Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

Thomas Smith, tcs1g20
March 23, 2023

Using Blockchain for Video Game Distribution

Project Supervisor: Leonardo Aniello
Second Examiner: Heather Packer

A project report submitted for the award of
BSc Computer Science

Abstract

Video game developers will often have to rely on third party platforms for the distribution of their games; this comes at a large monetary cost to the developer and leaves users at a greater risk of censorship and with weak digital ownership that is reliant on the platform staying active. This project uses the Ethereum blockchain to facilitate the large-scale distribution and continuous updating of video games that allows developers to directly interact with their users, who will now have true digital ownership.

Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

You must change the statements in the boxes if you do not agree with them.

We expect you to acknowledge all sources of information (e.g. ideas, algorithms, data) using citations. You must also put quotation marks around any sections of text that you have copied without paraphrasing. If any figures or tables have been taken or modified from another source, you must explain this in the caption and cite the original source.

I have acknowledged all sources, and identified any content taken from elsewhere.

If you have used any code (e.g. open-source code), reference designs, or similar resources that have been produced by anyone else, you must list them in the box below. In the report, you must explain what was used and how it relates to the work you have done.

I have not used any resources produced by anyone else.

You can consult with module teaching staff/demonstrators, but you should not show anyone else your work (this includes uploading your work to publicly-accessible repositories e.g. Github, unless expressly permitted by the module leader), or help them to do theirs. For individual assignments, we expect you to work on your own. For group assignments, we expect that you work only with your allocated group. You must get permission in writing from the module teaching staff before you seek outside assistance, e.g. a proofreading service, and declare it here.

I did all the work myself, or with my allocated group, and have not helped anyone else.

We expect that you have not fabricated, modified or distorted any data, evidence, references, experimental results, or other material used or presented in the report. You must clearly describe your experiments and how the results were obtained, and include all data, source code and/or designs (either in the report, or submitted as a separate file) so that your results could be reproduced.

The material in the report is genuine, and I have included all my data/-code/designs.

We expect that you have not previously submitted any part of this work for another assessment. You must get permission in writing from the module teaching staff before re-using any of your previously submitted work for this assessment.

I have not submitted any part of this work for another assessment.

If your work involved research/studies (including surveys) on human participants, their cells or data, or on animals, you must have been granted ethical approval before the work was carried out, and any experiments must have followed these requirements. You must give details of this in the report, and list the ethical approval reference number(s) in the box below.

My work did not involve human participants, their cells or data, or animals.

Acknowledgements

I would like to thank my supervisor, Leonardo Aniello, for his support throughout this project.

Contents

Chapter 1

Problem Statement

1.1 The Problem

Video games are often large and highly popular pieces of software that are typically distributed for developers by a third party platform like Steam or Epic Games. Whilst these platforms provide benefits such as availability, and some social features they have some major downsides that include:

(a) taking a large cut of all revenue,
Steam take a 30% cut [10, 2]
(b) being vulnerable to censorship from governments,
*The Chinese version of Steam is heavily censored [**?**]*
(c) the user's access to their games is linked to the platform.
If the platform shuts down, the user loses all their games

1.2 Goals

The goal of this project is to implement a large-scale distribution platform that will allow game developers to release and continuously update their games on a public network by directly interacting with their users. This is in the aim to provide greater profits to developer's, freedom from censorship, and better digital ownership for the user.

1.3 Scope

This project will be broken down into two distinct components:

1. **On-Chain** This component will consist of a set of Solidity Smart Contracts written for the Ethereum blockchain that will allow users to view metadata about and purchase games. It will be tested using a local test-net like Ganache using TypeScript. It will later be deployed to the Ethereum test-net to showcase the application in a live network.
2. **Off-Chain** This component will be what users will actually run. Each user will join a peer-to-peer network in which they can upload and download games off of other users. This will interface with the blockchain to allow users access to game metadata. See Section **??** for details about how this will be tested.

For both of these, a series of acceptance tests, that directly correlate to individual requirements, will be run and include a series of integration tests to show that my ap-

plication can meet the requirements and goals I set out. A more detailed description is given in Section **??**.

Chapter 2

Background Research

2.1 BitTorrent

It is an unrealistic to expect that every game uploaded to the network will be downloaded by every user so only a subset of users will have the game installed and available to share. In this section and Section 3.2, I will look at how various peer-to-peer file-sharing networks allow users to discover and download content that is fragmented across the network.

BitTorrent [6, 13] was chosen as part of my background research as it is one of the most popular P2P file-sharing platforms. In 2013 it was estimated that tens of millions of users used BitTorrent every day [16]. In BitTorrent, users barter for chunks of data by downloading and uploading them in a tit-for-tat fashion, such that peers with a high upload rate will typically also have a high download rate.

Download Protocol

For a user to download data from BitTorrent they would:

1. Find the corresponding .torrent file that contains metadata about the torrent.
2. The user will find peers, using a tracker identified in the .torrent, that are also interested in that content and will establish connections with them.
3. The user will download blocks[1], from peers, based upon the following priority:
   (a) **Strict Priority** Data is split into pieces and sub-pieces with the aim that once a given sub-piece is requested then all of the other sub-pieces in the same piece are requested.
   (b) **Rarest First** Aims to download the piece that the fewest peers have to increase supply.
   (c) **Random First Piece** When a peer has no pieces, it will try to get one as soon as possible to be able to contribute.
4. The node will continuously upload blocks it has while active.

Availability

One of the most significant issues facing BitTorrent is the availability of torrents, where *'38% of torrents become unavailable in the first month'* [6] and that *'the majority of users disconnect from the network within a few hours after the download has finished'* [13]. This

[1] nodes may reject downloads without the user providing data themselves in a tit-for-tat fashion

paper [12] looks at how the use of multiple trackers for the same content and DHTs can be used to boost availability.

2.2 Ethereum

Ethereum is a Turing-complete, distributed, transaction-based blockchain that allows the deployment of decentralized applications through the use of smart contracts. Ether is the currency used on Ethereum and can be traded between accounts and is used to execute smart contract code on the network.

Smart Contracts

A smart contract is an executable piece of code, written in Solidity, that will automatically execute on every node in the Ethereum network when certain conditions are met. Smart contracts are enforced by the blockchain network and remove the need for intermediaries and reduce the potential of contractual disputes.

Gas is used to measure the computational effort of running a smart contract and must be paid, in Ether, before being processed and added to the blockchain. This helps prevent DoS attacks and provides economic incentives for users to behave in a way that benefits the whole network.

Example Use Cases

Some examples of applications that can be deployed to the Ethereum network are:

- Financial applications, such as decentralised exchanges and payment systems,
- supply chain management and tracking,
- voting and governance systems,
- unique digital asset systems, and
- data storage and sharing platforms.

