UNIVERSITY COLLEGE LONDON

MSC COMPUTER SCIENCE

QuantiTeam

Blockchain architecture as a medium to verify collaborative work
A feasibility study

Author: Benjamin Kremer

Supervisor: Dr. Ghita Kouadri Mostefaoui

September 2016

Abstract

The ability to work proficiently in collaboration with others is highly valued within a large number of social contexts. Yet, a person's ability to do so is scarcely quantifiable in any meaningful way. The goal of this project was therefore to examine the feasibility of constructing a system which can verify and quantify collaborative work. The project was set in the specific context of attempting to solve student disengagement, as the larger concept arose from this concrete problem.

The project attempts to provide a high level of potential for true verification and quantification of collaborative data by utilising a distributed data structure known as a blockchain. Following an analysis of how the proposed system could be structured in terms of interactions between a client the blockchain, an API was constructed to provide as much functionality as was feasible within the time available. Additionally, a simple client-side mobile application was developed to showcase the API's functionality in a concrete manner.

While the project falls short of establishing a manner to truly verify and quantify collaborative work with a distributed blockchain, it demonstrates that such an endeavour is certainly feasible given more time and expertise in the topic area, thus providing a basis for future work to create a *bona fide* system for verification and quantification of team work.

A demo video which showcases the system's current main features can be found here:
https://youtu.be/yhUtKqT8j3s

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Ghita Kouadri Mostefaoui for her continuous guidance throughout this project. I would also like to thank my family, my flatmates, and Kimeshan Naidoo & Aditya Mukherjee in particular for their continuous support.

I owe a special thank you to Zach Ramsay from Eris Industries, whose enthusiasm for what I was trying to achieve provided a wealth of motivation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Outline

Motivation is the key to human action. Without it, the source of any action performed is likely to be grounded in external pressures rather than the actor's desire to act. In adolescents and teenagers, this situation frequently arises within a key element affecting their future: their education.

This observation stems from the author's personal experiences during his youth and, more recently, from experiences as a voluntary English language teacher. To the author, there were two clearly identifiable steps which could lessen the amount of disengagement from school-related matters that is often observed in adolescents and teenagers. Firstly, providing a way for students to quantify their achievements could drive re-engagement with the learning material, by breaking it into manageable chunks with clearly defined rewards for each chunk.

Secondly, adding an explicitly social aspect to this quantification process by focusing on collaborative work could further drive motivation and re-engagement, by creating a social net for students to feel valued and as carriers of responsibility towards others. Beyond achieving a greater sense of purpose for each individual, this quantification of collaborative work could also double as a useful metric of team work skills to refer back to at a later point in time, filling a gap for a skill that is currently not quantifiable or verifiable on paper.

1.2 Project Goals

This project began as an attempt to address this phenomenon of student disengagement and, over time, developed into a larger investigation of whether it is possible to utilise blockchain technology not just to form a network of re-engaged students, but as a tool for the verification and quantification of group-based work at large.
The major motivations that guided the project were thus twofold: in a broader sense, to demonstrate how the properties of a blockchain can be leveraged in a social, communal context, rather than the financial contexts they are usually applied to, and more concretely, to create a free, open and extensible platform for students of all ages to re-engage with their peers in a productive, educational manner, underpinned by a model of motivation which rewards collaborative work as a team.

Working from this technological and philosophical motivation, respectively, the following goals were established for the project:

- Create a public Application Programming Interface (henceforth API) to allow users to interact with a blockchain in a well-defined manner, independent of a chosen client-side implementation.

- Provide an iOS client-side app for basic task, user, and team management to offer a graphical representation of the API in a format familiar to non-technical individuals.

1.3 Personal Aims

Having been a follower of the bitcoin[1] movement for more than two years, the author has been looking for an appropriate opportunity to apply the blockchain data structure, which is a key part of the technology underlying bitcoin, to a project of his own. This project's goal of verifiability of team work fits a blockchain's ability to provide a distributed ledger of transactions or events.

The following points provide an overview of the author's personal aims for the project:

- Become familiar with native mobile app development by utilising existing JavaScript knowledge in the context of React Native[1].
- Learn how to write smart contracts in Solidity[2] that are useful in the context of the project and for existing platforms such as Ethereum[2].
- Implement a form of blockchain technology, by learning how to set up, run & maintain a blockchain proficiently.
- Gain a fundamental understanding of Docker[3] and how it leverages "containerisation" to achieve straightforward deployments of development environments.

1.4 Project Management

The project followed the methodology of the Unified Process[4] (henceforth UP) framework for software development, and was thus structured in terms of iterative phases. The system was at first modeled as a whole by using a theoretical deployment diagram in order to

[1]https://bitcoin.org/en/
[2]https://www.ethereum.org/

gain an understanding of which technological entities were required to neatly encapsulate the responsibilities within the system. Sequence diagrams were then used to model more complex client-server-blockchain interactions during each UP iteration, where a new stage usually took the form of defining and implementing a new domain within the API, e.g. task management or user management.

Given the limited amount of time available for a project of these dimensions, along with the research and experimentation required to validate whether the described goals were even achievable, user requirements were captured using a heuristic of what the essential functionality would be for the client-side application, and thus the API in general, in order for a user to create, manage and discuss their open tasks with their team. This was done by utilising the MoSCoW categorisation scheme[5].

During the implementation stage of the project, a daily log was kept of the key advances and issues encountered during that day of development. This served as a useful tool to summarise the progress made, as well as providing an immediate opportunity to document issues that were encountered.
Throughout the implementation stage, required programming work was broken down into separate units of work by using Github Issues[3] and organised into a Kanban[6] board with the help of waffle.io[4].

As the API continued to grow, unit testing the API's public functions became an essential part of the project workflow. Due to the primitive error handling of the still very young Solidity programming language (used to construct smart contracts for the blockchain), any responses coming from the blockchain had to be rigorously tested server-side in order avoid silent breakages and regressions.

[3]https://developer.github.com/v3/issues/
[4]https://waffle.io/

1.5 Report Overview

Chapter 1

This chapter outlines the problem addressed by the project, as well as stating its goals, the author's personal aims, and the approach to managing the project.

Chapter 2

This chapter establishes context for the stated problem domain, surveys previous work either directly or tangentially related to the project, and provides an overview of the programming languages, libraries and development tools used within the project.

Chapter 3

This chapter specifies the project's requirements and their associated use cases, whilst also assessing the system's architecture by analysing each major component of the its technology stack.

Chapter 4

This chapter details the key features and patterns that make up QuantiTeam's implementation, assessing the blockchain, the server and the client-side app in turn.

Chapter 5

This chapter explains the approach of test-driven development used for the project, as well as outlining how the system was tested at different levels of granularity, using unit tests, integration tests, and functional testing.

Chapter 6

This chapter assesses to what extent the project's goals and the author's personal aims were achieved, provides a critical review of the system as a whole, and offers pointers for possible future work.

Chapter 2

Background Information and Related Applications

2.1 Blockchains & Dropouts

A 2004 study by the National Research Council showed that upwards of 40% of high school students feel disengaged from learning and exert little effort on school work[7]. More worryingly still, 70% of dropouts stated lack of motivation, in a 2006 study, as the key reason for their departure from school[8]. In liaison with these findings, a study which tracked the educational careers of a group of individuals found that 23% of those who dropped out cited a sense of not belonging as the reason for their departure from school[9].

While the factors for a student's motivation (or lack of it) regarding their education may stem from a myriad of factors, engaging the individual on a social level may thus be posited as a key driver in motivating them to remain engaged in the process[10]. Of course, simply being bound into a social context does not mean that the student will make meaningful progress regarding their work, as attendance does not constitute attention to what is being taught.

Whereas there is little school staff can do to ensure the attention of a student within a physical school, supplementing the educational process with blockchain-based technology may provide a solution to this impasse between the individual student's aims and that of the educator. By establishing a system in which collaborative work is directly linked to both an individual and social reward, students are able to focus on a more immediate &

manageable target than an abstract and potentially far-off seeming concept of, for example, a high school diploma.

> "Rewarding specific actions that students can control, such as completing homework, yields better results than rewarding accomplishments that may seem beyond their reach or out of their control, such as whether they earn an A grade."[7]

Figure 2.1 visualises a widely held theory regarding educational conditions that promote intellectual engagement[11]. The proposed system would thus provide a shift in the educational context, providing a sense of control (teams register their own tasks), values & goals (wanting to help each other and/or earn a higher ranking) and social connectedness (forming and maintaining teams, reaching out for and offering help).

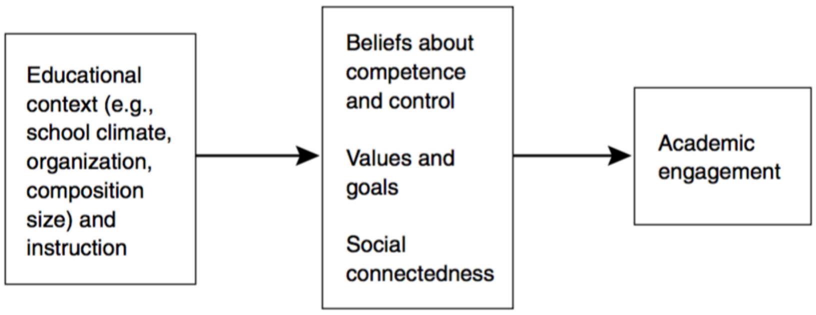

Educational context (e.g., school climate, organization, composition size) and instruction
Beliefs about competence and control
Values and goals
Social connectedness
Academic engagement

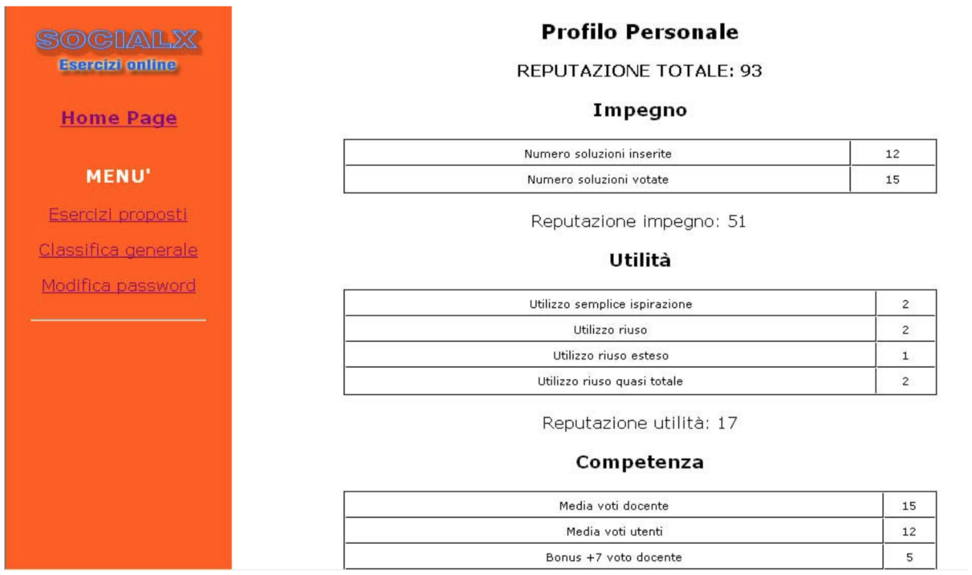
Figure 2.1: A theory on educational conditions that promote intellectual engagement

Using a blockchain as a data structure – in this case one which is built on a proof-of-stake consensus algorithm – to register, manage and save school-related educational tasks, along with a reward & ranking scheme linked to the blockchain, would enable students to reap immediate rewards from their individual & collaborative efforts. Furthermore, a blockchain's ability to act as a pseudo-anonymous public ledger of events, in this case school-related tasks, would provide students with the ability to refer back to said tasks at a later point in time to demonstrate their teamwork skills, therefore making a scarcely confirmable skill set auditable.

2.2 Previous Work/Existing Applications

This section gives an overview of related work within the context of heightening student engagement described above. An observation made while researching similar applications was that there seemed to be a number of applications focused on digitalising and strengthening student-to-student and student-to-teacher engagement, but none were built in terms of a distributed, reviewable entity, such as a blockchain.

SocialX



Profilo Personale

REPUTAZIONE TOTALE: 93

Impegno

Numero soluzioni inserite	12
Numero soluzioni votate	15

Reputazione impegno: 51

Utilità

Utilizzo semplice ispirazione	2
Utilizzo riuso	2
Utilizzo riuso esteso	1
Utilizzo riuso quasi totale	2

Reputazione utilità: 17

Competenza

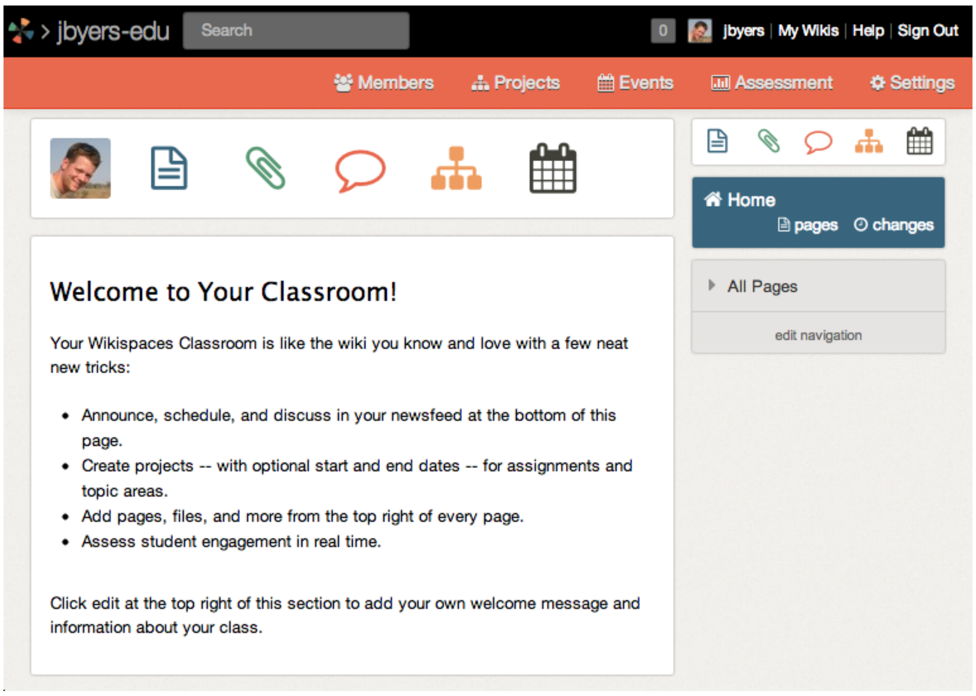
Media voti docente	15
Media voti utenti	12
Bonus +7 voto docente	5

Figure 2.2: SocialX's reputation overview

SocialX[12][13] is an exercise sharing tool which enables students to earn reputation points, which are visible to their peers and the given subject's teacher, by submitting solutions to exercises. A further source of reputation are endorsements, which a student may receive from their peers if they were inspired to reuse the student's solution.

SocialX differs to QuantiTeam in three key ways: firstly, reputation within SocialX is calculated according to factors which involve the judgment of each other's work, whereas in QuantiTeam's reputation is a function of team size and is either rewarded in full to all team members or not at all, depending on whether the task/exercise in question was completed. Secondly, QuantiTeam has no teacher role, and is therefore able to avoid a bottleneck SocialX suffers from. Thirdly, SocialX defines the term "global ranking" as a measurement of student reputation across subjects/courses, whereas QuantiTeam strives for a truly global, decentralised ranking system.

WikiSpaces



Welcome to Your Classroom!

