was best to interact with an underlying Git protocol, using OCaml. Irmin is a library for OCaml which provides

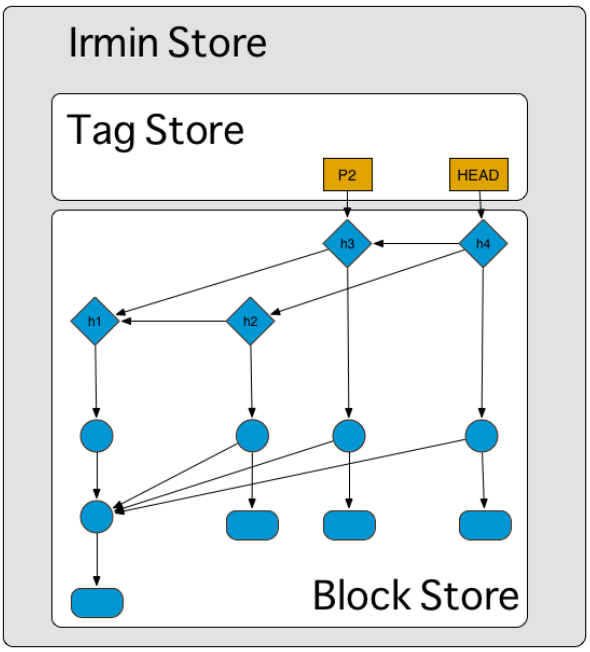

Figure 3.1: An Irmin Store composed of a mutable Tag Store and an immutable Block Store

Git-like, distributed, branchable storage [4].

Irmin uses a Git backend to store objects which immediately makes it suitable for use as a blockchain. However, it is also possible to consider Irmin's data store at a higher level.

Irmin provides access to a Block Store and a Tag Store as shown in figure 3.1. The block store is an append-only store of immutable key-value pairs. Mutability comes from the tag store, which provides a way of mapping global branch names to blocks. Both of these stores combine to form an Irmin store with promises which are beneficial for concurrency. For example, if a tag is not mutated, then you can be sure that the no change in the block store will be visible.

Irmin Stores can be though of in terms of the following API, expressed in OCaml code. Here, we expose types for keys which address values, and tags which point to keys. The read and update functions allow us to interact with the block store.

```
type t
(* The type for Irmin store. *)

type tag
type key
type value
```

```
val read: t -> ?branch:tag -> key -> value option
val update: t -> ?branch:tag -> key -> value -> unit
```

Amongst other different interfaces, Irmin provides a built in API for interacting with an append-only store. This key functionality of this module is expressed in the following code, where `mem` checks for the presence of a key, `find` reads values, and `add` writes to the store.

```
val mem: t -> key -> bool Lwt.t
val find: t -> key -> value option Lwt.t
val add: t -> value -> key Lwt.t
```

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.
For this reason, I decided that it would be a suitable data structure to use in this project.

3.2 Mempools

When making a transaction using Bitcoin or some other cryptocurrencies, a request for this transaction is placed in what is known as a Mempool[1]. This Mempool can be thought of as a waiting room for transactions. When miners are assembling and mining blocks, this is where they will pick the transactions from.

3.3 Consensus Algorithms

Typically, cryptocurrencies (which rely on blockchain technology) use a Proof of Work protocol to achieve consensus. There are, however, other ways of achieving consensus, and I evaluated a number of these in order to inform my approach to building consensus into my project.

Existing Algorithms

Proof of Work

The proof of work method for achieving consensus is actually very simple. Whenever a block is received by a participating node, the node will check that the block contains a proof of computational work done. In the field of cryptocurrencies, this proof takes the form of a random sequence of data which, when appended to the end of the block,

[1]Mempool is short for Memory Pool

causes its hash to be prefixed with a set number of 0s. Because the data appended to the end of a block can only be found by a brute force method, which will require lots of computation/work.

Proof of Stake

Paxos

Raft

A New Approach

Pros and Cons of a Leader Based Approach

Writing to the blockchain

How to deal with failure

Chapter 4

Evaluation

Chapter 5

Conclusion

Bibliography

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

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

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

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

[5] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. 1990.

[6] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

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

[8] Linus Torvalds. Git: A distributed version control system, 2005.

[9] Jason Hickey Yaron Minsky, Anil Madhavapeddy. *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

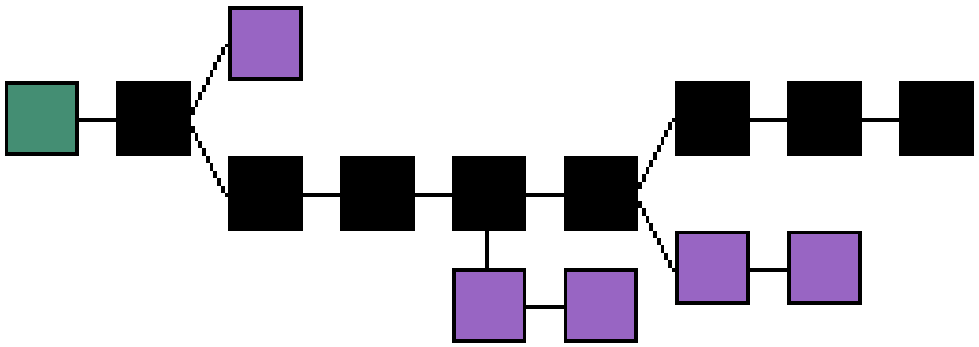

Figure 1: A typical blockchain structure [3]

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**