Chapter 3

Literature Review

3.1 Blockchain-Based Cloud Storage

Blockchain technology can be leveraged for distributed cloud storage to provide both public and private storage. In table 3.1, I detail some examples of how blockchain has been used to create cloud storage platforms:

One gap found when researching these solutions was that few offered file versioning that would allow a user to view previous versions of uploaded data. File versioning is a particularly important to this project as users will likely all have varying versions of the same software.

Paper	**Description of Solution**
Blockchain Based Data Integrity Verification in P2P Cloud Storage [18]	This paper uses Merkle trees to help verify the integrity of data within a P2P blockchain cloud storage network. It also looks at how different structures of Merkle trees effect the performance of the system.
Deduplication with Blockchain for Secure Cloud Storage [8]	This paper describes a deduplication scheme that uses the blockchain to record storage information and distribute files to multiple servers. This is implemented as a set of smart contracts.
Block-secure: Blockchain based scheme for secure P2P cloud storage [7]	A distributed cloud system in which users divide their own data into encrypted chunks and upload those chunks randomly into the blockchain, P2P network.
Blockchain-Based Medical Records Secure Storage and Medical Service Framework [3]	Describes a secure and immutable storage scheme to manage personal medical records as well as a service framework to allow for the sharing of these records.
A Blockchain-Based Framework for Data Sharing With Fine-Grained Access Control in Decentralized Storage Systems [17]	This solution uses IPFS, Ethereum and ABE technology to provide distributed cloud storage with an access rights management system using secret keys distributed by the data owner.

Blockchain based Proxy Re-Encryption Scheme for Secure IoT Data Sharing [9]	An IoT distributed cloud system for encrypted IoT data that uses a proxy re-encryption scheme that allows the data to only be visible to the owner and any persons present in the smart contract.

Table 3.1: *Examples of blockchain cloud storage systems [15]*

3.2 P2P File Sharing

It is unreasonable to expect every node to have a copy of each game uploaded to the blockchain so data will be fragmented across the network. This project will use ideas from various P2P file-sharing networks to help connect nodes interested in the same content Table 3.2 shows some example p2p file-sharing networks.

The main issues involving these networks are:

1. **Trust** Nodes are typically anonymous and you can never fully trust that what you're downloading isn't malicious, and
2. **Payment** These platform don't allow users to pay for content and are generally large sources of piracy.

System	Description of Solution
IPFS [1]	IPFS is a set of protocols for transferring and organising data over a content-addressable, peer-to-peer network. Data uploaded to an IPFS network is addressed using it content identfier CID, which is a cryptographic hash based upon its content. IPFS is open source and has many different implementations, such as Estuary or Kubo.
BitTorrent [13]	BitTorrent is a p2p file-sharing system that has user bartering for chunks of data in a tit-for-tat fashion, which provides incentive for users to contribute to the network. More on BitTorrent can be found in Section 2.1.
Swarm [4]	Swarm is a distributed storage solution linked with Ethereum that has many similarities with IPFS [13]. It uses an incentive mechanism, Swap (Swarm Accounting Protocol), that keeps track of data sent and received by each node in the network and then the payment owed for their contribution.
AFS [11, 5]	The Andrew File System was a prototype distributed system by IBM and Carnegie-Mellon University in the 1980s that allowed users to access their files from any computer in the network.
Napster [14]	Napster uses a cluster of centralized servers to maintain an index of every file currently available and which peers have access to it. A node will maintain a connection to this central server and will query it to find files; the server responds with a list of peers and their bandwidth and the node will form a connection with one or many of them and download the data.

Gnutella [14]	Gnutella nodes form an overlay network by sending *ping-pong* messages. When a node sends a *ping* message to their peers, each of them replies with a *pong* message and the *ping* is forwarded to their peers. To download a file, a node will flood a message to its neighbors, who will check if they have and return a message saying so; regardless, the node will continue to flood their request till they find a suitable node to download off of.

Table 3.2: *Various global distributed file systems.*

Chapter 4

Design

4.1 Stakeholders & Requirements

Stakeholders

Game Developers *primary*

This group will use the application to release their games and its updates to their users, who they will reward for helping to distribute it.

Players *primary*

This group will use this application to downloaded and update their games off of. They may also contribute to the distribution of the games to other players for an incentive provided by the developers.

Other Platforms *secondary*

This group consists of platforms like Steam or Epic Games, which serve as the main competitor to this application. It is likely that as more developers choose this application, this group will see a loss in revenue.

Requirements

Tables 4.1 and 4.2 show the functional and non-functional requirements of this project organized using MoSCoW prioritisation.

Functional Requirements

ID	Description
	Must
F-M1	Store game metadata on the Ethereum blockchain
F-M2	A node must download data as constant-sized shards from its peers
F-M3	A node must be able to discover peers who have their desired game installed
F-M4	Games must be updatable through the blockchain
F-M5	A node must be able to upload game data to other nodes in the network

F-M6	A node must be able to download games in their entirety from nodes in the network
F-M7	A node must be able to verify the integrity of each block it downloads
F-M8	The application should run on the Ethereum network
F-M9	Users must be able to purchase games from developers over the network
F-M10	Users must be able to prove they have purchased a game
	Should
F-S1	Seeders should have a way to prove how much data they have seeded
F-S2	Seeders will only upload content to users who have a valid proof of purchase
F-S3	Allow for the distribution of Downloadable Content (DLC) for games
	Could
F-C1	Allow users to request specific game versions
F-C2	Provide a simple GUI for interacting with the blockchain

Table 4.1: *These requirements define the functions of the application in terms of a behavioural specification*

Non-Functional Requirements

ID	Description
	Must
NF-M1	The application is decentralized and cannot be controlled by any one party
NF-M2	Any user must be able to join and contribute to the network
NF-M3	Game uploaders should be publicly identifiable
NF-M4	Metadata required to download the game should be immutable
	Should
NF-S1	This application must be scalable, such that many users can upload and download the same game at the same time.
NF-S2	Only the original uploader can upload an update to their game
NF-S3	Only the original uploader can upload a DLC for their game
	Could
NF-C1	The application could have an intuitive GUI

Table 4.2: *Requirements that specify the criteria used to judge the operation of this application*

4.2 Design Considerations

Data

The first consideration is what kind of data we are going to be storing and where is it going to be stored.

Data	Size	Location	Explanation
Game Metadata	100 – 200B	Ethereum	This data represents information about the game that help identify it, such as its title, developer, version, root hash, etc. This information should allow for the unique identification of every game uploaded, whilst remaining minimal. Due to the minimal amount of storage required and the fact that every user should be able to discover every game, this data is best stored on the blockchain.
Game Hash Data	S_g/B [1]	IPFS	This data will be the hash tree of a given game and will be used by a player to verify the contents of each block of data they download as well as the expected structure of the applications contents. Due to this data's moderate size and the fact that not every user will need to view every game's hash tree, there is no need to upload this to the blockchain as this will add an unnecessary expense in publishing games. IPFS is ideal for this as we do not need to restrict access to the data but still need to easily share it and we can simply include the IPFS ID within our game metadata stored on the blockchain. Swarm [4] was also considered but wasn't chosen as it would further couple the project with Ethereum and isn't as mature or widely used as IPFS.
Game Data	*avg.* $44GB$ [2]	Peers	This will be the actual data for the game that is used to run it. To play a game, a user will need access to all of its data. More information about this is in Section 4.2.2. An important property for uploaded games is that all users do not have access to all games without having payed for them first so we shouldn't store them on public platforms like the blockchain or IPFS. Section 4.2.2 will discuss how this can be achieved using a distributed peer-to-peer file sharing model.