Your Wikispaces Classroom is like the wiki you know and love with a few neat new tricks:

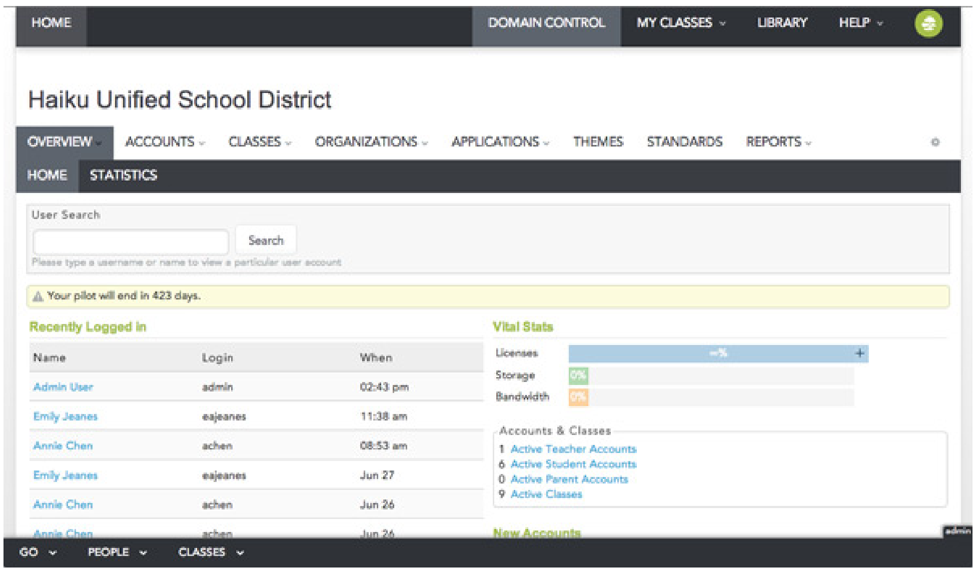
- Announce, schedule, and discuss in your newsfeed at the bottom of this page.
- Create projects -- with optional start and end dates -- for assignments and topic areas.
- Add pages, files, and more from the top right of every page.
- Assess student engagement in real time.

Click edit at the top right of this section to add your own welcome message and information about your class.

Figure 2.3: The WikiSpaces Classroom dashboard

WikiSpaces[14] is a web app which provides a virtual classroom workspace in which teachers and students can work on written projects individually or in teams. Features include a social media-like feed, collaborative writing and commenting tools, and a project management system.

HaikuLearning



Haiku Unified School District

User Search

Please type a username or name to view a particular user account

Your pilot will end in 423 days.

Recently Logged In

Name	Login	When
Admin User	admin	02:43 pm
Emily Jeanes	eajeanes	11:38 am
Annie Chen	achen	08:53 am
Emily Jeanes	eajeanes	Jun 27
Annie Chen	achen	Jun 26
Annie Chen	achen	Jun 26

Vital Stats

Licenses	--%
Storage	0%
Bandwidth	0%

Accounts & Classes

1 Active Teacher Accounts
6 Active Student Accounts
0 Active Parent Accounts
9 Active Classes

New Accounts

Figure 2.4: The HaikuLearning dashboard

HaikuLearning[15], similarly to WikiSpaces, is a web app which attempts to emulate and enhance the classroom experience for both school teachers and students. It offers a unified environment to hand in assignments and to receive feedback & grades, as well as enabling integrations with Google Apps for extensibility.

A notion which provides a key separation between QuantiTeam and these services is that both WikiSpaces and HaikuLearning place the metaphorical ball firmly in the court of teaching staff, as teachers still decide what material is worked on and how, even providing the ability to monitor students' progress. Within QuantiTeam, tasks are set by team members themselves and contextual information regarding the progress or completion of a user's tasks cannot be monitored by any other individual.

2.3 Programming Languages and Libraries

2.3.1 Blockchain

Before proceeding with the following section, two frequently used concepts should be clearly delineated: blockchains and smart contracts. A blockchain is a data structure composed of a list of grouped transactions, known as blocks, where each block is linked back to the previous block in the chain[16]. Each block is identified by a hash, which is stored in the block's header alongside the previous block's hash, known as the parent block[16], thus providing a traceable, reverse-chronological chain of transactions over time. The concept of a smart contract was originally developed by Nick Szabo in 1997, and is usually evoked in terms of a digital protocol which enforces and/or verifies the performance of a contract between two or more parties[17].

At the outset of the project, a key decision which had to be made was the choice of blockchain implementation to be used. This decision was critical, as discovering a major flaw in the implementation or an incompatibility between what the implementation could provide and what was needed, would have resulted in (at least temporary) deadlock for the entire project.
The author's initial experimentation with OpenChain[1], which seemed to offer the possibly useful ability to chain contracts to each other, was quickly abandoned as the available documentation was insufficiently detailed and partially ambiguous. This would have left the project open to a lot of guesswork and thus the above mentioned risk of deadlock.

[1]https://www.openchain.org/

MultiChain[2], on the other hand, was an implementation which had well-defined documentation, but it assumed a level of pre-existing familiarity with the API of a blockchain, as well as providing preciously little context as to how smart contracts could be developed for the platform in an effective way.
The apparent drawbacks of the two mentioned blockchain implementations therefore led the author to the Eris[18] platform. Eris itself provides a suite of tools which wrap and augment a Tendermint[19] blockchain implementation. Eris's wealth of documentation and its well-structured command line interface (henceforth CLI) were the deciding factors for it becoming the blockchain implementation of choice. The documentation provided not only step-by-step examples of how a developer could configure and deploy a blockchain, but also provided in-depth tutorials and examples on how to write & deploy Solidity[2] smart contracts for the platform; highly valuable for a smart contract novice such as the author. Furthermore, the fact that Tendermint is an entirely separate software project meant that there was an additional repository of documentation for this specific type of blockchain and the computer science it is based on.

2.3.2 Server-side

Considering the author's background as a JavaScript developer, the natural choice for a web server for the project was NodeJS[3]. NodeJS provides a great level of flexibility and extensibility thanks to its huge ecosystem of open-source libraries available through its package manager, NPM[4], thus providing an essential basis to help mitigate the possibly complicated technical details of interfacing with a low-level blockchain.
As JavaScript is an ever-evolving language, a choice had to be made as to whether the

[2]http://www.multichain.com/
[3]https://nodejs.org/en/
[4]https://www.npmjs.com/

commonly supported ECMAScript[5] 5 (ES5) specification should be used or the modern standard, ES6. ES5 was selected to provide the maximum level of compatibility for wherever the server is being deployed, as native support for ES6 is not widespread as of the point of writing and transpiling the server's code from ES6 to ES5 for each deployment would have been a lot more trouble than use for a humble web server.

QuantiTeam's server makes use of a handful of key libraries:

Express

Express[6] is a minimalist framework for NodeJS web servers which provides a powerful layer of abstraction on top of NodeJS's raw server interface. Express helped accelerate the development of the API, while leaving the possibility of fine-tuning the NodeJS server itself intact.

eris-wrapper & eris-logger

The eris-wrapper & eris-logger modules were adopted from a "Hello World"-style example Eris provides to showcase how a NodeJS server implementing their platform could be structured[7].
eris-wrapper provides a convenient abstraction of low-level bindings that have to take place between the NodeJS server and the Tendermint blockchain, while eris-logger simplifies setting up a logger with different logging types, such as `ERROR`, `INFO` and `DEBUG`.

[5]http://www.ecma-international.org/
[6]https://expressjs.com/
[7]https://github.com/eris-ltd/hello-eris

Async

Async[8] is a utility library that helps manage the flow of asynchronous functions, which are a common occurrence in NodeJS. Within the scope of the server, Async was primarily used to chain together multiple sequential interactions with the blockchain.

2.3.3 Client-side

The platform chosen for the client-side representation of the system was that of a mobile application. While building a web application would have been just as feasible in conjunction with the system's API, mobile development was chosen as the preferred paradigm in order to maximise the potential for the project's author to gain new skills, as well as two important usability properties:

Location independence - Users are able to check the status of tasks, communicate with team members and receive notifications regardless of their current surroundings.

Familiarity - Packaging the API's graphical representation into a mobile app provides an already familiar UI framework which users are accustomed to, whereas web apps often have a large degree of variance in appearance, structure and behaviour.

During the MSc's GC02 "App Design" course, the author and his team had the opportunity to build a React web application which was also required to run as a mobile application. This was achieved by wrapping the web app with the Apache Cordova[9] mobile development framework. This experience revealed two key insights regarding the intersection between web- and mobile-based client-side development:

[8]https://caolan.github.io/async/
[9]https://cordova.apache.org/

Firstly, that Facebook's React[10] web development library, in liaison with Dan Abramov's Redux[20] library for state management, could be applied almost seamlessly to the context of a mobile application. React's philosophy of thinking in terms of individual view components, together with Redux's implementation-agnostic approach to managing the application's state, largely abstracted away the conceptual differences between building an HTML document and building a mobile app view.

Secondly, that porting a web app to a mobile app simply does not provide an authentic native experience on a mobile device, as it is almost impossible to account for and implement all the differences and individual nuances in, for example, animation styles between mobile operating systems. This resulted in the look & feel of the application being closer to that of a mobile browser rather than a native mobile app.

Based on these two lessons learnt, the author decided to use Facebook's React Native[1] for the project, which grafts a native mobile development framework on top of the outstanding React library, along with the now familiar Redux library.

React Native offers all the benefits of React while allowing the developer to hook into events and effects of the native mobile operating system. This means that transitioning between views smoothly or sending a push notification, for example, becomes a trivial undertaking.

Redux provides a well-defined interface to manage the state of the application's data by managing all of it through a centralised `store` object, the contents of which can be altered exclusively by a set of actions previously defined by the developer. This provides a level of clarity of how the application's state mutates over time that is hard to achieve in a traditional Model-View-Controller[21] (henceforth MVC) approach, in which many controllers have access to the application's state simultaneously.

Further libraries which play a key role within the client-side application are the following:

[10]https://facebook.github.io/react/

redux-thunk

redux-thunk[11] is a helper library which enables the use of asynchronous functions as the aforementioned actions that affect the application's state. This is essential to allow control over how and when the client application sends data to the blockchain, and how it fetches data from the blockchain to then integrate it into its local state.

redux-logger

redux-logger[12] supplements Redux itself by automatically logging actions and the state changes they trigger to the development console. This meant a significant amount of time could be saved during the project by avoiding the need for a custom logger implementation, or worse yet, littering the codebase with sporadic logging statements.

tcomb-form-native

tcomb-form-native[13] provides a framework for forms and form validation in React Native. As form validation in particular can take up an inordinate amount of development time, this library was essential in order to keep the project on track given the severe time constraints in relation to its size.

[11] https://github.com/gaearon/redux-thunk

[12] https://github.com/evgenyrodionov/redux-logger

[13] https://github.com/gcanti/tcomb-form-native

2.3.4 Testing

Composing unit tests for each public function of the system's API was an essential part of the development process, especially as the Tendermint blockchain was able to report errors within its own processes only in the most rudimentary of terms. This meant that if no testing suite was in place, silent breakages and undefined behaviour were likely to occur as the API grew.

The following JavaScript libraries were used to implement unit & integration tests on the server:

Mocha

Mocha[14] was chosen as the testing framework for the project due to its specific aim of simplifying the testing of asynchronous JavaScript code. This clearly applied to the needs of the planned API which would be largely built on asynchronous functions.

Chai

Chai[15] is an assertion library for both test-driven development (TDD) and behaviour-driven development (BDD). Chai's TDD `assert` function was used to keep test assertions simple and testable against an expected result.

[14]https://mochajs.org/

[15]http://chaijs.com/

Istanbul

Istanbul[16] is a code coverage tool which was used to gain an overview of how much of the API was currently covered by tests. This occurred via coverage reports, which were generated after every run of the test suite.

2.3.5 Databases: SQL vs NoSQL vs Blockchain

Using a blockchain as the pivotal data structure backing the system evoked an interesting question: Could the entire system be built in a way that did not, at any point, rely on a traditional database to store data? This would automatically provide the ability to have a distributed database, avoiding any central point of failure and thus the potential for catastrophic data loss. Furthermore, this approach would be the significantly more cohesive one in terms of system architecture. Adding a further external data source to manage auxiliary data which doesn't neatly fit into the duties of the blockchain's smart contracts would have introduced a further agent within the planned API, thus significantly increasing the complexity of coordinating data retrieval and storage operations.
Yet, this approach could also have a significant impact on the size and integrity of the stored data. Unlike with a typical SQL or noSQL database, there is – at the time of writing – no commonly accepted approach as to how data should be structured in terms of smart contracts for persistent storage in a Tendermint blockchain. This meant an initial attempt at implementing an interface to do so would possibly contain inefficiencies and anti-patterns in terms of data relations.

After weighing up the above-mentioned points, the choice was made to go ahead and attempt to store all data generated by the system within the blockchain, as choosing to

[16]https://gotwarlost.github.io/istanbul/

add a further form of database would likely have been significantly more bug-prone and costly in terms of development time.

2.4 Tooling

2.4.1 Requirements and Design Tools

GanttPro[17], an online creator and editor for Gantt charts, was used to create a timeline for the project, while Google Docs[18] in conjunction with draw.io[19] were used to capture requirements and create UML diagrams.
This report was originally written in Markdown and transposed to the Latex format using pandoc[20].

2.4.2 Development Tools

Editor

In order to avoid switching between editors to get the best development support for disparate parts of the project's code, Github's Atom[21] editor was chosen due to its level of customisability and huge selection of plugins. Atom was particularly suited for writing React Native code thanks to Facebook's Nuclide[22] plugin, which enables Atom

[17]https://ganttpro.com/
[18]https://www.google.com/docs/about/
[19]https://www.draw.io/
[20]http://pandoc.org/
[21]https://atom.io/
[22]https://nuclide.io/

to approximate a richness of features typically only found in an Integrated Development Environment (IDE), by offering an in-built debugger, code snippets, and Facebook's own static type analyser which is discussed below.

Static Type Analyser

Flowtype[22], a static type analyser for JavaScript, is a further in-built feature of the Nuclide plugin. Flowtype allows the developer to define a type for a variable, a type signature for a function, and even to create custom union & intersection types. Flowtype then checks whether the defined type specifications are adhered to and warns if it detects a TypeError. This is an incredibly useful tool for a dynamically-typed language such as JavaScript, where unintended type coercion is a common issue.

Debugging

Debugging within the project was performed in three different ways due to the variability of environments within the technology stack. For the React Native app, Google Chrome's DevTools[23] were utilised alongside the Xcode iOS simulator[24] in order to interact with the app and receive log outputs side-by-side in real-time.

For the NodeJS server, the aforementioned `eris-logger` module was used to log API-related logging statements to the terminal.

At the bare-metal level, Eris provided a way of continuously logging the activity of the Tendermint blockchain. Due to all operations approximating those of assembly language and thus being represented in hexadecimal, this was not useful in terms of locating bugs, but it provided a sanity check to ensure that the chain was performing the expected

[23]https://developers.google.com/web/tools/chrome-devtools/?hl=en

[24]https://developer.apple.com/

operations when instructed to do so by the NodeJS server, excluding a possible cause if a bug was being searched for.

Version Control

The `git`[25] command line utility was used to manage the codebase's development over time, in conjunction with GitHub[26] to provide a remote backup. For more involved `git` operations, such as merging branches, Atlassian's SourceTree[27] GUI was used to avoid mistyping complex terminal commands and thus potentially performing unwanted changes to the version history.

Docker & Shell Scripts

Docker[3] is a platform which allows the creation of self-contained virtual environments for software development, to mitigate arbitrary local differences between development environments. It was an essential part of the development process as Eris's tooling leverages Docker heavily to deploy blockchain instances.
Bash shell scripts (see Appendix F) were constructed by the author in order to automate processes such as hydrating the local terminal environment with variables required to run the blockchain, or booting the Docker virtual machine and the local blockchain instance.

[25]https://git-scm.com/
[26]https://github.com/
[27]https://www.atlassian.com/software/sourcetree

Chapter 3

Requirements and Analysis

3.1 Problem Statement

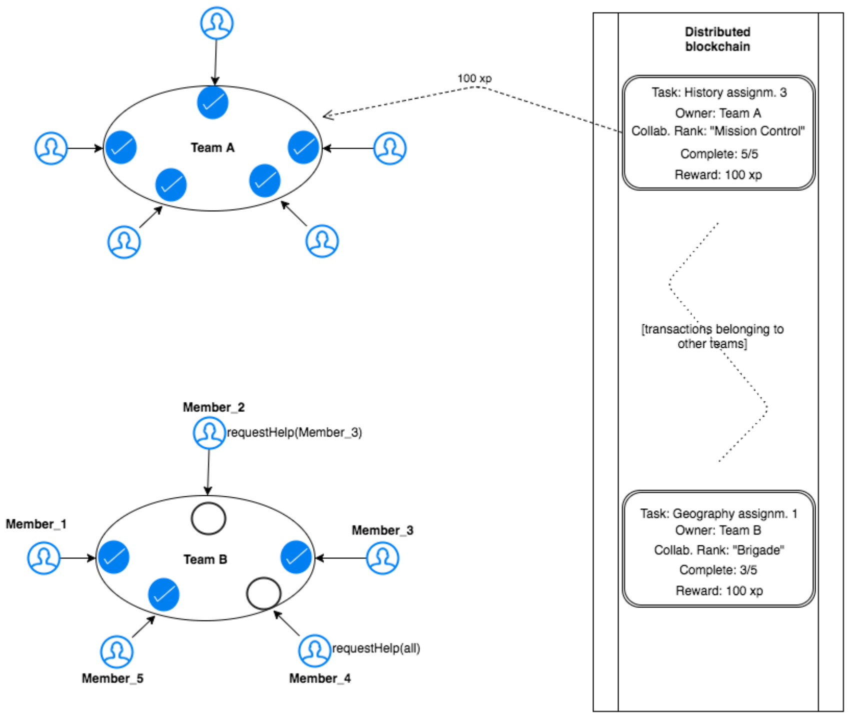


Figure 3.1: An early conceptual diagram demonstrating possible interactions with the system

As outlined in chapters 1 & 2, the key problem the project attempts to solve is the way in which students relate to their school work on an everyday basis by investigating the feasibility of a system which uses a distributed blockchain data structure to verify and store data about events which represent collaborative work.

