Chapter 1

Problem Statement

1.1 The Problem

Video games are large, highly popular pieces of software that are typically obtained through centralised platforms, like Steam or Epic Games. These platforms offer high availability, social features and customer support at the expense of:

- taking a large cut of revenue from developers[1],
- being prone to censorship from entities like governments[2],
- relying on a single platform to stay active, distribute games and maintain a user's ownership[3], and
- collecting large amounts of data from their users.

1.2 Goals

This project aims to produce a proof-of-concept, decentralised video game marketplace, using blockchain technology, that will allow developers to continuously release and update their games on a public network where they directly interact with their users. This network should be built on top of a peer-to-peer file-sharing network to allow users to to distribute games.

1.3 Scope

This project will consist of the following components:

1. An Ethereum smart contract that will allow us to maintain a library of games that can be queried and added to by any user.
2. A local application to be run by users to interact with the smart contract and allow users to join a peer-to-peer network where they can download and upload games to.

[1]Steam take a 30% cut [1, 2]

[2]See the Chinese version of Steam [3]

[3]For example, the Nintendo eShop closing down [4]

Chapter 2

Background Research

2.1 BitTorrent

Motivation Due to the typical size of video games, it is impractical for every user to have a copy of every game. Therefore, it is important that I understand how a popular example of a peer-to-peer P2P file sharing system works when data is fragmented across the network.

BitTorrent [5, 6] is a protocol for sharing data across a distributed network[1] and it was chosen here because of its popularity, being responsible for 3.35% of global bandwidth [10]. In BitTorrent, users barter for blocks of data from a network of peers in a tit-for-tat fashion, such that users with a high upload rate will also typically have a high download rate.

Key Ideas

Trackers A tracker server keeps track of which peers have what data, which of those peers are available at, and will provide network statistics to *recommend* which peers to connect to first.

Block Priority Users will download blocks from other peers using the following priority:

1. **Strict Priority** Data is split into pieces and sub-pieces where pieces are ideally downloaded together.
2. **Rarest First** Download the pieces that have the fewest copies on the network to boost availability.
3. **As Soon As Possible** A user with no peers will try get a random piece quickly so they can contribute to the network.

Hashing Hashing privdes a unique fingerprint of some data, which can be used to verify the integrity of it. As data is broken into pieces, invalid pieces can easily be fetched again without interfering with any other pieces.

Optimistic Unchoking A peer will allocate a portion of their bandwidth for communicating with unknown peers. This will allow new users to join the network and be able

[1]Popular implementations include BitTorrent Web [7], qBittorrent [8], and µTorrent [9]

to contribute without being ignored by most peers and it gives a way for existing peers to seek better peers.

Availability

One of the most significant issues facing BitTorrent is the availability of torrents, where *'38% of torrents become unavailable in the first month'* [5] and that *'the majority of users disconnect from the network within a few hours after the download has finished'* [6]. This paper [11] looks at how the use of multiple trackers for the same content and DHTs can be used to boost availability. However, good availability requires users to *choose* to contribute and often the built-in incentives aren't enough to encourage users to contribute for a long period of time.

2.2 Ethereum

Ethereum [12, 13] is a distributed transaction-based blockchain that comes with a built-in Turing-complete programming language that allows any user to design their own transactions. Each block will include a list of transactions, where each transaction includes bytecode that can be run by each node in the network to update state.

Smart Contracts

A smart contract is an executable piece of code, usually written in Solidity [14], that will automatically execute on every node in the Ethereum network when certain conditions are met. Smart contracts are enforced by the network and remove the need for intermediaries and reduce the potential of contractual disputes, due to their transparency and immutability.

Gas is the computational effort of running a smart contract and must be paid, in Ether, before a transaction can be 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.

Miners receive Ether for mining transactions based upon their gas price, which results in gas price varying according to supply and demand. For example, in a period of congestion the gas price will increase and users will want their transactions processed more quickly.

Test Networks

An Ethereum test network is an instance of Ethereum in which users can deploy their smart contracts and test them in a live environment. Ether for these networks can be gained for free from a faucet provided by a node from the network. Some notable examples include Sepolia [15], Goerli [16], and Ropsten[2].