[1] where S_g is the size of the game and B is the shard size

[2] Calculated based off of the top 30 games from https://steamdb.info/charts/ on 22/03/2022

4.2.1 Blockchain

Type of Blockchain

To satisfy (**NF-M1**) and (**NF-M2**), we will need to use a public blockchain, which will benefit our project by:

- being accessible to a larger user-base, which should boost availability and scalability (**NF-S1**),
- reducing the risk of censorship (**NF-M1**), and
- providing greater data integrity (**NF-M4**)

Ethereum is a public blockchain that allows developers to publish their own distributed applications to it. It comes with an extensive development toolchain so is an obvious choice for this project (**F-M8**).

Data to Store

To satisfy (**F-M1**), the data stored on the blockchain will be used for the identification games and will consist of the following fields, where *italic* fields will be automatically-generated for the user:

Name	Description
	For each game
title	The name of the game.
version	A version number of the game.
release date	When the game was released.
developer	The name of the developer releasing the game.
previousVersion	The root hash of the most previous version of the game if it exists.
price	The price of the game in Wei
uploader	The Ethereum address of the developer.
root hash	The root hash of the game that uniquely identifies it and is based upon its contents.
IPFS address	The ID of the hash tree of IPFS that can be used to download the hash tree before starting a download of a game.
	Other
library	A mapping for storing all games uploaded to the network, where a game's root hash is the key used to find its information.
gameHashes	Solidity doesn't allow us to enumerate maps so we will also store a list of hashes for all games uploaded.
purchased	A mapping which allows us to easily check if a user has purchased a game.

Table 4.4: *All the data to be stored on the Ethereum blockchain*

Purchasing Content

Users will purchase content from developers over Ethereum using Ether (**F-M9**) and this will be recorded on the blockchain (**F-M10**). Any user can see which other users have

purchased the game users can prove this between each other using their public/private keys.

4.2.2 Distributed File Sharing

Hash Tree

The hash tree of a given directory is a tree object that stores information about its structure and contents. This is used to tell users what information they need to download, where it goes and what its contents should be. For every file, the hash tree stores a series of SHA-256 hashes.

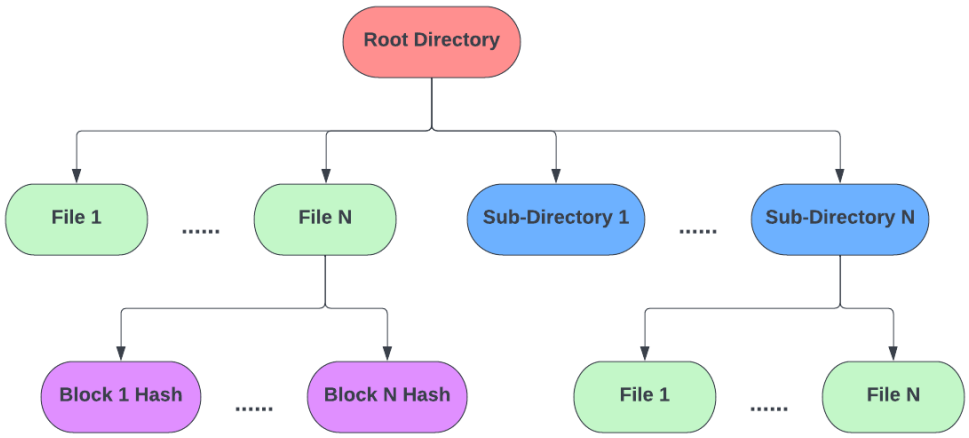

Figure 4.1: *The structure of a hash tree*

Uploading Content

For a developer to upload their game (**F-M5**) they must provide the required metadata outlined in Section 4.2.1 as well as the location of the game in storage. A hash tree is then produced of the game and this is uploaded to IPFS, and the metadata about the game is uploaded to Ethereum.

Downloading Content

Like mentioned in Section 4.2, it is impractical to store the game's data on the blockchain or even IPFS. Instead we will consider ideas from decentralised file-sharing networks, like BitTorrent, to facilitate the distribution of content.

Games are content addressable and are identified by their root hashes, which are stored on the blockchain and are calculated from the hash tree. Users will send messages using this hash to identify other users who are also interested in the same content (**F-M3**), so they can share data. When two nodes connect to share content the node seeking content will:

1. send their ethereum address along with an encrypted message to prove their address. The address is then looked up on the purchase list stored in the game's entry on the blockchain (**F-S2**).
2. Request individual shards from the node using the shard's hash (**F-M2**).

3. Use the hash tree, which has been fetched from IFPS, to verify the shard's contents (**F-M7**).
4. Send an encrypted encrypted certificate that the sender can use to prove their contribution.
5. Repeat this for an arbitrary number of shards.

Updating Content

To satisfy (**F-M4**), developers will perform the same steps outlined in Section 4.2.2 but must also provide the root hash of the most previous version of the game. Any users who have purchased the previous version, will be added to the list of users who have purchased the new version. Additionally, this will include the restriction that only the original uploader can upload an update for their game (**NF-S2**).

Each version is considered as its own game and will require users to download the updated version separately. Whilst this isn't reflective of how updates are typically managed, this will be acceptable for the scope of this project and any changes will be considered as a future extension to this project.

Downloadable Content

UNFINISHED

Proving Contribution

When a user successfully downloads a shard of data, they will reply with a confirmation certificate to prove that they have downloaded the game. Each Ethereum address is related to a public/private key pair so a confirmation certificate will be encrypted by a given node's private key to prove authenticity. A user will prove their contribution (**F-S1**) by sending a collection of certificates to the uploader, who will validate them and reward the user accordingly.

4.3 Limitations

This project will not attempt to mimic any of the social features (friends, achievements, message boards, etc.) provided by platforms like Steam.

Section 2.1 highlights the issue of availability in p2p sharing networks and this platform will likely suffer from similar issues. These can be mitigated by having an active community or good incentives to help distribute the game.

Chapter 5

Project Management

5.1 Risk Assessment

Risk	Loss	Prob	Risk	Mitigation
Difficulty with blockchain development	2	3	6	I will seek advice from my supervisor about how to tackle certain problems and if necessary, what aspects of my project I should change.
Personal illness	3	2	6	Depending on the amount of lost time, I may ignore some of the SHOULD or COULD requirements.
Laptop damaged or lost	3	1	5	All work is stored using version control and periodic backups will be made and stored locally and in cloud storage. I have other devices that could be used to continue development.
The application is not finished	1	3	3	Using agile development will ensure that I will at least have a minimal working application. If I feel that I am running out of time, I will focus on testing and finalising the report.