Figure 3.1 shows how the basic components and interactions within this system might look.

3.2 Requirements

The act of establishing requirements was entirely focused around the question: “What functionality does the system require, at a minimum, to fulfill its stated aims?”. This provided a clear focus on what was absolutely needed for a minimum viable product (henceforth MVP)[23] that an individual and/or group of individuals could use in a meaningful way. This meant both functional and non-functional requirements were focused around three domains: tasks, users, and teams. The requirements were prioritised according to the MoSCoW system[5], with requirements being ranked from “Must Have” through “Should Have” and “Could Have”, with the final category being “Won’t Have (in this development cycle)”. The project’s “Must Have” requirements had to be strictly limited to what was viable within the project’s timeframe, as there were a large number of known unknowns (e.g. the Solidity language) and unknown unknowns (unforeseeable issues with the API, blockchain or development environment). The “Should Have” and “Could Have” categories therefore largely express targets for more sophisticated future iterations of the system.

ID	Functional Requirements	Category	MoSCoW Priority
FRQ1	Users should be able to sign up by providing simply a username, password and, optionally, an email address	Registration	Must
FRQ2	Users should be able to login using a username and password	Login	Must
FRQ3	Any team member should be able to register a task for their team on the network, by specifying team members involved	Tasks	Must
FRQ4	Any team member is able to view both outstanding and completed tasks	Tasks	Must
FRQ5	If user is not a member of a team, they should be able to start a new team by adding other members via their usernames	Team	Must
FRQ6	Upon resolution the task reward should be added to the participants' scores automatically and immediately	Tasks	Must
FRQ7	Users should be able to issue a request to join an existing team or create a new one upon signup	Registration	Should
FRQ8	Users should receive sign up confirmation to verify email address	Registration	Should
FRQ9	Any team member should be able to opt out of a task they were entered for if they are not in fact involved	Tasks	Should
FRQ10	Each team member can see how many other members have completed a given task without identities being revealed in the process (i.e. by simply showing a count or percentage)	Tasks	Should

FRQ11	Individuals should be able to provide public links to their completed tasks and general profile	Tasks	Should
FRQ12	Users should be able to see their team's current global ranking	Team	Should
FRQ13	Users should be able to recover their password if they have forgotten it	Login	Should
FRQ14	Users should be able to view incomplete tasks their involved in and past completed tasks	Tasks	Should
FRQ15	Users should be able to upload attachments related to their tasks	Tasks	Should
FRQ16	Tasks should automatically resolve to status "Complete" once all participants have completed them	Tasks	Should
FRQ17	Users should be able to edit their account details after signing up (email address, password);	User	Should
FRQ18	Users should be able to delete their account	Settings	Should
FRQ19	An individual can request help for a task from other task participants	Team	Could
FRQ20	Members may offer help to the rest of the team without a specific request being present, to speed up any potential rendez-vous (i.e. set a flag on that user for a given task)	Team	Could
FRQ21	Once a rendez-vous occurs, the parties involved should be able to chat with each other	Team	Could
FRQ22	Team members should be notified via a mobile alert if a help request affects them	Team	Could
FRQ23	Users could be able to upload a profile picture	User	Could

FRQ24	Users should be able to mute help request and help offer notifications	Settings	Could

Table 3.1: Functional Requirements

ID	Non-Functional Requirements	Category	MoSCoW Priority
NFRQ1	A User who is not logged in is prevented from accessing the app	Security	Must
NFRQ2	The system shall store all user passwords in the blockchain in an encrypted format	Security	Must
NFRQ3	The app should provide offline sync, meaning that any state changes made during a period of no-connection should be saved and synced with the DB at the next possible instance	Persistence	Should
NFRQ4	A team member should be able to signal a task as complete within a maximum of 1∼2 UI interactions	Tasks	Should
NFRQ5	The system should accomodate for at least 1,000 teams	Capacity	Should
NFRQ6	The system should accomodate for at least 100,000 tasks	Capacity	Should
NFRQ7	The system should log in a user within approximately 5 seconds	UX	Should
NFRQ8	The system shall respond to any interaction in no more than 10 seconds	UX	Should

Table 3.2: Non-Functional Requirements

3.3 Use Cases

To remain realistic in regards to the allotted time for the project, use cases were aimed at providing a full exploration of the API rather than a rich user experience within the first iteration of the system, therefore ensuring that the API could remain agnostic regarding any particular client-side context.
Use cases for the client-side application and the API itself were straightforward, as the only actors within the initial scope of the system were the individual user and the user as part of a team, meaning there was no need for elevated permissions within the API or specialised UI interactions, as one might see with an administrative dashboard in other applications.

Use cases were constructed in parallel with the initial UI sketches, thus helping to visualise the flow of user-system interactions.
The table below represents an overview of the uses cases that were constructed, while the detailed use cases can be reviewed in Appendix B.

ID	Use Case	Primary Actor	Secondary Actor
UC1	Signup	User	System
UC2	Login	User	System
UC3	AddTask	User	System
UC4	ViewTasks	User	System
UC5	TaskComment	User	System
UC6	OptOutOfTask	User	System
UC7	CheckTaskCompletion	User	System
UC8	ExternalLinkToTask	User	System
UC9	CreateTeam	User	System

ID	Use Case	Primary Actor	Secondary Actor
UC10	CheckTeamScore	User	System
UC11	RequestHelp	User	System
UC12	OfferHelp	User	System
UC13	MuteNotifications	User	System
UC14	EditAccount	User	System
UC15	SetProfilePicture	User	System
UC16	DeleteAccount	User	System
UC17	AttachTaskFile	User	System

Table 3.3: Use Case Overview

3.4 Sketches

To get a sense of a viable layout and structure for the client-side app, the author used rough sketches to visualise each view that was needed to meet the requirements. Once a convincing layout had been established for a view, the author created a static mockup in React's JSX, which could then be broken down into separate components and given dynamic properties, such as data retrieval methods, at a later point.

3.5 System Design

The design of the system's architecture started with creating a simple deployment diagram (Figure 3.2), which provided a high-level view of the system's required components and the roles they would play.

Establishing a high-level understanding of what the system required to provide proficient communication between any client application and the Tendermint blockchain was an essential aspect of the initial design phase. Decoupling the system's functionality from the specific implementation with which the system may be represented to a user was a crucial provision to ensure maximum reusability and compatibility of the system, therefore following the common software engineering mantra of *"program to an interface, not an implementation"*[24]. Within the context of the web, the most appropriate form to follow this approach was by developing a RESTful[25] API, thus providing a uniform interface of data endpoints, regardless of the client requesting said data.

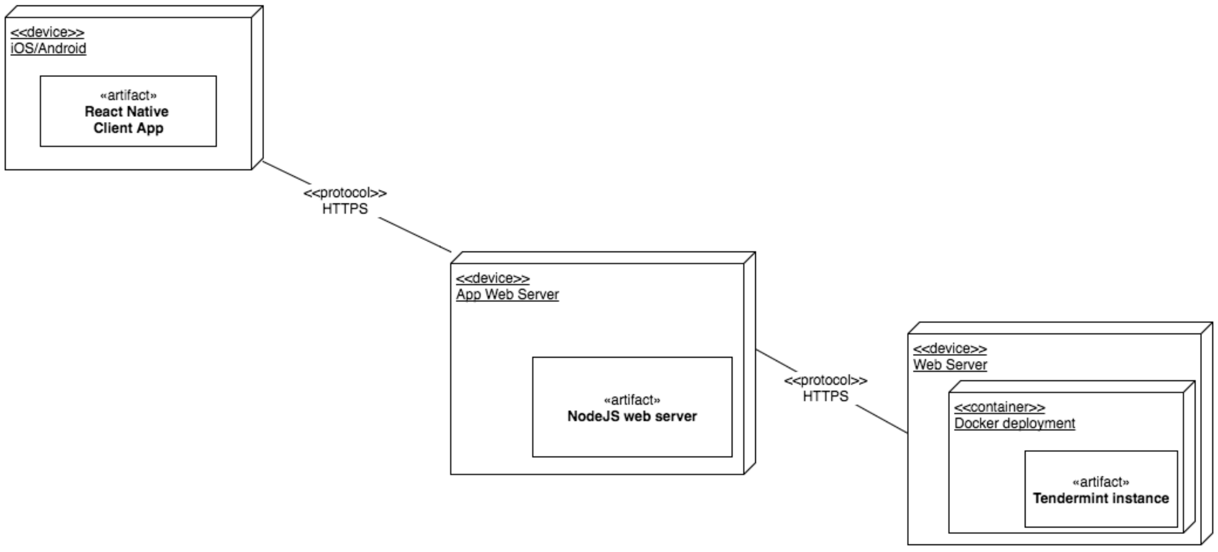


Figure 3.2: Deployment diagram showing the system's main components

3.5.1 Smart Contract Analysis

The role of the smart contracts played in this project – rather than attempting to literally track and enforce contractual obligations between multiple parties – was more closely related to that of typical classes in object-oriented programming, with the majority of the contracts either acting as factories[26] for composite types or implementing operations on these types as “manager” contracts.

Factory Contracts

In order to define composite types, known in Solidity as `struct`s, a contract was defined for each domain of the system as revealed by the requirements. This meant there was a need for `User`, `Team` and `Task` contracts.

Data Structure Contract

Solidity – as a young and constantly developing language for smart contracts – does not currently provide a way to iterate over storage arrays. The language’s version of objects, the `mapping` type, which approximates what is usually known as a hash map, is also non-iterable. This meant a custom data structure contract was needed to store, edit and retrieve collections of type contracts.

Manager Contracts

To avoid violating the Single Responsibility Principle[27] of object-oriented design, there was a need for a set of contracts which would perform operations on instances returned by the factory contracts. This meant there was a need for a `UserManager`, `TeamManager` and `TaskManager` contract.

Linker Contract

The previously discussed decision to encapsulate all data generated by the system within the blockchain led to an issue of data relations amongst the contracts: How could a new `Task` contract be linked to the appropriate `User` contract if there was no structured way to do so in comparison to, for example, an SQL interface? This meant there was a need for a higher-order `Linker` contract, dedicated to locating and linking related contracts in such a situation.

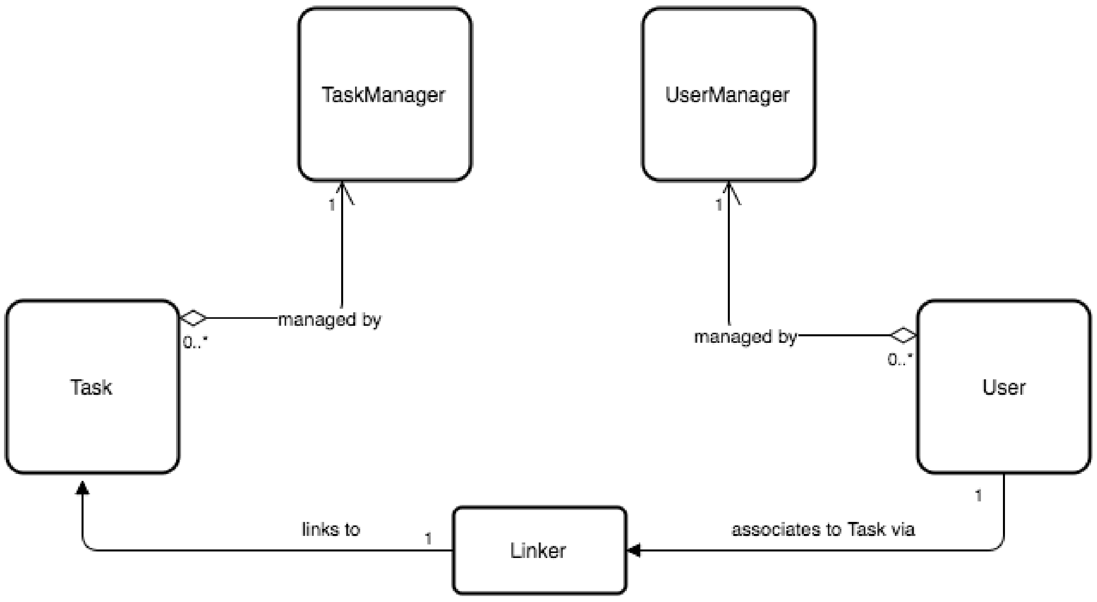


Figure 3.3: A simplified class diagram to illustrate the relationships between Factory, Manager and Linker contracts

3.5.2 Server-side Analysis

API Router

Once the structure of the smart contracts had been established, the analysis of the server-side implementation quickly revealed that the server's structure would largely mirror that of the smart contracts. This was the case partially due to the way Eris's JavaScript library was implemented, but largely due to the fact that this would keep the the final API succinct and free of unnecessary cognitive load for the author and for any future developers.

The server would therefore simply act as a relay and transformer for data travelling between the client-side application and the blockchain, acting as a *de facto* middleman. In more concrete terms, this would involve calling relevant contract methods on the blockchain when a certain API endpoint was requested and transforming data between hexadecimal and UTF-8 encoding, for example. This is necessary due to Solidity's poor support for strings at the time of writing, meaning that all strings would have to be encoded into 32-byte fields of hexadecimal, known in Solidity as the `bytes32` type.

Uploader

Meeting the requirement of letting task participants attach files related to the their tasks was trickier than anticipated, as choosing to develop the system's client-side as a mobile application came with a key drawback: file management. Mobile operating systems provide only limited capabilities to edit and manage even simple text files, meaning that the platform was suboptimal for attaching files related to a task. For example, if the task involves writing a report, task participants may want to attach the final report to the task in the blockchain, thus further corroborating their participation in it.

To resolve this usability bottleneck, the author decided to utilise the server not just as a headless API router, but also as an uploader, composed of a single web page, to attach files to tasks present in the blockchain. To implement the uploader effectively, two key aspects had to be considered:

UX consistency - From a usability perspective, the transition between the mobile app interface and the browser-based uploader should be as seamless as possible. To avoid disorienting the user, the uploader was therefore kept as straightforward as possible; a single page with a form and a submit button.

Security - The user would have to enter a token – issued by the mobile app when a new task is created – to confirm that the upload has a legitimate task associated to it. The use of tokens therefore prevents automated spam and enables the identification of users uploading material which may be malicious or illegal in nature.

3.5.3 Client-side Analysis

Abstracting view components with React

React takes markedly different approach to view templating compared to the other major web development frameworks such as Google's AngularJS[1]. React's philosophy focuses on breaking a view down into its constituent parts to form reusable components. These components can then be composed in whichever way the developer sees fit. This level of composability comes to its full potential in React Native, as it enables the abstraction of common UI components within a mobile app, providing ample opportunity to reuse said components even across different platforms. The decision to abstract a given component was therefore made according to the following criteria, ordered by highest weighting first:

[1]https://angularjs.org/

Frequency & variability of use - How sensible it was to abstract a specific component into a more generic one heavily depended on the range of use cases it could be applied to. For example, creating a generic navigation bar component which could contain different buttons for different views made sense, whereas creating an abstract component for the user's profile summary did not, since it would only be used a single time, in a single place within the app.

Cross-platform potential - A secondary consideration was whether the abstraction would be useful across platforms, i.e. between iOS and Android. For the initial iteration of QuantiTeam, which would only focus on iOS, this aspect was less significant than the frequency of use, but it played a significant role in keeping the client-side application open to extension at a later point nonetheless.

Managing state with Redux

While considering the responsibilities that a client-side implementation would have to fulfill to meet the established requirements, it also became clear that even for a simple implementation there was a considerable amount of application state that would have to be managed by the client. Besides being the most common choice within React Native applications, the Redux library provides a well-structured approach to state management, derived from the philosophy behind the Elm[2] programming language. Redux ensures that events which mutate state are well-defined and that they may only take place in a single direction (unidirectionally).
In concrete terms, Redux achieves these orderly state mutations by following three interdependent principles[20]:

Single source of truth - All of the application's state is stored in a single object tree known

[2]http://elm-lang.org/

as the `store`.

State is read-only - Actions, which are objects describing a possible state mutation, are the only way to modify the store and are therefore its only source of information.

Mutation through pure functions - Reducers, a type of pure function, are used to take the application's current state object in conjunction with an action describing a state mutation, to return a new state object.

Based on these 3 principles which impose the need for a `store` object tree, actions and reducers, state mutations within Redux can be visualised in the following manner:

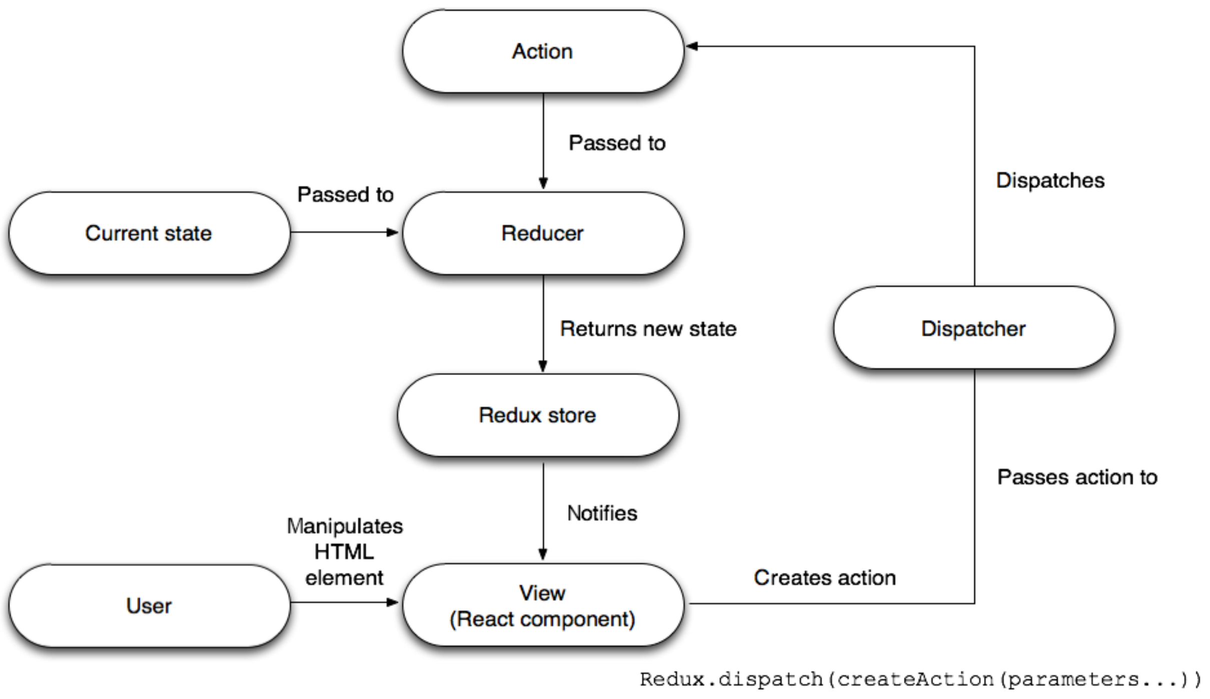


Figure 3.4: State mutations within a Redux architecture[28]

Chapter 4

Design & Implementation

4.1 System Architecture

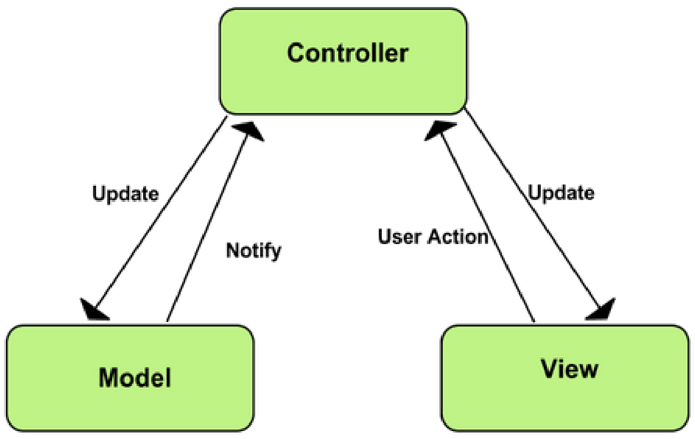


Figure 4.1: A typical MVC Model[29]

QuantiTeam broadly follows the three-tier architecture of a typical Model-View-Controller[21] application with the important distinction that the roles within the MVC pattern are applied to an entire system of various applications, rather than a single application. In concrete terms, this means that blockchain represents the Model element by establishing the system's data model through the smart contracts applied to it, while the NodeJS server represents the Controller element, providing a public interface for client applications to issue requests to the blockchain and handling raw responses from the blockchain. The View element of the implementation is therefore interchangeable, as the RESTful API formed by the web server and the blockchain provides a uniform set of endpoints to communicate with, tying no special or unique value to the client, in this case a mobile app.

4.2 Blockchain Design & Implementation

4.2.1 The Model

Within the MVC paradigm, models represent the central structure of the application and “are concerned with neither the user-interface nor presentation layers but instead represent unique forms of data that an application may require”[30].
The Tendermint blockchain serves as a model by defining the system’s domain through the collection of smart contracts it holds, thus setting the boundaries for what kinds of data the system is able to store and what kind of operations can be performed on the data.

4.2.2 Working with Solidity

To put a number of the design decisions made during the construction of the blockchain’s smart contracts into perspective, some context regarding the current capabilities of the Solidity programming language is required.

Strings and Numbers

As was briefly touched upon in chapter 3, Solidity is still in its infancy. This meant that some data types such as strings, had to be translated into fields of 32 byte arrays in hexadecimal when fed into the blockchain and translated vice versa when being retrieved from the blockchain.

Furthermore, translating numeric values from a dynamically-typed language such as JavaScript to a strict, statically-typed one such as Solidity without side effects or data corruption is also no trivial task. A point which illustrates these possibly unpredictable side effects is the possibility of an unusually large integer being fed into the API by a client application, which is then translated to an 8-bit unsigned integer (`uint8`) field within a Solidity contract, thus writing an irreparably truncated and corrupted value to the blockchain.
A conscious decision was therefore made to encode all values – aside from booleans which are able to move across the API unaltered – in hexadecimal before they are fed into the blockchain, regardless of initial type, to utilise Solidity's `bytes32` type. This approach provided two important advantages:

Predictability - Transforming all data associated with the blockchain in a regular manner enhances the testability of the API by fixing the data's representation, both when it is entered into and retrieved from the chain. This increases the API's level testability by making expected outputs for a given input more uniform and free of special circumstances.

Simplicity - Establishing a standardised format for any data to be entered into the blockchain helped keep the API straightforward to work with. Any numeric or string data could be encoded for the chain by the NodeJS server using the `eris-wrapper` library module's `str2hex()` (string-to-hex) function, and decoded using its `hex2str()` (hex-to-string) function.

Arrays and Objects

A further hindrance imposed by Solidity was its poor support for arrays and its lack of objects, also known as lists and dictionaries, respectively.

While JavaScript is composed entirely of objects, Solidity provides the `mapping` type;

a data structure similar to what is commonly known as a hash map. This would have provided a sufficient parallel to translate data formatted in JavaScript Object Notation[1] (henceforth JSON) into a format digestible by Solidity contracts, were it not for the fact that Solidity's ability to store and perform operations on its mappings and arrays is limited at best. For example, storing user profiles (i.e. `structs`) in a dictionary (i.e. a `mapping`) would have been possible, but retrieving a particular collection of dictionary entries would have been impossible, as neither Solidity's arrays nor its `mapping` type are iterable data structures by themselves. More specifically, a `mapping` can only be "probed" for a specific value, making it impossible to iterate over the structure and retrieve a subset of its properties which match a given criteria. This would have meant searching the blockchain in any non-trivial manner would have been highly unfeasible.
To mitigate this functional bottleneck in Solidity, the author adopted an implementation of a SequenceArray[31], which became the `SequenceArray.sol` contract. The SequenceArray provided a well-defined interface of methods on top of Solidity's native array and `mapping` types, allowing the author to focus on *when* and *why* data should be manipulated or searched, rather than *how* these operations are performed.

4.2.3 Smart Contracts

Data: Factory Contracts

The blockchain's factory contracts were a straightforward undertaking, both in regards of design and implementation, as their only role was to return a new instance of the contract, which would then be handled by the associated manager contracted.

Solidity's inability to process an externally passed JavaScript object is nicely exemplified

[1]http://www.json.org/

by the `Task.sol` factory contract (see Appendix F), due to the sheer verbosity of the constructor function. While a task could be encapsulated as a single object and therefore passed as a single parameter within the client- and server-side JavaScript applications, the object had to be split into its constituent fields by the NodeJS server before it could be handed to the blockchain. This created what is commonly referred to as a “code smell”[32], by forcing the author to implement an excessively large amount of parameters. This made both the factory contracts themselves and the server’s methods that deconstructed the initial JavaScript object for the blockchain brittle, as any change to a factory contract’s fields had cascading effects throughout the API, thus creating an unnecessary opportunity for bugs to appear if refactoring was not done in a meticulous manner.

Operations: Manager Contracts

As a contract type, Manager contracts are chiefly responsible for indexing and modifying instances of data domain contracts, returned to them by their respective factory contract. How a Manager-type contract fulfills this role is exemplified with the code snippet below, which is an extract from the `UserManager` contract’s `addUser()` method:

```
// ...
if (isOverwrite) {
    return 0x0;
} else {
    User u = new User(_id, _username, _email, _name, _password);
    userList.insert(_username, u);
    return u;
}
// ...
```

The method previously checks whether the username already exists in the `userList` sequence array and writes the result to the `isOverwrite` variable, indicating that this would be an "overwrite" rather than a "create" operation. If the username already exists `addUser()` returns `0x0`, a null pointer address, thus indicating to the NodeJS server that adding the user failed. This again exemplifies the rather primitive state of the Solidity language at the time of writing, as the null address has to serve as an implicit error due to the absence of an error type within the language.
If `isOverwrite` is false on the other hand, the `UserManager` contract creates a new instance of the `User` factory contract by invoking `new User(...)`, returning a pointer address which indicates the position of the newly created `User` contract in the blockchain. This address is then inserted into `userList`, where entries are indexed by using the passed `_username` variable as a key, to facilitate later retrieval.

Furthermore, the flexibility provided by the data structure contract – in this case a sequence array – became especially clear during the implementation of the Manager contracts. A Manager contract could simply instantiate a `SequenceArray.sol` contract for its own purposes and use only the minimum amount of SequenceArray methods it required to fulfill its functions, by creating a wrapper function around the method, and adding additional context as required, such as logging an event:

```
contract UserManager {
    SequenceArray userList = new SequenceArray();

    // other methods...

    function isUsernameTaken(bytes32 _username) constant returns (bool) {
        registerActionEvent("IS USERNAME TAKEN");
        return userList.exists(_username);
    }
    // ...
}
```

Relations: Linker Contract

The Linker contract, which is best described as a utility contract, is responsible for linking together instances of factory contracts. As one of the blockchain's primary purposes within QuantiTeam is to act as persistent storage layer, this type of functionality was required in order to emulate the entity integrity provided by primary key/foreign key relations between tables in a relational database[33].

For example, when a user creates a new task, thus spawning a new `Task` contract via the `TaskManager`, the system should be able to later identify the creator of said task. A possible solution would have been to simply let the `TaskManager` contract establish the link itself. Although being the most obvious solution to the issue, it would not have scaled well across the suite of Manager-type contracts. As the author was aware that a similar need for linkage would arise again between `Team` and `Task` contracts, allowing each Manager-type contract to implement its own linking mechanism was a notion which seemed to call for a layer of abstraction. A separate `Linker` contract was therefore established, whose sole responsibility is to create relational links between different types of factory contract instances. To achieve this, each new instance of a `User` or `Team` contract also contains a `taskAddressList` sequence array. This list is used to hold addresses (i.e. pointers) of tasks related to this instance of a user or team within the system. The `Linker` contract is then responsible for adding relevant addresses to the `taskAddressList`, an action which is performed whenever a new task is created.

The following sequence diagram illustrates how the system processes a user attempting to add a task to the blockchain, exemplifying how and when the `Linker` contract is invoked.

Appendix A contains a further sequence diagram and a full class diagram for the system's smart contracts.

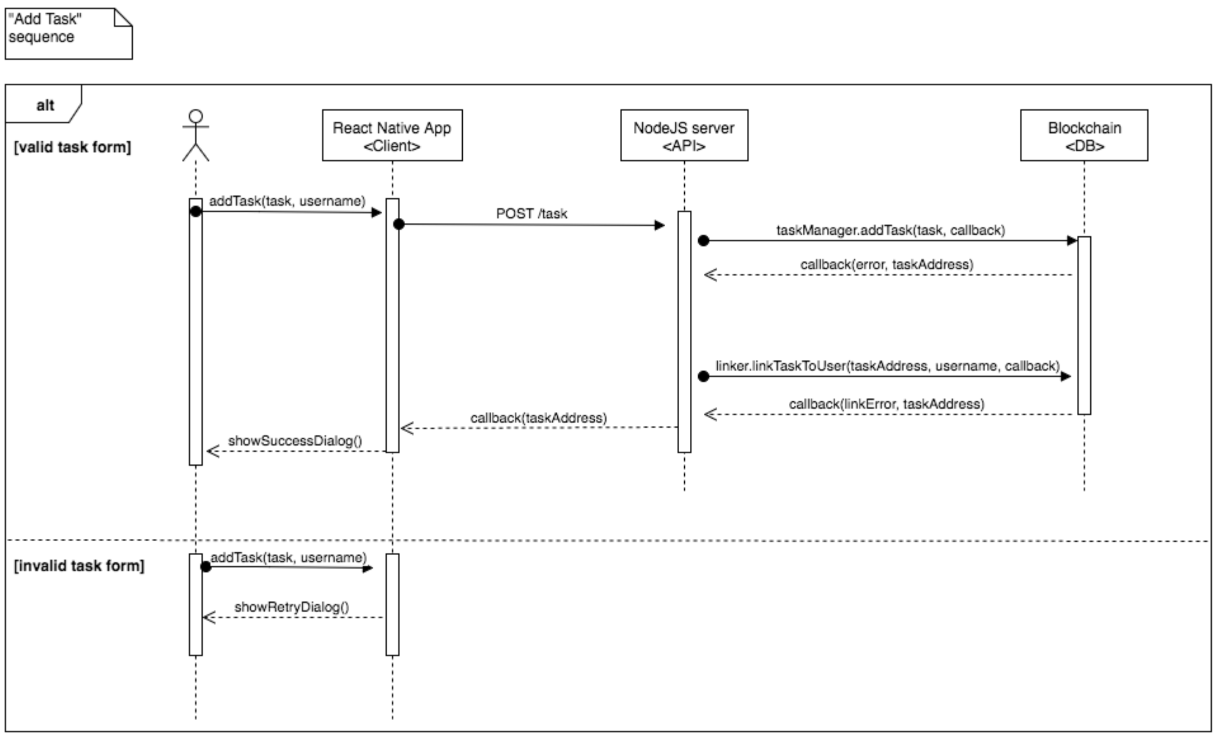


Figure 4.2: Sequence of events for an "Add Task" action

4.3 Server Design & Implementation

As touched upon in the server-side analysis, the REST API server's role is first and foremost that of a data transformer and relay, forming a bridge between the blockchain and any given client-side implementation. The following subsections initially present how the server was designed to adhere to principles of both the MVC and REST design patterns, followed by an exploration how the server performs its bridging responsibilities in concrete terms.

4.3.1 The Controller

Controllers can be regarded as an intermediary which sit between models and views, and are typically responsible for updating the model when changes in the view take place[30]. Within QuantiTeam, the server is able to fulfill its role as a Controller component within the system's overarching MVC architecture, by being the deciding factor concerning the logic executed between the moment an HTTP request is received and the moment a response issued from the API. Furthermore, the server is also responsible for piping any changes in data in the client-side application to the blockchain. For example, the event of a user marking a task as "Completed" is persisted by the server forwarding this change to the blockchain and altering the relevant `Task` contract accordingly.