Table 5.1: *The risk assessment of this project*

5.2 Work to Date

My work has primarily been on research, looking at how blockchain has been used to build cloud storage systems as well as how various peer-to-peer function and perform. I have proposed a design for the application to be built on the EVM.

5.3 Plan of Future Work

Below is a high level plan for the work I have remaining.

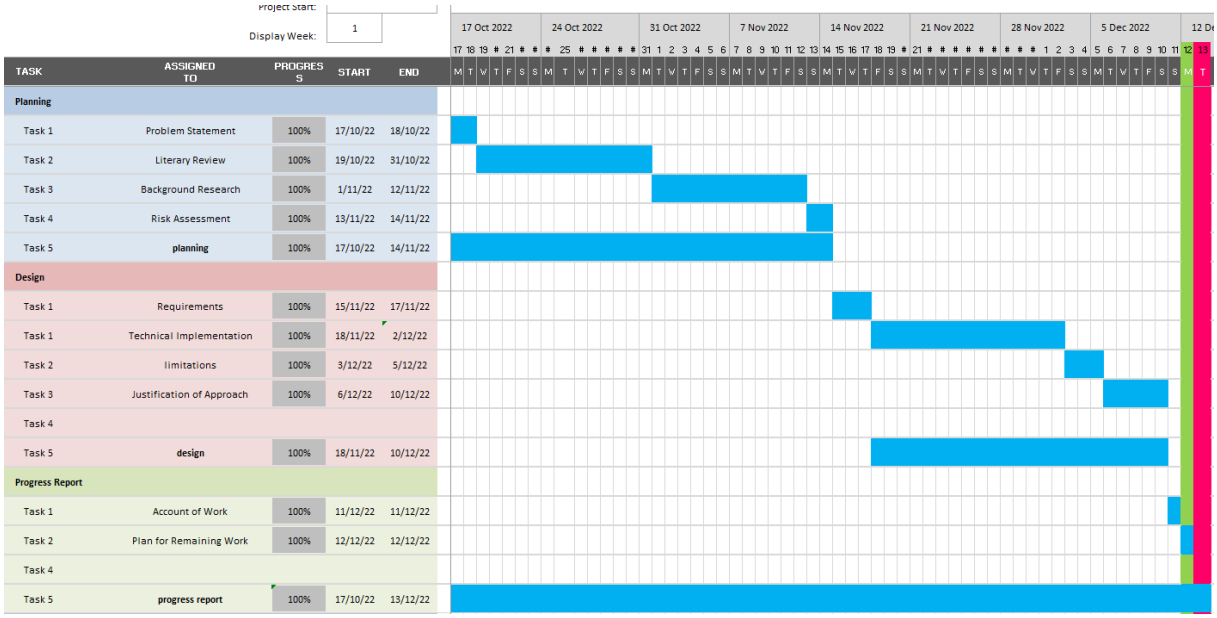

Figure 5.1: *A Gantt chart for my work up until the progress report.*

Implementation & Testing

In this phase I will use the agile development methodology to build my application. My sprints will all be structured into three phases:

1. **Preparation** Deciding on the set of requirements to complete and making any initial design decisions and diagrams,
2. **Implementation** Using test-driven development, I will work on requirements based on their prioritization, and
3. **Review** I will discuss the completed work in that sprint including design choices, what was completed, and any issues.

By using a preparation and review phase with each sprint, I can make detailed notes on my implementation as I go so when I come to write the full report I have a strong starting point.

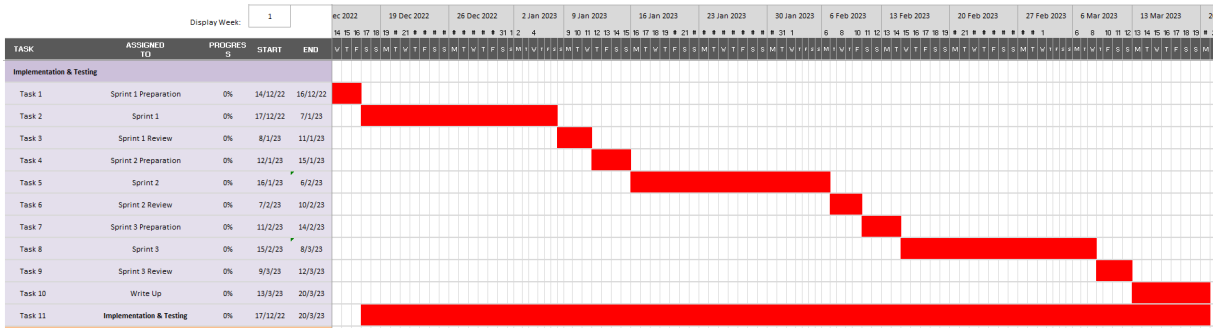

Figure 5.2: *The plan for my three sprints*

Testing Strategy and Results

This phase will consists of several sections:

1. **Testing Strategy** The approach I used for writing tests throughout the implementation phase and how these were used to evaluate the completion of the requirements set out in Section 4.1.
2. **Test Results** A report on the results of my automated and manual testing.

Evaluation

This phase will have me evaluating how successful my project was, as a whole, by focusing on several key areas:

1. **Project Organisation** How successfully did I structure my time in this project?
2. **Outcome of the Application** How successful was my application in regards to a solution to the problem set out in Section 1.1 and in terms of the requirements set out in Section 4.1?
3. **Limitations and Future Improvements** What were the limitations of my project and what would I change about it if I had more time or were to start again?

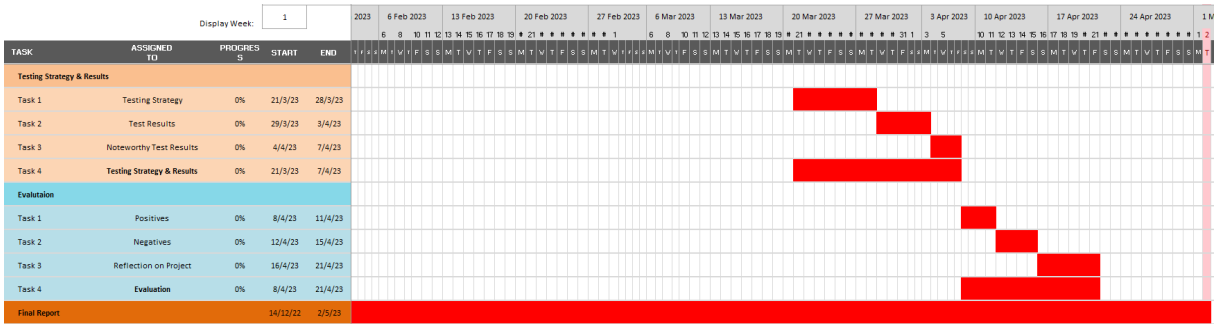

Figure 5.3: *The plan for my testing and evaluation phases*

Final Report

I will be writing the first draft of my report throughout the project, aiming to finalise sections at the end of each phase of this project. Any leftover time at the end will be focused on finishing the report.