4.3.2 RESTfulness

REST, which is an acronym for Representational State Transfer, is a commonly used web development pattern which attempts to ensure reliability and scalability for the web service implementing it[25]. Within the context of QuantiTeam, achieving RESTfulness was key for the system's API, as this would help ensure the system's usefulness to any context of client-side implementation, rather than specifically a mobile paradigm.
This subsection therefore describes the properties of a RESTful service and how each applies to QuantiTeam's server-side architecture.

Client-Server Dichotomy

Crucial to the creation of an implementation-agnostic REST interface is a separation of concerns between the client and server[25]. Strictly separating client and the server roles from each other provides portability and replaceability, as the underlying implementation

of either may change without affecting the standardised form of communication established by the REST interface.
Within QuantiTeam, there is a clear client-server dichotomy between the server which is solely concerned with handling incoming requests, and the React Native client app, which requests data from the server and then decides how to represent said data to the user within its local state.

Stateless

Statelessness is a key constraint within the REST design, which holds that each request should contain all information required by the service to process the request, and the service's response should contain all the required data to fulfill said request[25].
Applied to QuantiTeam, the REST pattern's property of statelessness not only helped to decouple the system's architecture by avoiding the need for managing state across different applications, it was also the most feasible approach to enable a straightforward way of communicating with the Tendermint blockchain, due to the previously described frequent requirement to transform data as it travelled to and from the blockchain. Statelessness, in this context, meant that there was no need to worry about intermediate representations of the data being stored and inappropriately forwarded to either the client or the blockchain at a later point in time.

A simple example of how the server remains stateless while fulfilling its function as an API interface is shown in the following snippet, taken from the `server.js` module:

```
app.post('/user/taken', function (req, res) {
    var username = req.body.username;

    log.info("POST /user/taken: ", username);
    userManager.isUsernameTaken(username, function (err, isTaken) {
        _handleErr(err, res);
        res.json({isTaken: isTaken});
    });
});
```

This example shows the `/user/taken` API endpoint, which is responsible for validating the availability of usernames. When a user tries to sign up in the client-side app, the server receives the proposed username in the request body (the only piece of information required for the API to fulfill the request), with which it calls the `userManager`'s `isUsernameTaken()` method. The method returns an `isTaken` boolean which is written to the `res` response object, thus providing all data necessary for the client to make a decision regarding the availability of the proposed username.

Cacheable

Responses from the REST service being implicitly or explicitly declared as cacheable or non-cacheable is a further component of a RESTful service[25], as cacheable responses provide an opportunity to to optimise the amount of client-server communication that is needed.

QuantiTeam's API currently does not provide explicit cache labels in its responses and is therefore implicitly cacheable on the client side. While cacheability is key to scaling a developed API, investing significant amounts of time to implement explicit cache invalidation within QuantiTeam's first iteration was outside of the project's scope. Implicit caching takes place within the React Native client-side app, which retrieves and then locally retains static information such as the user's profile data, for example.

Uniform Interface

A uniform technical interface – which provides a generic, high-level method of communication across all services within a REST architecture – is seen as the primary constraint which distinguishes the REST approach from other approaches to web service architectures[25]. For QuantiTeam, this interface was constructed by using HTTP URI paths to define API endpoints, which a client could then issue requests to. Uniformity was further enforced by creating the following semantic template which all of the URI paths defined for the API would follow:

```
/<data-domain>/<optional-subdomain>/:<optional-parameter>
```

This URI template declares that the first path segment is mandatory and shall be the data domain the request is directed towards, for example `/user` or `/tasks`.
The second segment may be composed of a further specification within the data domain, such as `/profile` for `/user`, thus forming the path `/user/profile`.
The third and final segment consists of an optional request parameter, such as `:username` which may be used to pass a parameter to the API in a HTTP GET request. Building on the previous example, we may therefore be able to request the profile data for user `foo` by issuing a request to the API via the path `/profile/user/foo`.

By requiring all interactions with the API to take place in this form, the system is thus able to guarantee a high level of independence from concrete implementation details, as the requirements and formatting of HTTP do not vary across implementation contexts.

Layered System

The final constraint within the REST pattern concerns composability. A RESTful implementation should allow for intermediate layers, commonly known as middleware, to be inserted into the service without affecting the interface for communication in any way[25]. QuantiTeam meets this constraint through its HTTP URI interface, which enables middleware to implement the same interface and forward a given request to the service itself using the same exact URI once it has completed its part of processing the request.

4.3.3 Data Handling

An essential function of the NodeJS server is to act as data handler, thus defining the way data has to be formatted to flow between the React Native client app and the blockchain. As the server implements an API interface which should be suitable for any client-side implementation, the data transformations required took the form of translating JavaScript objects coming from the client to a hexadecimal format in order to adhere to Solidity's `bytes32` type, for the reasons that were laid out in chapter 3. This was achieved by use of a "pipeline" function.
The pipeline function was defined within the utility module `chainUtils.js` as `marshalForChain(<data-object>)`, which can be reviewed in full in Appendix F. The function takes its `<data-object>` parameter, identifies the type of each object property (e.g. `Array.isArray(<property>)`) and transforms it into a string representation of itself. Representing all of the `<data-object>`s properties as strings in an intermediate step is necessary in order to utilise the `eris-wrapper` library module's `str2hex(<string>)` (string-to-hex) function, which accepts a string as a parameter and transforms it into a 32-byte hexadecimal string. `str2hex` is therefore the final step in the `marshalForChain` function, preparing each of the object's properties to be fed into the blockchain.

When retrieving data from the blockchain, the transformations are reversed via `eris-wrapper`'s `convertibleCallback()` function, which the author modified to accept an array of transformation functions, rather than a single function. This enables a similar pipeline effect to that of the `marshalForChain` function, as a property that is known to be an array can therefore be transformed back to this representation after it has been decoded from hexadecimal to a string via the `hex2str` (hex-to-string) function.

```
contract.participants( eris.convertibleCallback(callback, [eris.hex2str,
JSON.parse]) )
```

The snippet above shows `convertibleCallback` is invoked on a `Task` contract's `participants` field. Aside from being passed the `callback` parameter which will receive the transformation's result, the function also receives an array of transformation functions; in this case `hex2str` to decode the hexadecimal string, followed by `JSON.parse` to convert the string back into its original form: a JavaScript array.

4.4 Client-side Design & Implementation

4.4.1 The View

Within the MVC pattern, views are the component responsible for the graphical representation of the application's – or in this case the system's – data models[21]. Although a view may also be described as "a visual representation of models that present a filtered view of the current state"[30], the parallels between the MVC pattern and QuantiTeam's structure become somewhat less applicable. While the client-side React Native app does of course act as a filtered graphical representation of the blockchain's models, it also contains its own local state and therefore deviates from the typical description of an MVC pattern's view.

4.4.2 JavaScript: Emulating Strict Typing

One of the key elements in designing and implementing a well-defined client-side application for this system was the ability to define types in a static manner and compose union and intersection types with Facebook's Flowtype. The author found that having to think in terms of explicit, strict type constraints made the React Native app's code more robust and helped create better abstractions, as JavaScript's dynamically-typed nature seemed more of a hindrance rather than a tool when the goal was to enforce types between a client and the system's API.
A pertinent example of how Flowtype helped define more robust React components is the `Tab` type:

```
type Tab =
    'tasks'
  | 'team'
  | 'me'
  ;
```

Here Flowtype allows the definition of a `Tab` enum by using literal types[22]. The `Tab` enum was useful to specify which string identifiers, each associated with a given view, were legal values in the `TabsView` component, which renders the navigation bar at the bottom of the app's viewport.

While the ability to define enums was certainly handy, Flowtype's true usefulness is revealed when looking at one of the system's key data types, the `User` type:

```
type User = {
    id: number;
    name: string;
    username: string;
    score: number;
    teamname?: string;
    email?: string;
    address?: string;
}
```

The snippet above shows how Flowtype enabled the author to define what type primitives a `User` object's fields should adhere to. This significantly reduced bug frequency, both within the app and the API itself, by catching inadvertent type coercions or possibly undefined values before the app's runtime environment was even entered.

Flowtype elevated the ability to define actions and the expected types of their payloads in a more fine-grained manner compared to regular JavaScript, by enabling the author to add type constraints to variables and functions where needed.

```
{ type: 'FETCH_TASKS_SUCCESS', tasks: Array<Task>, receivedAt: number }
```

The action above is triggered upon successfully fetching a user's tasks from from the blockchain via the API. The payload includes a `tasks` property, which is expected to be an array of `Task` type objects, and a `receivedAt` property, which should be a Unix timestamp and is therefore expected to be of the type `number`.

4.4.3 React: Common Components

As explained in chapter 3, React's approach towards view components aims at enabling the developer to achieve a high level of reusability and adaptability from said components. The following section therefore shows how the two component attributes discussed in chapter 3, namely frequency/variability of use and cross-platform potential, were applied to QuantiTeam's React components.

Frequency & Variability of Use

A component which encapsulates both these attributes is the `Header` component. The `Header` component creates a generic framework for the app's header which usually contains navigation and action buttons, such as “Go Back” or “Add Task”. This means it has to appear within almost all of the app's views, clearly meeting the frequency of use attribute. From a variability perspective, the `Header` component is high-level and agnostic to specific implementation details. For example, the header's `leftItem` and `rightItem` attributes, representing placeholders for specific buttons, can be implemented by using a text-based or icon-based button. This permits for concrete instances of the `Header` component to apply the button layout most suitable for a given situation. For example, a “Settings” button is commonly identifiable as a cog icon, whereas the “Add Task” action button has no commonly identifiable icon, and therefore benefits from simply displaying text to disambiguate what action the button will perform if tapped.

Cross-Platform Potential

The flexibility of the app header's button layouts also comes into play regarding the component's potential for reuse across differing mobile platforms. If neither a text- nor icon-option is specified in an instance of the `Header` component, React Native is able to identify the current device's platform via its `Platform.OS` variable and can thus resort to the current platform's defaults (text buttons on iOS, icon buttons on Android). This enhanced the flexibility and robustness of the app overall, as the header would always be able to render its button elements, instead of simply throwing an error due to lack of a defined layout on one platform or another. Furthermore, the aforementioned `Platform.OS` variable provided the opportunity to define standard behaviour for whichever platform the `Header` component was to be rendered on, as shown here:

```
let STATUS_BAR_HEIGHT = Platform.OS === 'ios' ? 20 : 25;
let HEADER_HEIGHT = Platform.OS === 'ios'
    ? 44 + STATUS_BAR_HEIGHT
    : 56 + STATUS_BAR_HEIGHT;
```

Here the header's height is automatically determined according to the current operating system's defaults, providing a clean high-level abstraction that avoided the need for multiple `HEADER_HEIGHT` definitions. Applied to only this single instance this may seems like a trivial abstraction, but it provided the author with a useful mechanism to avoid unnecessary code reuse and redefinition in a number of cases, keeping the `Header` component more transparent and less verbose.

4.4.4 Redux: UI and I/O

To apply the Redux philosophy to QuantiTeam, actions and reducers were modularised into a schema which follows the system's data domain (users, teams, tasks) as shown below:

```
...
 reducers
    rootReducer.js
    tasks.js
    team.js
    user.js
...
```

Splitting reducers in this manner ensured that the reasoning involved in managing the client application's state could be broken down into its constituent parts, providing minimal cognitive load when processing data coming from the system's API.

The client-side application's actions were broadly split along the lines of UI-based and I/O-based interactions, representing synchronous and asynchronous actions, respectively. During this first iteration of QuantiTeam the author was focused on providing a useful graphical representation of the system's API, the vast majority of actions were of the more complex asynchronous I/O-bound type, required to interact with the HTTP API.

Synchronous UI Actions

UI actions within QuantiTeam are utilised to regulate animations and transitions within the application's UI and are limited to manipulating the application's local state only.
A useful example of a UI action that is frequently invoked is the `REFRESH_TASKLIST` action. This action is triggered when a user pulls down on their screen to refresh their list of tasks, thus setting the `didRefresh` boolean flag in the app's state (the `store`) to `true`, which in turn causes a loading icon to be shown to the user. As the `didRefresh` flag can only be reset by the asynchronous `FETCH_TASKS_SUCCESS` action, indicating that the tasks have been retrieved from the blockchain, the loading icon is displayed until this action is triggered. This exemplifies how Redux imbues the application's UI state with precise controls and succinct behaviour if actions are used effectively.

Asynchronous I/O Actions

I/O actions within QuantiTeam are structured to enable reliable communication with the system's API. In abstract terms, any event within the client-side application which required data from or sent data to the blockchain, follows a pattern involving three types of Redux actions[20]:

Request - A Request action is dispatched the moment the client-side application registers an event that requires an API interaction to be completed, indicating that either a Success or Failure action should soon follow. Using a user signup event as an example, the `SIGNUP_REQUEST` action is dispatched along with the data from the signup form filled in by the user.

Success - If the request issued to the API succeeds and receives a valid response, a Success action is dispatched with said response, which is in turn handled by its associated reducer, thus incorporating the new data into the application's state. Within a user signup event flow, this would dispatch the `SIGNUP_SUCCESS` action to the `user` reducer, along with the new `User` contract's blockchain address pointer as its payload.

Failure - If the request issued to the API fails for any reason, a Failure action is dispatched, which includes the error message returned by the failed attempt to communicate with the API. In the context of a user's signup, this scenario would dispatch the `SIGNUP_FAIL` action, logging the associated error the console.

By combining these three action types, the application's I/O-bound interactions with the API follow a standardised sequence, thus providing clear and deterministic behaviour, as the methodology of retrieving data remains the same while the underlying data being operated upon may change.

Chapter 5

Testing

5.1 Testing Strategy

Once the required communication abilities for the system had been established, meaning the author was able to trigger smart contract events within the blockchain via HTTP requests, the system's API was constructed and expanded via test-driven development. This meant that any new functions of the system would be developed in the following manner[34]:

1. A new test is added for the unwritten, proposed function. This ensures that the functions requirements are clear from the outset.
2. The test suite is run and the new test should fail. This excludes false positives (i.e. the test always passes) and confirms that the functionality does not in fact exist yet.
3. The minimum amount of code necessary to make the new test pass is written.
4. The test suite is run again. If tests fail, adjustments are made to the new function until all tests pass.

The system was therefore tested through a sequence of tests which became increasingly high-level over time as its functionality grew; beginning with modular unit tests, then moving onto integration tests for the API as a whole, and concluding with functional testing from the perspective of the client-side GUI.

5.2 Unit Testing

Within QuantiTeam, unit tests are focused on providing extensive coverage for all public API functions (known in JavaScript as “exported functions”) of the server’s modules. As the server simply provides an interface to the the Solidity contracts on the blockchain, this provided a method of simultaneously testing the contracts’ public functions; a significantly more robust solution compared to attempting to construct a reliable test suite within the Solidity language. Unit tests purposefully excluded modules’ private functions to avoid fixating the test suite on implementation details and to avoid breaking the encapsulation of each module[35]. This allowed the author to focus on ensuring that the API’s components provided expected outputs to given inputs, thus implicitly testing the functionality of private functions which are called in the process.

The Mocha test framework – in combination with the Chai assertion library’s `assert` function – were used to compare a function’s returned result to an expected value:

```
describe("addTask", function () {
    it("adds the given task object to the chain and returns the registered
        task's hex address", function(done) {
        taskManager.addTask(testTask, function (error, address) {
            assert.isNull(error);
            assert.isString(address, "`address` should be the registered task's
                hex address");
            done();
        });
    });
});
```

The snippet above shows the unit test for the `TaskManager` module's public `addTask` function. Mocha's straightforward support for asynchronous JavaScript is nicely exemplified here. The asynchronous function's callback parameters (`error, address`) are checked, after which `done()` is declared, indicating to Mocha that no further values are being awaited for this unit test. API data was provided to the unit tests via mocked objects where required.

5.3 Integration Testing

Integration tests were implemented in a similar manner to the system's unit tests, but were focused on providing a full examination of the API's endpoints the system would provide and thus how differing modules/contracts coordinate with one another. This was achieved by encapsulating the sequence of functions required for each API endpoint in a `chain` module with an associated `chain_test` suite of tests. For example, attempting to retrieve a user's tasks via the `/tasks/:username` endpoint calls the `chain` module's `getUserTasks` function, which in turn simply defines a sequence of `UserManager` and `TaskManager` functions required to fulfill the request, returning the final result. Through this encapsulation, the author was able to define the `chain_test` suite of API tests which examined the API's outputs for given input parameters, ensuring that its constituent parts integrated with one another as expected.