Chapter 6

Implementation

6.1 Components

The diagram in Figure 6.1 shows the main components of the application and their relationships. The following sections will discuss these in detail and provide a justification as to their existence.

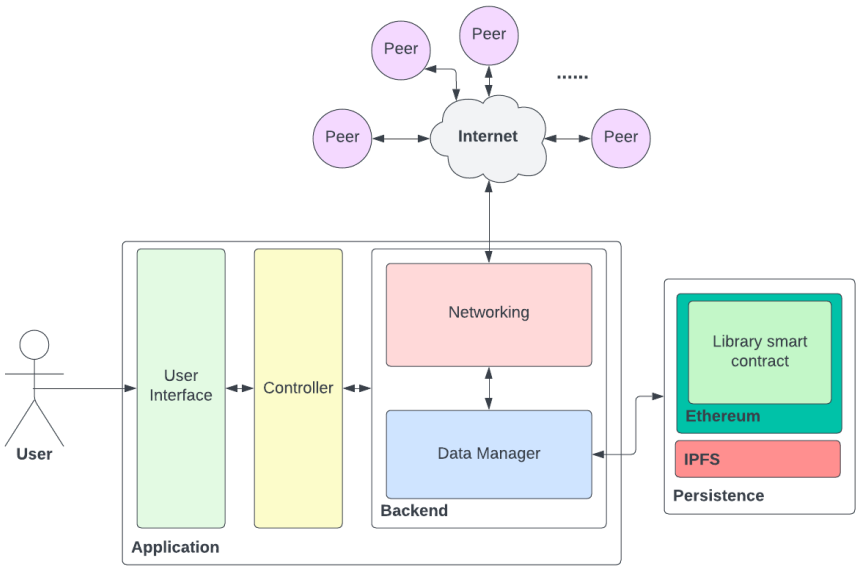

Figure 6.1: *The layers of the application*

6.1.1 Persistence

The Persistence layer represents the public data that should be accessible by any user of the application at any point in time.

Ethereum

An Ethereum Smart Contract, written in Solidity https://docs.soliditylang.org/en/v0.8.19/, will be used to store the set of data about games that is required for the identification of each game, see Section 4.2.1. The Smart Contract will also be used to perform the following:

1. **Purchasing Games** Users will purchase games over the Ethereum network and will have their address added to the set of users who own a given game. This means all users can see whether a given peer owns a game before sending them the data.
2. **Uploading a Game** Users will need to be able to upload a game to the Ethereum network and have it be visible and purchasable by all users of the application.

The go-ethereum package https://geth.ethereum.org/ will allow us to interact with the ethereum blockchain and Abigen https://docs.avax.network/specs/abigen will allow us to compile any smart contracts to Go code. This will allow us to interact with our smart contract on ethereum using a set of Go functions.

IPFS

Storing data on Ethereum is costly to the uploader, so it is important that the data we do store on it is important and minimal. However we still need to be able to store the following pieces of data such that they are publicly available:

- **Hash Tree** The tree representation of the game's data contents. Only users who purchase the game will need this.
- **Assets** Digital assets such as artwork, description, promotional work, etc.

This project will use the IPFS implementation Kubo https://github.com/ipfs/kubo, due to it being the most widely used implementation of IPFS. We will use the go-ipfs-api library https://github.com/ipfs/go-ipfs-api to interact with Kubo and upload/-download the data specified above.

Game Data

One limitation to IPFS is that the only way to store private data on it is to encrypt it. This means it would be difficult to manage and control access rights of data on IPFS over a large, distributed network where only a subset of users will actually be allowed access to it. On top of this, we would not be able to track the contributions of individual nodes sharing data such that a game developer could reward a user for their contribution. The lack of middleware support by IPFS shows that, to meet the requirements set out in Section 4.1, we would need to develop a separate platform to share game data over.

6.1.2 Backend

The backend for this application was written using Google's Go https://go.dev/ due to its simple syntax, high performance, strong standard library and excellent third parties for interacting with the Ethereum blockchain. This component can be broken down into two major parts:

- **Networking** the creation and maintenance of network connections with other peers over the internet and with the Ethereum blockchain.
- **Data Manager** the management of local data and processing of data received and to be uploaded by the networking part.

Networking

Users running this application will be a part of a distributed network of peers by creating and maintaining a set of TCP connections with other users in the network and will

communicate by sending structured messages to each other. Section 6.1.2 describes these commands in detail.

Address Verification To verify the Ethereum address of each peer, and thus which games they own and are allowed to be sent the data of, we will request a signed message using the peer's Ethereum private key (See the VALIDATE_REQ and VALIDATE_RES commands).

When two peer's initiate a connection they will need to perform a handshake that will allow them identify each other according to their Ethereum addresses.

Message Handling The main responsibility of this section is to respond to requests sent by the Data Manager by sending and tracking messages to other peers to gain the necessary information. Each message should be tracked by the peer for a given time period and resent if an appropriate response has not been received. Any duplicate requests sent by the data manager will be ignored if a pending request is active.

Commands Structured messages will typically come as part of a request/response pair involving the sharing of information between peers.

Message Format	**Description**
LIBRARY	Request that a peer sends their library of games in the form of a BLOCK message.
GAMES;[$hash_1$];[$hash_2$];...;	The user sends a list of their games as a series of unique root hashes. These root hashes will map to games on the blockchain.
BLOCK;[$gameHash$];[$blockHash$];	The user will request a block of data off of a user by sending the root hash of the game and the hash of the block being requested. The response will be a SEND_BLOCK message.
SEND_BLOCK;[$gameHash$]; [$blockHash$];[$compressedData$];	The user sends a block of data in response to a BLOCK message. The data is compressed using the *compress/flate* package to reduce message size.
VALIDATE_REQ;[$message$]	The user is requesting for this message to be signed using the receiver's Ethereum private key. This is used to verify the receiver's identity and owned collection of games.
VALIDATE_RES;[$signedmessage$]	The user responds to a VALIDATE_REQ message with a signed version of the given message. From this signature, the receiver can determine the address and public key of the user.
REQ_RECEIPT	A user will request a RECEIPT message from a peer detailing the data that has been sent by the user.

RECEIPT;[*signature*];[*message*]	A user will respond to a REQ_RECEIPT message with a signed message detailing all of the blocks that the requester has sent to the user. This will allow for users to prove their contributions to the game developer who could then reward them.
ERROR;[*message*]	An error message that can be used to prompt a peer to resend a message.

Data Manager

The data manager has several responsibilities:

1. Track the user's owned games and know which are installed and which aren't.
2. Interact with the Library contract deployed to the Ethereum blockchain to discover, upload and purchase games.
3. Fetch shards of game data to send to other peers.
4. Interact with IPFS to upload/fetch a game hash tree and store assets.
5. Sending requests to the networking layer to find and retrieve blocks of data from peers in the network.

Optimisations

Worker Pools Tasks are queued down a FIFO channel and each one is collected by a single worker thread who will perform a specific task based upon what data was sent. This allowed us to have many worker threads listening on the same channel who will complete tasks in parallel to largely increase the performance of the application.

Some of the areas this pattern was used include:

- Sharding files to create a hash tree,
- To locate and request blocks from many downloads at once, and
- To insert received data and free up the thread listening on a connection,

6.1.3 User Interface

Frontend

This application will have a GUI where users can interact with the platform and will need to include the following pages:

- **Library** The user's collection of owned games, where they can view details of each game owned as well as manage their download status.
- **Store** Where user's can find new games that have been uploaded by other users and purchase them.
- **Upload** Where user's can fill in details about a new game and have it be processed and uploaded to the blockchain.
- **Downloads** Where user's can track all of their owngoing downloads.
- **Peers** Where users can manage their list of connected peers.

Wails https://wails.io/ was chosen to create the GUI because it allows us to use a reactive web-based framework. This meant building the UI was straight-forward and allowed me to use tools that I was familiar with; namely: Vue.js v3 https://vuejs.org/, https://pinia.vuejs.org/, and SCSS https://sass-lang.com/.

Controller

The Wails package allows us to write controller functions in Go that are callable from our frontend code. These functions can be used to fetch data and trigger events in the backend code. This package also allows us to emit events that can be watched by the frontend; this will allow us to easily create a reactive user interface.

Chapter 7

Testing

7.1 Overview

My approach to testing will consist of the following principles:

1. **Test Driven Development** Tests should be written alongside the code to reduce the risk of bugs and improve robustness.
2. **Fail Fast (Smoke testing)** Automated tests should be ran in a pipeline where the fastest tests are always ran first to reduce the time spent ranning tests.
3. **Documentation** Test cases should be well documented and group contextually.

Tools

Below are the different tools I used to test my application and a justification as to why they were included.

Tool/Package	Justification
Go testing	The testing package is part of Go's standard library and will be sufficient to produce most test cases and also includes support for benchmarking and fuzzing tests.
testify by stretchr	This package is included as it provides several useful testing features that aren't present in the standard library testing package. This includes assert functions to boost code readability, mocking tools for better unit testing, setup/teardown functionality, and more.

Table 7.1: *The tools used for testing my project*

7.2 Unit Testing

Unit tests are to ensure the correct functionality of individual blocks of code within the application. This is to ensure that each function responds appropriately to both well-formed and illegal arguments and considers

7.3 Integration Testing

Profiles

Profiles are minimal versions of the application that can be run on external devices for the purposes of testing how my application fares in a simulated environment. The following profiles are included:

- **Listen Only** A client which will fetch a repository from Git and upload it as a game to the network. Once uploaded it will listen for incoming messages and reply accordingly. This is supposed to simulate an ideal peer.
- **Send Only** This client will never respond to messages but will send them periodically to the peer. This represents a selfish client.
- **Spam** This client will spam the peer with expensive messages, such as requesting a block, and should trigger the client to disconnect the spammer.
- **Unreliable** This client will represent an unreliable peer who will take a long time to respond to messasges and may reply with the wrong contents at random.

Deployed Implementation

Distributing Instances

To prove that the application could work in a live environment, we will need to deploy a set of instances of the application to different devices that communicate over the internet. An example deployment might consist of instances deployed to:

- a local machine to initiate tests,
- a VPS running on a cloud service provider,
-

Ethereum Test-Net

The Library Smart Contract, from Section 6.1.1, will need to be deployed to an Ethereum test-net to allow instances of the application, that are distributed over the

7.4 Acceptence Testing

To ensure that this application meets the requirements as described in Section 4.1, each requirement will be given a set of tests that aim prove its completeness. The type of tests will vary based upon the requirement and be ran using various devices connect to each other over the internet (**NF-M1**) (**NF-M2**) and interacting with a Smart Contract deployed to an ethereum test-net (**F-M8**).

The following acceptence tests will be included:

Id	Requirements	Description

1	(**F-M1**) (**F-M9**) (**F-M10**) (**F-C2**) (**NF-M4**) (**NF-S2**) (**NF-C1**)	A user walkthrough that shows a user uploading a game and that game being visible on the store to a separate user. A separate user will then purchase that game. Unit tests are also provided to show the correct functionality of the Smart Contract.
2	(**F-M2**) (**F-M5**) (**F-M6**) (**F-M7**) (**F-M10**) (**F-S2**) (**F-C2**) (**NF-C1**)	A user walkthrough that shows a user connecting to a peer and downloading a game off of them and those files being successfully downloaded. This assumes that the user has successfully purchased the game. Integration tests show the processing for verifying a user and data sent over the network.
3	(**F-M6**) (**F-M7**) (**NF-S1**)	Benchmark tests to show the scalability of downloading a game by varying given conditions. See Section 7.5.
4	(**F-S2**)	A user walkthrough in which a user does not validate their Ethereum address when connecting to peers. The user attempts to download blocks off of their peers but is rejected due to not being verified.
5	(**F-M4**) (**NF-S2**) (**NF-M3**) (**NF-M4**)	User walkthroughs showing two attempted uploads for a given game that already exists on the network. The first by the owner showing the process of selecting their previous game and uploading an update and another by a non-owner attempting the same thing. Unit tests for the Smart Contract can also show functionality.

7.5 Benchmarking

The key benchmark to evaluate in this project is related to how game downloads scale by varying the following factors:

1. how many of the peers we are connected to have the data we need,
2. how large is the game (in terms of average file size and number of files), and
3. the shard size used to create the hash tree.

Number of Peers

For each run, we will create a project with 500 files, each of size 80MB [1] and a shard size of $2^{22} = 4$MiB. We will then run N peers locally to simulate a perfect network connection.

Peer Count N	**Time (ms)**
1	
2	
4	

[1] These numbers were chosen to match our average game size from Section 4.2, where $500 \times 80MB = 40GB$

8	

Table 7.3: *How varying peer count affects download speed*

Game Size

For each run, we will create a project with F files of size SMB, such that $F \times S = 40GB$, and a shard size of 4MiB. We will then run 1 peer locally to simulate a perfect network connection.

File Count F	**File Size** S **(MB)**	**Time (ms)**
100	400	
50	800	
25	1,600	
5	8,000	
1	40,000	

Table 7.4: *How varying file count and size affects download speed*

Shard Size

For each run we will create a new project with 500 files, each of size 80MB, and a shard size of BMiB. We will then run 1 peer locally to simulate a perfect network connection.