5.4 Functional Testing

As the blockchain and server had been tested rigorously with both unit tests and integration tests, functional testing efforts could be focused on examining the behaviour of the client-side app and ensuring that its interactions with the API were represented correctly within the user interface. Functional tests were therefore executed by running the React Native app in the iOS simulator alongside Google Chrome's developer console, in order to examine log output generated by Redux actions. The `redux-logger` library showed its strength during the functional testing stage, by letting the author easily examine the app's state object before and after any action was triggered, thus significantly simplifying any debugging that was required.

5.5 Coverage

Test coverage reports were generated automatically following each execution of the API's test suite. A coverage summary is shown below, while the full report can be located as browsable collection of HTML files in the repository's `chain/test/coverage` directory. Appendix E contains extracts from the report.

```
  35 passing (46s)


=============================== Coverage summary ===============================
Statements   : 91.28% ( 429/470 ), 7 ignored
Branches     : 58.02% ( 76/131 ), 7 ignored
Functions    : 98.56% ( 137/139 ), 3 ignored
Lines        : 91.86% ( 429/467 )
================================================================================
```

Chapter 6

Conclusions and Project Evaluation

6.1 Project Goals

At the outset of this project, the author intentionally defined the goals at a level which would be challenging, even possibly unfeasible, to reach in the time space available, with no other developers to split workloads with. Considering this context, the project's goals were largely met, forming a healthy basis for future endeavours of this kind to build on.
The first and most challenging goal was to establish a functioning public API which would allow individuals to form teams of their own choosing and register the team's completed tasks on a blockchain. This was achieved in what the author would classify as a "naïve" implementation. Section 6.4 provides details on the author's reasoning for the term "naïve".
The second goal of the project was to build a client-side application to showcase the API's functionality in visual terms. This goal was achieved at a level satisfactory to the author, considering that the primary focus of the project lay in establishing the API and the stack of technologies backing it.

6.2 Personal Aims Achieved

The following section reiterates each personal aim set out by the author, providing commentary on whether the aim has been achieved and in what way.

- *Become familiar with native mobile app development by utilising existing JavaScript knowledge in the context of React Native.*
  Using React Native to build the client-side application was as useful an exercise as the author had hoped it would be. Previous experience with React and JavaScript allowed a full focus on elements specific to a mobile platform, such as self-contained ListViews, which are a common feature in mobile development but absent in web development, where a list is generally an abstract concept constructed from HTML elements.

- *Learn how to write smart contracts in Solidity that are useful in the context of the project and for existing platforms such as Ethereum.*
  Despite initial struggles due to Solidity still being in its early days as a programming language, the project offered ample opportunity to learn how to design and implement various types of smart contracts and how to write idiomatic Solidity in general.

- *Implement a form of blockchain technology, by learning how to set up, run & maintain a a blockchain proficiently.*
  The author feels that he was able to cover all key aspects involved in developing and maintaining a piece of software which leverages a form of blockchain technology.

- *Gain a fundamental understanding of Docker and how it leverages "containerisation" to achieve straightforward deployments of development environments.*
  Working with the `docker-machine` CLI to run and manage virtual operating system environments which could in turn run pre-packaged ("containerised") software taught the author how much the usually arduous and fiddly task of setting up dependencies for a piece of software can be abstracted and simplified, by bundling the software along with its dependencies into a single unit, a container.

6.3 Critical Evaluation

6.3.1 Scope Feasibility

The author feels that the project's scope was challenging but appropriate overall for the timeframe available, considering that the project's goals were intentionally set to a high level. A realisation which began to materialise during the final weeks of the project's implementation stage was that it may have been more pertinent to focus efforts entirely on the API, meaning the design and implementation of the server and blockchain, rather than additionally providing an example client-side implementation. While building a mobile app in React Native was undoubtedly a useful learning experience, creating client-side features for each feature of the API added significant time costs, meaning a number of "Should Have" and "Could Have" requirements which would have enhanced the API's capabilities significantly had to be omitted due to time constraints.

6.3.2 Suitability of Chosen Technologies

Solidity Contracts & Eris Tooling

It took the author a considerable amount of time to become accustomed with how to write idiomatic Solidity contracts, as the programming style required is similar to that of typical object-oriented languages, but deviates in a number of significant ways. Despite this, Solidity's extensive levels of documentation, along with the Eris platform's Solidity tutorials, enabled the author to utilise the language to meet the needs of the project.
The well-defined Eris CLI provided a smooth learning curve regarding how to create, run and deploy Tendermint blockchain instances, allowing the author to avoid significant

amounts of time being wasted by the amount of trial-and-error experimentation that would have been required to run a Tendermint blockchain from scratch.

NodeJS & Express

NodeJS in combination with the Express framework, which added a powerful level of abstraction, were the perfect choice to build the system's API router. The author's level of familiarity with both of these technologies, as well as JavaScript as a whole, allowed the author to immediately mock up some functionality for the server and refactor in confidence until a suitable production-level implementation was established.

React Native, Redux & Flowtype

React Native was as effective as the author had hoped within the scope of the project. Breaking views down into their constituent components not only helped to enforce better design, it also aided in breaking down data requirements within the client-side app, as each component is only responsible for the data it actually requires. Prior knowledge of the React library avoided a lengthy learning phase for the client-side implementation, as the few concepts React Native adds are well documented by its team at Facebook.
Redux was a solid choice for state management, as it ensured a level of control and transparency out-of-the-box which is usually only achievable through careful engineering in other libraries focused on state- and data-management.
Flowtype was a powerful addition to the client-side app, allowing the author to reason about the app's data domain in a more explicit, type-safe manner.

6.4 Future Work

Verifiability

In Section 6.1, the author referred to the system's current implementation as being naïve in its current state. This attribution stems from the fact that the system currently has no means of providing true verifiability for collaborative work in the way, for example, a cryptographic hash may verify an entity or individual. The system is therefore naïve in the sense of being unaware of the possibility of coercive agents interacting with it. QuantiTeam therefore lays the groundwork for a system maintaining verified data on collaborative work in a distributed manner, but has a long way to go to truly achieve this ideal.

Ranking

In order to drive motivation to engage with the system on a continual basis, providing a global ranking system in future iterations as described in the requirements is crucial. This would significantly aid driving a sense of achievement in task participants, as resolving a task would immediately feed back into a reward scheme. Similar to StackOverflow's reputation scheme[1], QuantiTeam scores could become more meaningful the more people use the system, by acting as an abstract representation of participation in one's team to help get collaborative work done.

[1]http://stackoverflow.com/help/whats-reputation

Location Data

From a perspective of extensibility, QuantiTeam could be expanded to include geolocation data related to users and tasks, thus providing a further possibility to corroborate the occurrence of collaborative work between team members at a certain point in time, in a certain place. Yet, this brings with it its own complications regarding data privacy and security, as the ability to identify an individual's location at a given moment in time can easily be utilised in malicious ways.

6.5 Final Thoughts

The significant challenges of this project - both anticipated and unanticipated – were clear to the author from the outset, making the above assessment of QuantiTeam's shortcomings in comparison to its requirements somewhat less painful. While the system this project has birthed still requires significant work to fulfill its theoretical aims, the author feels that new ground has been broken in regards to personal programming skills and knowledge, while also hoping that this exploration of a blockchain's capabilities in a social context will inspire further endeavours along these lines in the future.

Bibliography

[1] Facebook Inc., *React native*, 2016. [Online]. Available: https://facebook.github.io/react-native/ (visited on 09/10/2016).

[2] Ethereum, *Solidity*, 2016. [Online]. Available: http://solidity.readthedocs.io/en/latest/# (visited on 09/10/2016).

[3] Docker Inc., *Docker*, 2016. [Online]. Available: https://www.docker.com/ (visited on 09/10/2016).

[4] I. Jacobson, G. Booch, J. Rumbaugh, J. Rumbaugh, and G. Booch, *The unified software development process*. Addison-wesley Reading, 1999, vol. 1.

[5] DSDM Consortium, *Moscow prioritisation*, 2015. [Online]. Available: https://www.dsdm.org/content/moscow-prioritisation (visited on 09/10/2016).

[6] Atlassian, *Kanban*, 2016. [Online]. Available: https://www.atlassian.com/agile/kanban (visited on 09/11/2016).

[7] A. Usher and N. Kober, "Student motivation an overlooked piece of school reform," *Education Digest*, vol. 78, pp. 9–16, 2013.

[8] J. M. Bridgeland, J. J. DiIulio Jr, and K. B. Morison, "The silent epidemic: Perspectives of high school dropouts.," *Civic Enterprises*, 2006.

[9] J. Berktold, S. Geis, and P. Kaufman, *Subsequent Educational Attainment of High School Dropouts. Postsecondary Education Descriptive Analysis Reports. Statistical Analysis Report.* ERIC, 1998.

[10] M. K. Johnson, R. Crosnoe, and G. H. Elder Jr, "Students' attachment and academic engagement: The role of race and ethnicity," *Sociology of Education*, pp. 318–340, 2001.

[11] Committee on Increasing High School Students' Engagement and Motivation to Learn, *Engaging Schools*. Washington, D.C.: The National Academies Press, 2003, ISBN: 978-0-309-08435-2. DOI: 10.17226/10421. [Online]. Available: http://www.nap.edu/catalog/10421.

[12] M. Temperini and A. Sterbini, "Learning from peers: motivating students through reputation systems," *Proceedings - 2008 International Symposium on Applications and the Internet, SAINT 2008*, pp. 305–308, 2008. DOI: 10.1109/SAINT.2008.107.

[13] A. Sterbini and M. Temperini, "Collaborative projects and self evaluation within a social reputation-based exercise-sharing system," *Proceedings - 2009 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology - Workshops, WI-IAT Workshops 2009*, vol. 3, pp. 243–246, 2009. DOI: 10.1109/WI-IAT.2009.273.

[14] Tangient LLC., *Wikispaces*, 2016. [Online]. Available: https://www.wikispaces.com/content/classroom/about (visited on 09/10/2016).

[15] Haiku Learning, *Haikulearning*, 2016. [Online]. Available: https://www.haikulearning.com/ (visited on 09/10/2016).

[16] A. M. Antonopoulos, *Mastering Bitcoin: Unlocking digital cryptocurrencies.* " O'Reilly Media, Inc.", 2014.

[17] N. Szabo, *The idea of smart contracts*, 1997. [Online]. Available: http://szabo.best.vwh.net/smart_contracts_idea.html (visited on 09/10/2016).

[18] Eris Industries, *Eris: The smart contract application platform*, 2016. [Online]. Available: https://erisindustries.com/ (visited on 09/10/2016).

[19] Tendermint, *Tendermint*, 2016. [Online]. Available: http://tendermint.com/ (visited on 09/10/2016).

[20] D. Abramov, *Redux.js*, 2016. [Online]. Available: http://redux.js.org/ (visited on 09/10/2016).

[21] G. E. Krasner, S. T. Pope, *et al.*, “A description of the model-view-controller user interface paradigm in the smalltalk-80 system,” 1988.

[22] Facebook Inc., *Flow - a static type checker for javascript*, 2016. [Online]. Available: https://www.flowtype.org/ (visited on 09/10/2016).

[23] E. Ries, “Minimum viable product: A guide,” *Startup Lessons Learned*, 2009.

[24] M. Jensen, *Program to an interface, not an implementation*, 2009. [Online]. Available: http://www.fatagnus.com/program-to-an-interface-not-an-implementation/ (visited on 09/10/2016).

[25] Arcitura Education Inc., *What is rest?* 2016. [Online]. Available: http://www.whatisrest.com/ (visited on 09/10/2016).

[26] E. Gamma, *Design patterns: Elements of reusable object-oriented software.* Pearson Education India, 1995.

[27] R. C. Martin, *Agile software development: Principles, patterns, and practices.* Prentice Hall PTR, 2003.

[28] D. Geary, *Introducing redux*, 2016. [Online]. Available: http://www.ibm.com/developerworks/library/wa-manage-state-with-redux-p1-david-geary/ (visited on 09/10/2016).

[29] Google Inc., *Mvc architecture*, 2016. [Online]. Available: https://developer.chrome.com/apps/app_frameworks (visited on 09/10/2016).

[30] A. Osmani, *Learning JavaScript design patterns.* " O'Reilly Media, Inc.", 2012.

[31] C. a. Shaffer and V. Tech, *Data Structures and Algorithm Analysis*, 3.2. 2013, vol. 2, p. 231, ISBN: 0486485811.

[32] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, “When and why your code starts to smell bad,” in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, IEEE Press, 2015, pp. 403–414.

[33] Microsoft Inc., *Primary and foreign key constraints*, 2016. [Online]. Available: https://msdn.microsoft.com/en-GB/library/ms179610.aspx (visited on 09/10/2016).

[34] K. Beck, *Test-driven development: By example.* Addison-Wesley Professional, 2003.

[35] A. Hunt and D. Thomas, *Pragmatic unit test: In java with junit*, 2003.