Shard Size B **(MiB)**	**Time (ms)**
1	
2	
4	
8	
16	

Table 7.5: *How varying the shard size of the hash tree affects download speed*

Chapter 8

Evaluation

8.1 Development

8.1.1 Overview

Complexity My overall impression of this project was that it was too complex and took a lot more time and effort than was initially expected when planning. Whilst use of agile development helped me organise my time and prioritise features, it still felt like it wasn't enough and that I should have reduced the scope and focused on a more minimal version of the application.

Test-Driven Development TDD TDD was a critical part to the success of this project and helped me catch bugs when writing and extending the application. By having an automated test-suite, I could rapidly test large portions of my codebase and find where potential issues appeared.

However, one issue with writing tests in Go was that the test libraries used often didn't feel robust enough when compared to libraries from other languages (namely JUnit). To give some examples:

- Dependency injection and mocking are much harder in Go and rely on an interface based approach. This would add a large amount of complexity and boilerplate to my codebase for the sake of writing tests. Instead it was generally easier to write tests that relied on several other components unrelated to the tested code.
- There is no easy way to configure functions to run before/after all/each test. You can run functions before and after all tests in a package but not individual test files, and before/after each function have to specified in every test they're in.
- Tests are not easily groupable and tags can only be specified per file but not on a test by test basis. The standard package only supports categorising individual tests as *short* or not.

8.2 Future Work

8.2.1 Optimisations

There are several changes that we could make to improve the performance of the project as a whole and make the downloading of data more efficient. These include:

1. **New Commands** We could extend the commands described in Section 6.1.2 by including support for batch block requests. This would allow users to request many blocks at once and avoid the overhead of having to perform one request per block.
2. **Better Block Selection** Currently blocks are not ranked in any way, and by considering the ideas set out in Section 2.1, we could improve the download speed and overall availability and efficiency of data throughout the network.
3. **Represent Data Differently** The main issue with modelling our data as a Hash Tree is that each file needs to have at least one block. This means that for projects with a large amount of small files, we are substantially increasing the download time as blocks are requested one at a time.
   One possible solution would be to represent the directory as a contiguous block of data that contains breakpoints between files and directories that can be indexed. This would mean we could reduce the overall number of blocks and reduce the amount of redundant data sent between nodes.

8.2.2 New Features

Downloadable Content DLC

DLC was initially considered as a requirement for this project (F-S3 & NF-S3) but was not completed due to time constraints. DLC would be a necessary addition to this application to ensure the viability of it as a competitor to existing platforms. See the potential implementation details specified in Section

Discovery

Content Discovery The store aspect of the application is limited in that users have to know the root hash of the game to find the game's store page and purchase it. This could be achieved using several techniques:

- **Indexing** Periodically generate an index of all games uploaded that allows users to easily search the store without having to make a large amount of requests to the blockchain. Games could also be ranked based upon factors like popularity of availability.
- **More Metadata** Add further metadata to games that allows them to be grouped and thus more easily searchable. For example, developers could add a set of tags to games to identify what type of game it is or what features it has.

Peer Discovery Currently, there is not good way to discover peers who have the content that a user is interested in. This could be implemented using the following:

- **Tracker** BitTorrent uses tracker's to store a list of peers who are interested in a particular piece of data. A similar technique could be implemented to allow users to easily find new peers but would require this data to be hosted somewhere.
- **Neighbour Discovery** Query known peers if they know any users who are interested in a particular piece of content. However given that the network is fragmented, there is no guarantee that a peer could be found this way.

Streamline Updates

At the moment, each update to a game is considered as a new game entirely and this has the limitation that a user will have to download the entirety of the game again and not just the updated sections. This also doesn't overwrite any existing data so the user will have to manually uninstall the older version.

A future improvement would be to streamline this process so that for a given update, only the changed data is downloaded and it is overwritten in-place on the older version of the game. This would drastically reduce the bandwidth requirement for each update and require less manual work from the user.

Appendix A

Screenshots

Figure A.1: *The login page where a user will enter their Ethereum private key and connect to a BlockWare contract instance.*

Figure A.2: *The home page where users can navigate between the main individual pages.*

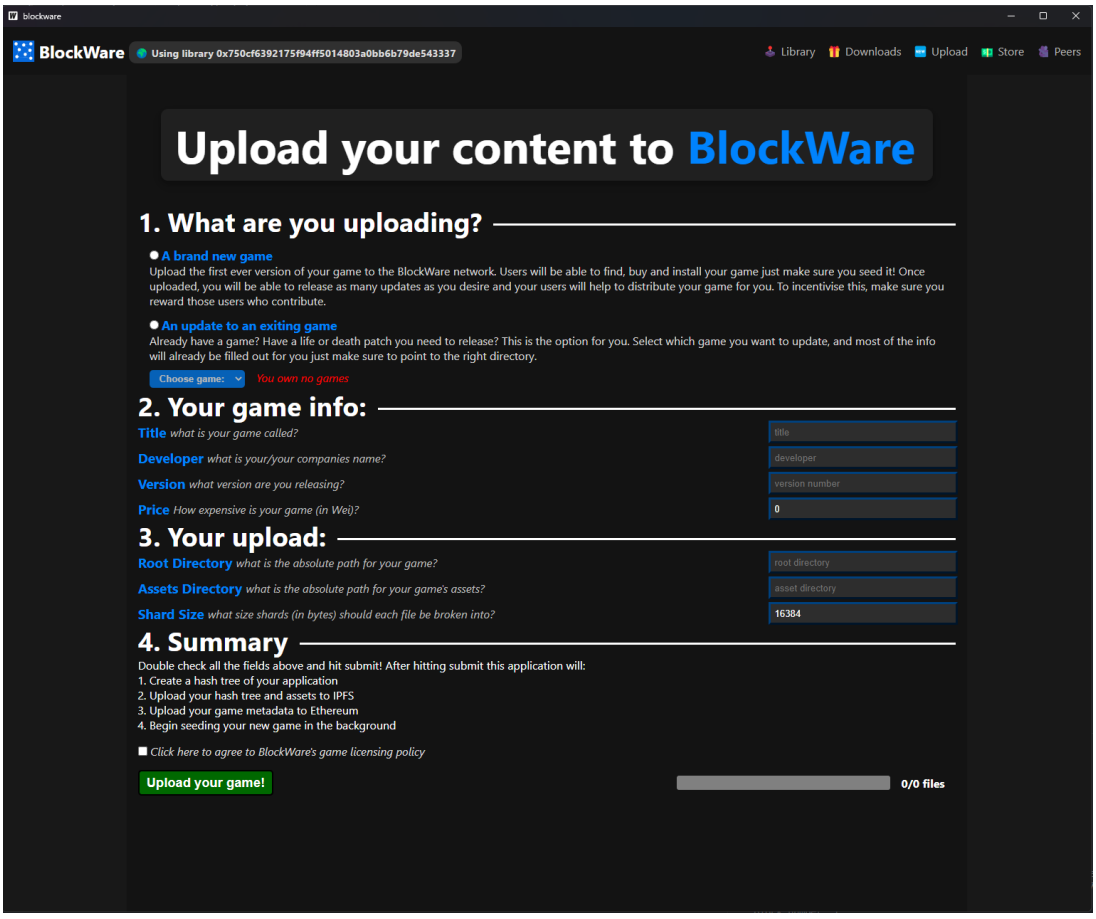

Figure A.3: *The page where users input the details about their game and can upload it to the Ethereum network.*

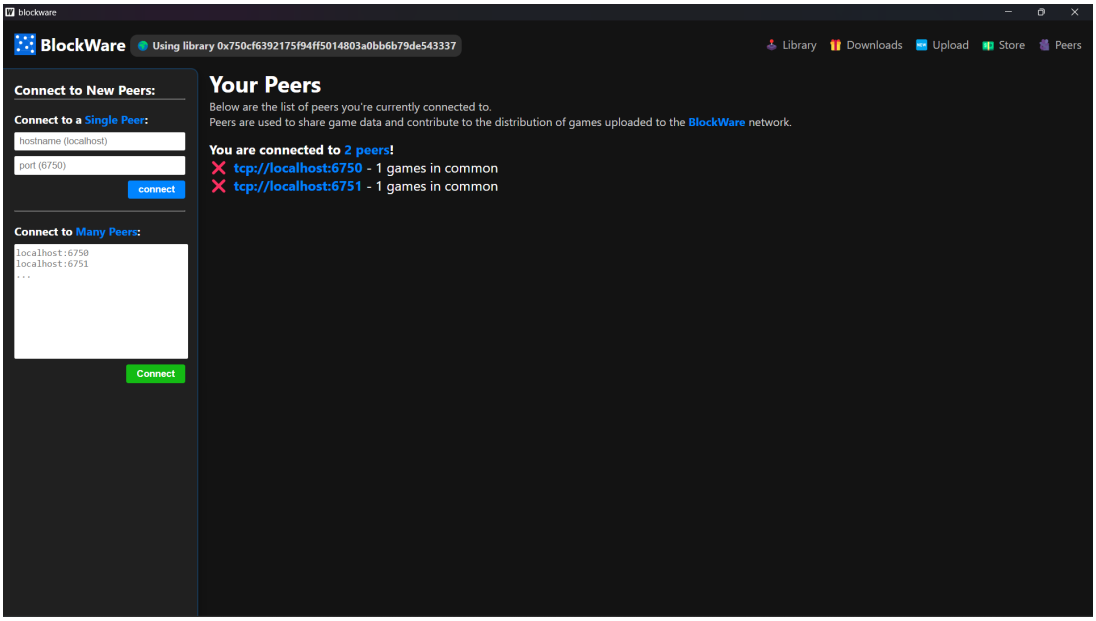

Figure A.4: *The page where users can manage their connections to peers with whom they will download game data off of.*

Bibliography

[1] Benet, J. IPFS - Content Addressed, Versioned, P2P File System, July 2014. arXiv:1407.3561 [cs].

[2] Brown, A. Valve defends taking 30 per cent cut of Steam sales in response to lawsuit, July 2021.

[3] Chen, Y., Ding, S., Xu, Z., Zheng, H., and Yang, S. Blockchain-Based Medical Records Secure Storage and Medical Service Framework. *Journal of Medical Systems 43*, 1 (Nov. 2018), 5.

[4] Hartman, J., Murdock, I., and Spalink, T. The Swarm scalable storage system. In *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)* (June 1999), pp. 74–81. ISSN: 1063-6927.

[5] Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, 1 (Feb. 1988), 51–81.

[6] Kaune, S., Rumín, R. C., Tyson, G., Mauthe, A., Guerrero, C., and Steinmetz, R. Unraveling BitTorrent's File Unavailability: Measurements and Analysis. In *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)* (Aug. 2010), pp. 1–9. ISSN: 2161-3567.

[7] Li, J., Wu, J., and Chen, L. Block-secure: Blockchain based scheme for secure P2P cloud storage. *Information Sciences 465* (Oct. 2018), 219–231.

[8] Li, J., Wu, J., Chen, L., and Li, J. Deduplication with Blockchain for Secure Cloud Storage. In *Big Data* (2018), Z. Xu, X. Gao, Q. Miao, Y. Zhang, and J. Bu, Eds., Communications in Computer and Information Science, Springer, pp. 558–570. event-place: Singapore.

[9] Manzoor, A., Liyanage, M., Braeke, A., Kanhere, S. S., and Ylianttila, M. Blockchain based Proxy Re-Encryption Scheme for Secure IoT Data Sharing. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)* (May 2019), pp. 99–103.

[10] Marks, T. Report: Steam's 30% Cut Is Actually the Industry Standard, Oct. 2019.

[11] Morris, J. H., Satyanarayanan, M., Conner, M. H., Howard, J. H., Rosenthal, D. S., and Smith, F. D. Andrew: a distributed personal computing environment. *Communications of the ACM 29*, 3 (Mar. 1986), 184–201.

[12] Neglia, G., Reina, G., Zhang, H., Towsley, D., Venkataramani, A., and Danaher, J. Availability in BitTorrent Systems. In *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications* (May 2007), pp. 2216–2224. ISSN: 0743-166X.

[13] Pouwelse, J., Garbacki, P., Epema, D., and Sips, H. The Bittorrent P2P File-Sharing System: Measurements and Analysis. In *Peer-to-Peer Systems IV* (Berlin, Heidelberg, 2005), M. Castro and R. van Renesse, Eds., Lecture Notes in Computer Science, Springer, pp. 205–216.

[14] Saroiu, S., Gummadi, P. K., and Gribble, S. D. Measurement study of peer-to-peer file sharing systems. In *Multimedia Computing and Networking 2002* (Dec. 2001), vol. 4673, SPIE, pp. 156–170.

[15] Sharma, P., Jindal, R., and Borah, M. D. Blockchain Technology for Cloud Storage: A Systematic Literature Review. *ACM Computing Surveys 53*, 4 (July 2021), 1–32.

[16] Wang, L., and Kangasharju, J. Measuring large-scale distributed systems: case of BitTorrent Mainline DHT. In *IEEE P2P 2013 Proceedings* (Sept. 2013), pp. 1–10. ISSN: 2161-3567.

[17] Wang, S., Zhang, Y., and Zhang, Y. A Blockchain-Based Framework for Data Sharing With Fine-Grained Access Control in Decentralized Storage Systems. *IEEE Access 6* (2018), 38437–38450. Conference Name: IEEE Access.

[18] Yue, D., Li, R., Zhang, Y., Tian, W., and Peng, C. Blockchain Based Data Integrity Verification in P2P Cloud Storage. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)* (Dec. 2018), pp. 561–568.