Appendix A

Design and Implementation Diagrams

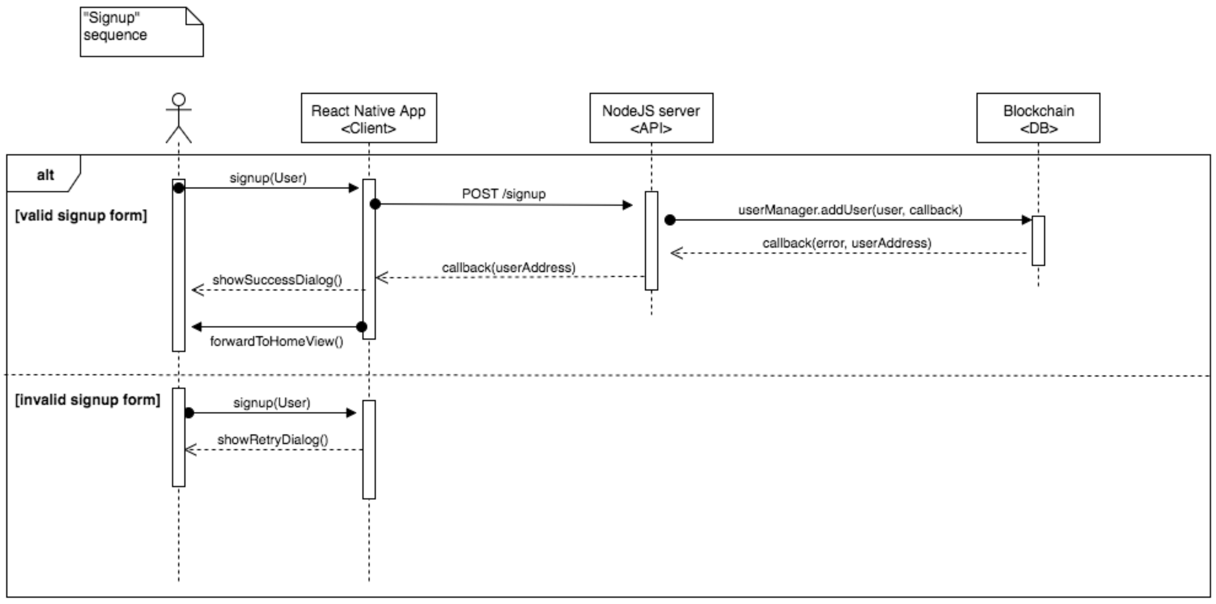


Figure A.1: Sequence of events for a "Signup" action

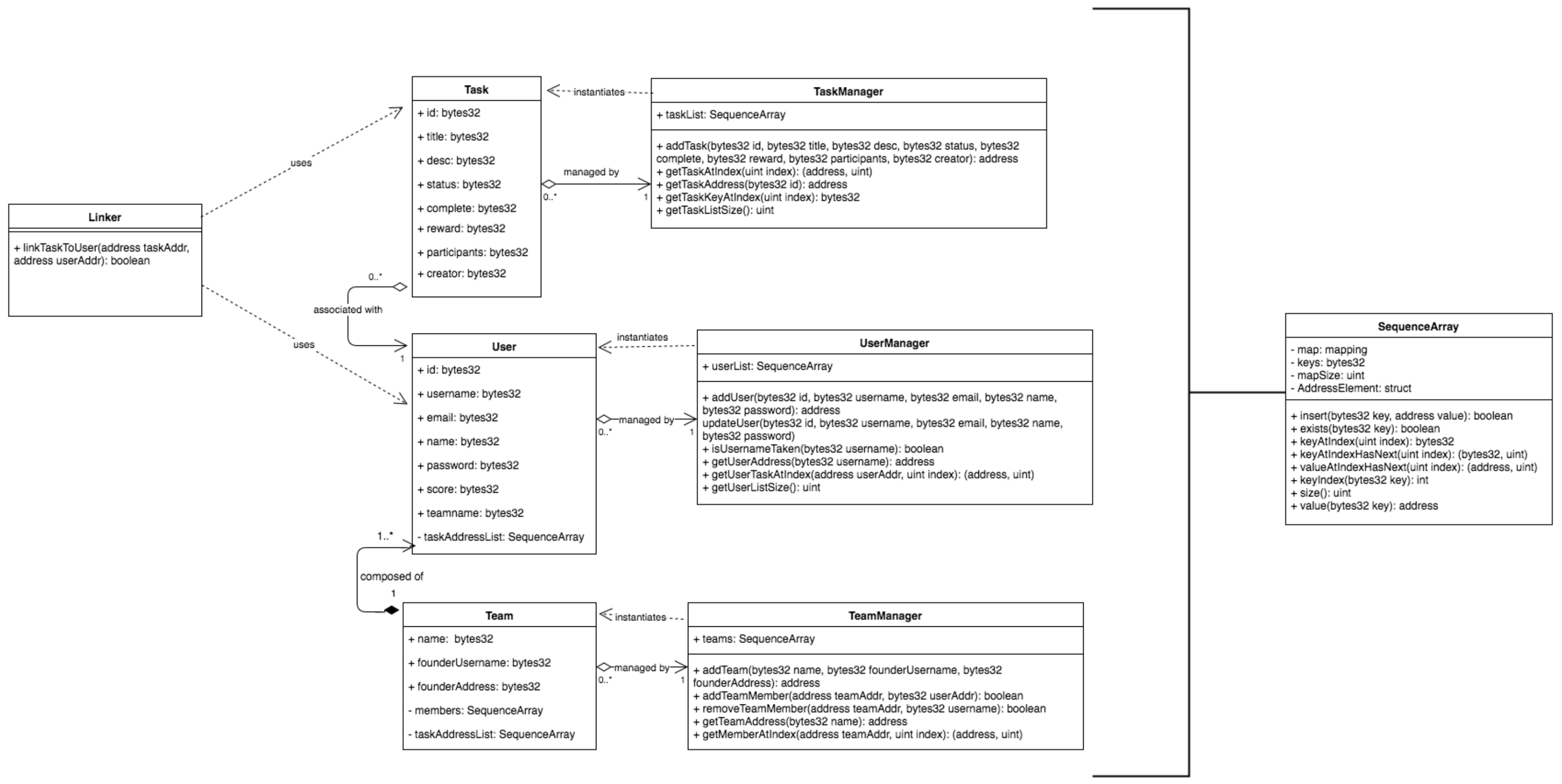

Linker
+ linkTaskToUser(address taskAddr, address userAddr): boolean
uses
uses
Task
+ id: bytes32
+ title: bytes32
+ desc: bytes32
+ status: bytes32
+ complete: bytes32
+ reward: bytes32
+ participants: bytes32
+ creator: bytes32
instantiates
TaskManager
+ taskList: SequenceArray
+ addTask(bytes32 id, bytes32 title, bytes32 desc, bytes32 status, bytes32 complete, bytes32 reward, bytes32 participants, bytes32 creator): address
+ getTaskAtIndex(uint index): (address, uint)
+ getTaskAddress(bytes32 id): address
+ getTaskKeyAtIndex(uint index): bytes32
+ getTaskListSize(): uint
managed by
0..*
1
0..*
associated with
1
User
+ id: bytes32
+ username: bytes32
+ email: bytes32
+ name: bytes32
+ password: bytes32
+ score: bytes32
+ teamname: bytes32
- taskAddressList: SequenceArray
instantiates
UserManager
+ userList: SequenceArray
+ addUser(bytes32 id, bytes32 username, bytes32 email, bytes32 name, bytes32 password): address
updateUser(bytes32 id, bytes32 username, bytes32 email, bytes32 name, bytes32 password)
+ isUsernameTaken(bytes32 username): boolean
+ getUserAddress(bytes32 username): address
+ getUserTaskAtIndex(address userAddr, uint index): (address, uint)
+ getUserListSize(): uint
managed by
0..*
1
1..*
composed of
1
Team
+ name: bytes32
+ founderUsername: bytes32
+ founderAddress: bytes32
- members: SequenceArray
- taskAddressList: SequenceArray
instantiates
TeamManager
+ teams: SequenceArray
+ addTeam(bytes32 name, bytes32 founderUsername, bytes32 founderAddress): address
+ addTeamMember(address teamAddr, bytes32 userAddr): boolean
+ removeTeamMember(address teamAddr, bytes32 username): boolean
+ getTeamAddress(bytes32 name): address
+ getMemberAtIndex(address teamAddr, uint index): (address, uint)
managed by
0..*
1
SequenceArray
- map: mapping
- keys: bytes32
- mapSize: uint
- AddressElement: struct
+ insert(bytes32 key, address value): boolean
+ exists(bytes32 key): boolean
+ keyAtIndex(uint index): bytes32
+ keyAtIndexHasNext(uint index): (bytes32, uint)
+ valueAtIndexHasNext(uint index): (address, uint)
+ keyIndex(bytes32 key): int
+ size(): uint
+ value(bytes32 key): address

Figure A.2: A full Class diagram for the system's smart contracts

Appendix B

Use Cases

Case	Signup
ID	UC1
Description	A user should be able to sign up to the app
Primary Actor	User
Secondary Actor	System
Preconditions	The User is not already logged into the app
Main Flow	1) User downloads and opens the app on their device 2) User taps "Sign up" on the app's login screen 3) User enters their details in the form shown by the Signup view (go to Alternative Flow if there are validation errors) 4) If the user's details are successfully validated, the user is logged into their new account and forwarded to the Home view
Post Conditions	None
Alternative Flow	If validation fails at step 3, the user is notified of the reason for the failed attempt by being shown a popup dialog and asked to try again
Case	Login
ID	UC2
Description	A user must be able to log into the app
Primary Actor	User
Secondary Actor	System
Preconditions	The User is not already logged into the app

Main Flow	1) User opens the app on their device 2) User enters their username and password into the Login view's form (go to Alternative Flow if there are validation errors) 3) If the user's details are successfully validated, the user is logged into their account and forwarded to the Home view
Post Conditions	None
Alternative Flow	If validation fails at step 2, the user is notified of the reason for the failed attempt by being shown a popup dialog and asked to try again
Case	AddTask
ID	UC3
Description	A user must be able to add a task to be tracked by the system
Primary Actor	User
Secondary Actor	System
Preconditions	The user is logged into the app
Main Flow	1) User navigates to the "Tasks" view via the navigation bar 2) User taps "Add Task" action button, is redirected to "Add Task" view 3) User enters the name, description and usernames of any team members involved in the task 4) User taps "Add" and receives confirmation whether the task was successfully added or not
Post Conditions	The specified task is registered on the blockchain
Alternative Flow	None
Case	ViewTasks
ID	UC4

Description	A User should be able to see an overview of the tasks they are currently involved/have previously completed, and inspect each task's details
Primary Actor	User
Secondary Actor	System
Preconditions	User is logged in
Main Flow	1) User navigates to the Tasks view via the navigation bar, the system shows a list of tasks under the headers "To Do" or "Completed" (go to Alternative Flow if no tasks related to this user can be found) 2) User taps an individual task in the list and is forwarded to the TaskDetail view
Post Conditions	None
Alternative Flow	If the User has no tasks associated to them, an empty list is rendered with a note stating that no tasks were found
Case	TaskComment
ID	UC5
Description	A User should be able to comment on a specific task
Primary Actor	User
Secondary Actor	System
Preconditions	At least one task related to this User
Main Flow	1) User navigates to Tasks view via the navigation bar and taps the task in the list they wish to comment on 2) User is shown the TaskDetail view, in which they are able to enter a comment into the provided comment form 3) User taps the submit action button 4) The system displays the newly added comment

Post Conditions	The comment is associated to said task and displayed alongside it
Alternative Flow	If the comment fails to be registered for any reason, the user is informed of this via a popup dialog and asked to try again
Case	OptOutOfTask
ID	UC6
Description	Any team member should be able to opt out of a task they were enrolled in if they are not in fact involved in it
Primary Actor	User
Secondary Actor	System
Preconditions	User has been enrolled into a task by another team member
Main Flow	1) User navigates to Tasks view via the navigation bar and taps the task in the list they wish to opt-out of 2) In the TaskDetail view, the user taps the "Remove me from this task" action button and confirms the wanted action in a popup dialog (go to Alternative Flow if User does not confirm) 3) User receives confirmation that they have been removed from the task and is redirected to the Tasks view
Post Conditions	The user is removed from the task's participants list and it is no longer shown in their task overview
Alternative Flow	If the User does not confirm that they should be removed from the task, they are simply returned to the underlying TaskDetail view
Case	CheckTaskCompletion
ID	UC7

Description	For a given task, a team member should be able to see how many other team members have completed the task if they are registered as participants
Primary Actor	User
Secondary Actor	System
Preconditions	The task involved multiple participants from a team
Main Flow	1) User navigates to Tasks view via the navigation bar and taps the task in the list they wish to inspect 2) User is able to see a completion count or percentage, indicating how many other participants have yet to confirm their completion of the task
Post Conditions	None
Alternative Flow	None

Case	ExternalLinkToTask
ID	UC8
Description	A User should be able to provide public links to their completed tasks and general prhlineofile
Primary Actor	User
Secondary Actor	System
Preconditions	The User has completed tasks associated with their account
Main Flow	1) User navigates to Tasks view via the navigation bar, scrolls to the "Completed" section, and taps the task in the list they wish to inspect 2) In the TaskDetail view, the User taps the "Create link to this task" action button 3) A public link to the task is copied to the device's clipboard
Post Conditions	Device's clipboard contains a link to the given task

Alternative Flow	None
Case	CreateTeam
ID	UC9
Description	A User should be able to start a new team by adding other members via their usernames
Primary Actor	User
Secondary Actor	System
Preconditions	User is not already part of a team
Main Flow	1) User navigates to the Team view via the navigation bar 2) As the User is not part of a team yet, they are shown a "Create Team" button and a hint inviting them to use it 3) User taps "Create Team" and is forwarded to a form to enter team-related details 4) User enters a team name and other usernames to add these users to the team-to-be 5) User taps "Create" action button and is shown a confirmation dialog whether the team creation was successful or not (go to Alternative Flow for usernames that are already part of a team)
Post Conditions	New team with the specified details and members is registered on the blockchain
Alternative Flow	If the team creation fails for any reason, the User is informed of this in the dialog instead of receiving a confirmation. If chosen usernames are already assiocated with another team, the dialog indicates which usernames are affected

Case	CheckTeamScore
ID	UC10
Description	A User should be able to see their team's current global ranking
Primary Actor	User
Secondary Actor	System
Preconditions	User is member of a team
Main Flow	1) User navigates to the Team view via the navigation bar 2) The team's current ranking is displayed in the team's overview section
Post Conditions	None
Alternative Flow	None
Case	RequestHelp
ID	UC11
Description	A User can request help for a task from other task participants
Primary Actor	User
Secondary Actor	System
Preconditions	User is enrolled in a task
Main Flow	1) User navigates to Tasks view via the navigation bar and taps the task in the list they wish to request help for 2) User taps the "Request Help" action button 3) User is shown confirmation of help request being registered and other task participants are notified of the help request
Post Conditions	A "help request" push notification has been sent to all other task participants
Alternative Flow	If the request fails for any reason or there are no other participants within the task, the user is informed of this via a popup dialog

Case	OfferHelp
ID	UC12
Description	A User may offer help to the rest of the participants without a specific request being present
Primary Actor	User
Secondary Actor	System
Preconditions	User is enrolled in a task
Main Flow	1) User navigates to Tasks view via the navigation bar and taps the task in the list they wish to offer help for 2) User taps "Offer Help" action button 3) User is shown confirmation of help offer being registered, other task participants are notified of the help offer
Post Conditions	A "help offer" push notification has been sent to all other task participants
Alternative Flow	If the request fails for any reason or there are no other participants within the task, the user is informed of this via a popup dialog
Case	MuteNotifications
ID	UC13
Description	A User should be able to mute help request and help offer notifications
Primary Actor	User
Secondary Actor	System
Preconditions	None
Main Flow	1) User navigates to the Profile view and taps Settings icon 2) User moves the "Mute Help Notifications" slider to "ON"
Post Conditions	User no longer receives notifications for help requests or help offers

Alternative Flow	None
Case	EditAccount
ID	UC14
Description	A User can edit their account details after signing up
Primary Actor	User
Secondary Actor	System
Preconditions	User has an account on the app
Main Flow	1) User navigates to the Profile view and taps Settings icon 2) User taps "Edit Account Details" action button 3) User is forwarded to the EditAccount view in which the current details are displayed and each field can be edited 4) User taps "Save" action button and receives confirmation that the changes have been applied
Post Conditions	The changes to the User profile are registered in the blockchain
Alternative Flow	If saving the proposed changes fails for any reason, the User is informed via a popup dialog
Case	SetProfilePicture
ID	UC15
Description	A User can edit their profile picture which appears to other Users
Primary Actor	User
Secondary Actor	System
Preconditions	User has an account on the app

Main Flow	1) User navigates to the Profile view and taps Settings icon 2) User taps "Set Profile Picture" action button 3) User is asked to select a picture from the device's photo gallery 4) User taps "Set" and is redirected to the Profile view in order to confirm the change has taken place
Post Conditions	The Users image is locally stored within the app and displayed alongside their profile
Alternative Flow	If setting the profile picture fails for any reason, the User is informed via a popup dialog
Case	DeleteAccount
ID	UC16
Description	A User is able to delete their account
Primary Actor	User
Secondary Actor	System
Preconditions	User has an account on the app
Main Flow	1) User navigates to the Profile view and taps Settings icon 2) User taps "Delete My Account" action button 3) User is asked to confirm the proposed action in a popup dialog window 4) User taps "Confirm", upon which they are logged out and shown the app's login view (go to Alternate Flow for non-confirmation)
Post Conditions	The User no longer appears within the app and their account is removed from the blockchain
Alternative Flow	If the User taps "Abort" in the dialog window, the dialog window disappears and no other action is taken
Case	AttachTaskFile

ID	UC17
Description	User is able to upload attachments related to their tasks
Primary Actor	User
Secondary Actor	System
Preconditions	User has an account on the app
Main Flow	1) User opens a browser and navigates to the web server's address 2) User enters their task token previously provided by the mobile app 3) User selects the file to be attached to the task associated with the task token 4) User taps "Confirm", upon which they are shown a confirmation that the file has been attached (go to Alternate Flow for non-confirmation)
Post Conditions	The chosen file is stored in the blockchain and linked to the task identified via the task token
Alternative Flow	If the user fails to select a file or enters an invalid task token, they are informed of this via a dialog window

Table B.1: Use Cases

Appendix C

System Manual

Installation

Dependencies

The following dependencies are required to run a local development instance of QuantiTeam's Tendermint blockchain:

- Docker CLI - `docker` (& `docker-machine` on OSX).
- Eris CLI - `eris` provides a wrapper and toolchain around the Tendermint blockchain and is used extensively.
- Node.js - v4.x upwards.
- NPM - QuantiTeam relies on NPM scripts to run tests and various other tasks.

To install the project's Node.js dependencies, ensure your present working directory (`pwd`) is `quantiteam/chain` and run:

```
npm install
```

Setting up

First, let's move into QuantiTeam's API directory with the following command:

```
cd chain
```

Creating a local dev `simplechain`:

- Automatically: Run `simplechain.sh` in the repository's root directory, which should start logging the chain's activities after setup.
- Manually: Follow Eris's brief tutorial.

Running a local development chain

Assuming we now have a functioning `simplechain` instance, let's boot it up and configure some local environment variables. In the repo's root directory run the following two shell scripts:

```
. ./chain-up.sh; . ./envsetup.sh
```

We can easily verify whether the `simplechain` instance is running as expected by following its log output:

```
npm run chainlog
```

Running the Node.js chain service

Now that we have a local development chain running which we can interact with, let's spin up the chain service which will act as our API router to interface with the local chain itself.

Deploying contracts

First, let's compile and deploy our Solidity smart contracts in the `contracts` directory:

```
# `$addr` should be defined from previously running `envsetup.sh`
npm run compile -- $addr
```

Anytime a change is made to the smart contracts, `compile` should be run to deploy these changes to the running `simplechain` instance.

Running tests

Next, we'll want to run QuantiTeam's test suite to ensure everything is working as expected:

```
npm test
```

This should also provide a coverage report once all unit tests have run.

Booting the service

The chain service itself can be built & booted simply with:

```
npm run build
```

Anytime a change is made to the node server, `build` should be run as it builds the new service container via `docker` and replaces it with the previous one via `eris`.
Please refer to `package.json` for more detailed insights into which shell commands each NPM script executes.

API

QuantiTeam's API exposes the following HTTP endpoints:

- POST `/upload` - Upload a task related file via `multipart/form-data`.

- POST `/user/taken` - Check whether the the username in `req.body.username` is already taken.
- POST `/user/signup` - Sign up a new user with the form data passed in `req.body`.
- POST `/user/login` - Log in an existing user with the form data passed in `req.body`.
- GET `/user/profile/:username` - Get the profile of the username passed via `req.params.username`.
- GET `/tasks/:username` - Get the tasks of the username passed via `req.params.username`.
- POST `/task` - Add a new task to the blockchain via the form data passed in `req.body.task` for the username in `req.body.username`.
- GET `/task/completed/:token` - Mark the task associated with the token passed in `req.params.token` as completed.
- GET `/team/taken/:teamname` - Check whether the teamname passed as `req.params.teamname` is already taken.
- POST `/team` - Add a new team to the blockchain via the form data passed in `req.body.form`.
- GET `/team/:teamname` - Get the team profile for the teamname passed as `req.params.teamname`.
- POST `/team/add-member` - Add a new member to a team with the form data passed in `req.body.form`.

Shutting down

To shut down the local chain and the `docker-machine` instance, simply run:

```
. ./chain-down.sh
```

Appendix D

User Manual

Introduction

This user manual provides instructions on how to use QuantiTeam's mobile app, which serves as a basic showcase of the system's current features and its API in the context of a real application, as well as the web uploader which can be used to attach text files to tasks registered in the mobile app.

Registration

Login

Upon opening the app, the user is greeted with QuantiTeam's login view. If the user has previously created an account in the QuantiTeam blockchain, they can simply enter their username and password, followed by a tap on "Login" to be forwarded to their task overview. If the login fails for any reason, the user is informed via dialog window.

Signup

If this is the first time using QuantiTeam, the user can tap on "Sign up here" under the login form. This forwards the user to a signup form, requiring details such as a name, email, username, and password. Once the user is happy with their entries, tapping “Sign up” will validate the entered information. If mandatory fields have been left blank, the app will notify the user by marking them red. If the entered username is already taken or the entered passwords don’t match, the user will be informed via a dialog window.

Team

Creating a Team

Using the navigation bar at the bottom of the app, the user can navigate to the “Team” tab. If the user is currently not part of a team, the main view will contain instructions on how to create a team. By tapping “Create Team” in the app’s header, the user is forwarded to a simple text entry where they can enter their chosen team name and tap “Create Team” again.
After a short delay to a account for the event being registered in a transaction block within the blockchain, the user should receive confirmation whether creating the team succeeded.

When navigating back to the team overview tab, the user should now see their team’s details, as well as a list of team members. Instead of “Create Team”, the header now contains an “Add Member” action button.

Adding Members

To add a member to their team, a user can tap “Add Member” in the header of the app’s “Team” tab. This forwards the user to a text entry where they are able to enter the username of another QuantiTeam user. After entering a username, the user taps the “Add

Member” button and, after a short delay to register the transaction in the blockchain, receives confirmation whether adding the user to their team was successful or not.

When navigating back to the team overview tab, the user should now see the new member in the list of team members.

Tasks

Using the navigation bar at the bottom of the app, the user can navigate to the “Tasks” tab. By pulling downwards on the screen, the user can refresh the list of existing tasks if any are currently registered to their username in the blockchain.

Adding a Task

To add a task, the user can tap “Add Task” in the app’s header, which forwards the user to a task form. Here the user is able to enter a title and short description of the task they want to register in the QuantiTeam blockchain, as well as set the status of the task as “To Do” or “Completed”. After filling in these details, tapping the “Add Task” button attempts to register the new task in the blockchain.

Upon returning to the “Tasks” tab via the back-arrow icon in the app’s header, the user can refresh the task list, which should now display the newly registered task.

Inspecting a Task

When tapping a task in the task list, the user is forwarded to detailed overview of the task. Here the user can inspect the task’s details.

Marking a Task as Complete

Once a task has been completed, the user is able to mark it as complete in the blockchain. This can be done by tapping a specific task in the task list, followed by a tap on the “Mark as complete” button in the detailed task view.

Profile

Inspecting the User Profile

Using the navigation bar at the bottom of the app, the user can navigate to the “Me” tab. Here the user is able to see their details, such as their name, username and email address.

Logging out

To log out of the mobile app, the user can tap on “Log out” in the app’s header within the “Me” tab. This resets the app’s state and returns the user to the “Login” view.

Attachments

A user is able to attach text files related to a task via the web uploader.

Prerequisites

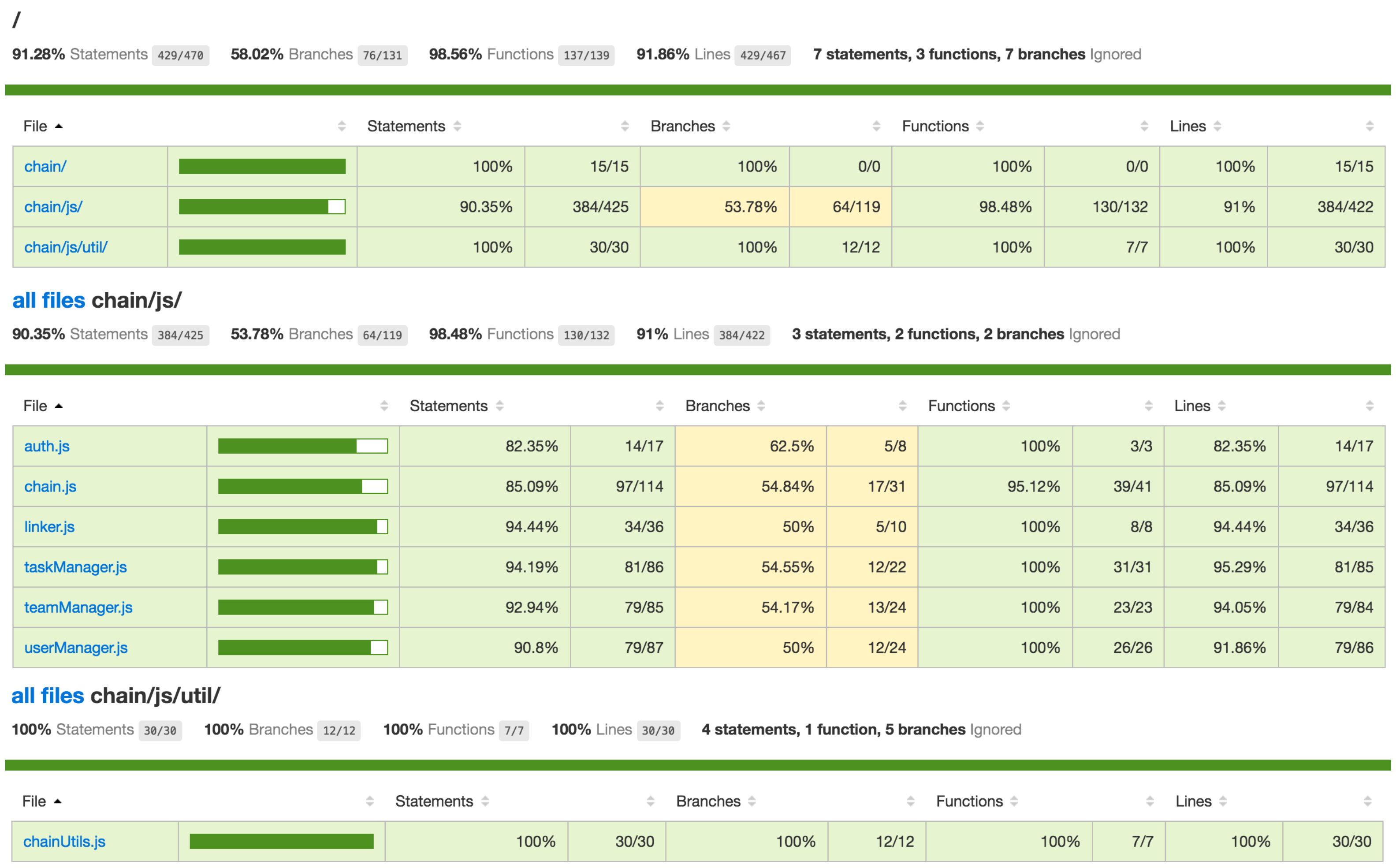
To add a text file attachment to the blockchain, the user requires a task token, which can be copied from a task's "Token" field in the task detail view within the mobile app.

Attaching Text Files to a Task

When opening a browser window and navigating to the QuantiTeam web server, the user is shown a file attachment form. The user can enter the previously copied task token and select a text (`.txt`) file to attach to the task associated with the token. After tapping submit, the user receives confirmation whether attaching the file was successful. Multiple files can be attached to a single task.

Appendix E

Test Coverage Overview



/

91.28% Statements 429/470 **58.02%** Branches 76/131 **98.56%** Functions 137/139 **91.86%** Lines 429/467 **7 statements, 3 functions, 7 branches** Ignored

File ▲		Statements		Branches		Functions		Lines	
chain/		100%	15/15	100%	0/0	100%	0/0	100%	15/15
chain/js/		90.35%	384/425	53.78%	64/119	98.48%	130/132	91%	384/422
chain/js/util/		100%	30/30	100%	12/12	100%	7/7	100%	30/30

all files **chain/js/**

90.35% Statements 384/425 **53.78%** Branches 64/119 **98.48%** Functions 130/132 **91%** Lines 384/422 **3 statements, 2 functions, 2 branches** Ignored

File ▲		Statements		Branches		Functions		Lines	
auth.js		82.35%	14/17	62.5%	5/8	100%	3/3	82.35%	14/17
chain.js		85.09%	97/114	54.84%	17/31	95.12%	39/41	85.09%	97/114
linker.js		94.44%	34/36	50%	5/10	100%	8/8	94.44%	34/36
taskManager.js		94.19%	81/86	54.55%	12/22	100%	31/31	95.29%	81/85
teamManager.js		92.94%	79/85	54.17%	13/24	100%	23/23	94.05%	79/84
userManager.js		90.8%	79/87	50%	12/24	100%	26/26	91.86%	79/86

all files **chain/js/util/**

100% Statements 30/30 **100%** Branches 12/12 **100%** Functions 7/7 **100%** Lines 30/30 **4 statements, 1 function, 5 branches** Ignored

File ▲		Statements		Branches		Functions		Lines	
chainUtils.js		100%	30/30	100%	12/12	100%	7/7	100%	30/30

Figure E.1: Coverage report extracts

Appendix F

Code Extracts

The following section contains extracts from the project's code base which were explicitly mentioned in the main text.

envsetup.sh

```
#!/usr/bin/env bash

# This script should be run to configure `docker-machine`'s environment within
   the
# terminal session, as well as to set up local environment variables which
# simplify the usage of `eris` drastically, such as the chain directory
   ($chain_dir)
# and the address of the contract owner ($addr) if a contract is to be
   deployed onto the chain.

echo "Running QuantiTeam environment setup..."

# Set references to the chain & account
chain_dir=$HOME/.eris/chains/simplechain
chain_dir_this=$chain_dir/simplechain_full_000
echo "chain_dir: ${chain_dir}"
echo "chain_dir_this: ${chain_dir_this}"
```

```
# Isolate the account address into a variable
addr=$(cat $chain_dir/addresses.csv|grep simplechain_full_000|cut -d ',' -f 1)
echo "addr: ${addr}"

# Set up local variables for `docker-machine`
eval $(docker-machine env)
# Set the $host variable to the IP of the running docker-machine container
host=$(docker-machine ip)
echo "'host' set to docker-machine IP: ${host}"

# Get the IP address for the local compiler
compiler_addr=$(eris services inspect compilers NetworkSettings.IPAddress)
echo "compiler_addr: ${compiler_addr}"

# Set the port for the node app to listen to for requests by querying the eris
   service for the port
export IDI_PORT=$( eris services ports idi|cut -d ":" -f 2 )
echo "IDI_PORT set to ${IDI_PORT}"
```

Task.sol

```
import "SequenceArray.sol";

contract Task {

   bytes32 public id; // immutable
   bytes32 public title; // mutable
```

```
bytes32 public desc; // mutable
bytes32 public status; // mutable
bytes32 public complete; // mutable
bytes32 public reward; // immutable
bytes32 public participants; // mutable
bytes32 public creator; // immutable
bytes32 public createdAt; // immutable
bytes32 public token; // immutable

SequenceArray attachments = new SequenceArray();

// Constructor
function Task(
    bytes32 _id,
    bytes32 _title,
    bytes32 _desc,
    bytes32 _status,
    bytes32 _complete,
    bytes32 _reward,
    bytes32 _participants,
    bytes32 _creator,
    bytes32 _createdAt,
    bytes32 _token) {
    id = _id;
    title = _title;
    desc = _desc;
    status = _status;
    complete = _complete;
    reward = _reward;
```

```
        participants = _participants;
        creator = _creator;
        createdAt = _createdAt;
        token = _token;
    }

    function associateWithFile(bytes32 fileHash) returns (bool isOverwrite) {
        isOverwrite = attachments.insert(fileHash, this);
        return isOverwrite;
    }

    function markComplete(bytes32 _status) returns (bool success) {
        status = _status;
        success = true;
        return success;
    }
}
```

chainUtils.js

```
/*
 * Some handy utility functions to handle data coming off the blockchain.
 */

var util = require('util');
var EventEmitter = require('events');
var eris = require(__libs+'/eris/eris-wrapper');
```

```
/* istanbul ignore next */
var chainUtils = {


    /**
     * createContractEventHandler - Creates an event handler
     * for `contract` to log any `ActionEvent` events triggered
     * within the contract.
     *
     * @param {Object} contract description
     * @param {Object} log     description
     * @return {null}        description
     */
    createContractEventHandler: function (contract, log) {
        // Set up event emitter
        function ChainEventEmitter () {
            EventEmitter.call(this);
        }
        util.inherits(ChainEventEmitter, EventEmitter);
        var chainEvents = new ChainEventEmitter();

        contract.ActionEvent(
            function (error, eventSub) {
                if (error)
                    throw error;
            },
            function (error, event) {
                if (event) {
```

```
                var eventString = eris.hex2str(event.args.actionType);

                log.info("***CONTRACT EVENT:***\n", eventString);
                chainEvents.emit(eventString, event.args);
            }
        });
},

/**
 * extractInt - Extracts an integer from a `uint`/`int` Solidity type
     value.
 *
 * @param {Object} bcObject The blockchain object to be extracted from.
 * @return {int}           The extracted integer.
 */
extractInt: function (bcObject) {
    return bcObject['c'][0];
},

/**
 * extractIntFromArray - Extracts an integer from a `uint`/`int` Solidity
 * type value nested inside an array.
 *
 * @param {Object} bcObject The blockchain object to be extracted from.
 * @param {int} index   Index position of the int in the array.
 * @return {int}        The extracted integer.
 */
extractIntFromArray: function (bcObject, index) {
    return bcObject[index]['c'][0];
```

```
},

/**
 * marshalForChain - Takes an object, iterates its own properties and
 * encodes them as hexadecimal strings so they can be fed into Solidity
 * contracts easily.
 *
 * @param {Object} obj The object to be marshalled.
 * @return {Object} hexObj `obj` with hexadecimal-encoded properties
 */
marshalForChain: function (obj) {
    var hexObj = {};

    for (var prop in obj) {
        if ({}.hasOwnProperty.call(obj, prop)) {
            var val = obj[prop];

            if (Number.isInteger(val)) {
                val = String(val);
            } else if (Array.isArray(val)) {
                val = JSON.stringify(val);
            } else if (typeof val !== "string") {
                throw new Error("Error at marshalForChain: " + prop + ":" +
                    val + " is not a string.");
            }
            hexObj[prop] = eris.str2hex(val);
        }
    }
```

```
        return hexObj;
    }
};

module.exports = chainUtils;
```