[2]The Ropsten test-net is declared as end of life as of December 2022 [17]

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. Table 3.1 details some examples of how blockchain has been used to create cloud storage platforms:

Paper	**Description of Solution**
Blockchain Based Data Integrity Verification in P2P Cloud Storage [18]	This paper looks at how varying structures of Merkle Trees can be used to verify the integrity of data within a P2P blockchain cloud storage system.
Deduplication with Blockchain for Secure Cloud Storage [19]	A deduplication scheme that uses the Ethereum blockchain to record storage information and distribute files across multiple servers
Block-secure: Blockchain based scheme for secure P2P cloud storage [20]	Users divide their own data into encrypted blocks and upload them randomly into a blockchain, P2P network. It uses a custom genetic algorithm to solve the file block replica placement problem and ensure data availability.
Blockchain-Based Medical Records Secure Storage and Medical Service Framework [21]	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 [22]	This solution proposes an attribute-based encryption scheme that allows the distribution of access keys to users based upon their allocated groups. Data is uploaded to IPFS and uses an Ethereum smart contract to implement a keyword search of the data stored.
Blockchain based Proxy Re-Encryption Scheme for Secure IoT Data Sharing [23]	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 [24]*

Gaps Some of the gaps found include:

- **Access Control** Many of these include no dynamic access-control system, and [22] requires the owner of the data to manually manage access rights.
- **Versioning** Many of these systems rely on platforms like IPFS, which do not natively allow for versioning without uploading a new set of data entirely.

This project will allow for dynamic access control through a payment system based on Ethereum and will allow for pieces of data to be versioned to match a more realistic lifecycle for a piece of software.

3.2 P2P File Sharing

Table 3.2 looks at various implementations of P2P file-sharing networks. Using the pros and cons of each will help me identify key considerations of my own implementation.

System	**Description of Solution**
IPFS [25]	IPFS is a set of protocols for transferring and organising data over a content-addressable, peer-to-peer network. IPFS is open source and has many different implementations, such as Estuary [26] or Kubo [27].
Swarm [28]	Swarm is a distributed storage solution linked with Ethereum that has many similarities with IPFS [6]. 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.
BitTorrent [6]	See Section 2.1.
AFS [29, 30]	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 [31]	Napster uses a central cluster of servers that maintain an index of every file on the network and which users have a copy of it. Clients will query the cluster for this information and will choose peers based upon their bandwidth.
Gnutella [31]	In Gnutella, nodes form an overlay network and will discover other peers through *ping-pong* messages, where any node that receives a *ping* message will forward it to their neighbours and send a *pong* message to the originator. A user will flood a download request to their peers until they find a suitable peer to download off of.

Table 3.2: *Various global distributed file systems.*

Gaps The main issues involving these networks are:

1. **Trust** Nodes are typically anonymous so users have to trust that the data they download isn't malicious. Moreover, there are typically no measures included to take down harmful or illegal content.
2. **Payment** None of the networks natively support payment for content or include mechanisms for large-scale access control.

This project seeks to tie a traditional P2P file-sharing network with the Ethereum blockchain to allow for greater trust through publicly identifiable uploaders and allow users to pay for content and be stored in a public record.

Chapter 4

Design

4.1 Analysis

4.1.1 Stakeholders

Game Developers PRIMARY

This group will upload games, and any subsequent updates, to the applications. Users will purchase access to their games and contribute to the distribution of it in return for some reward that could be provided by this group.

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.

4.1.2 Requirements

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

Functional

ID	**Description**
	Must
F-M1	Developers must be able to release games by uploading metadata to the Ethereum blockchain.
F-M2	Developers must be able to release updates to their existing games.
F-M3	An owner of an existing game must be an owner of all future updates to that game.

F-M4	This application must include a smart contract that is deployable to the Ethereum blockchain[1].
F-M5	Users must be able to purchase games off of develoeprs.
F-M6	Users must be able to prove they have purchased a game.
F-M7	Users must be able to create and maintain many concurrent connections to other users.
F-M8	A user must be able to communicate with other users by exchanging structured messages.
F-M9	A user must be able to upload and download data to and from other users.
F-M10	A user must be able to verify the integrity of all data that they download.
F-M11	A user must be able to download games in their entirety.
F-M12	A developer must be able to upload a hash tree[2] of a game such that all users can access it.
	Should
F-S1	Users should only upload game data to users who own that game.
F-S2	Users should interact with the application using a GUI.
F-S3	Users should be able to prove the amount of data that they have uploaded to other users.
F-S4	Users should have a way to discover new peers from their existing ones.
	Could
F-C1	Developers could be able to release downloadable content (DLC) for their games.
F-C2	Allow developers to upload promotional materials such as cover art and an overview to be shown to the user.

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

Non-Functional

ID	Description
	Must
NF-M1	This application must be decentralised and cannot be controlled by any singular party.
NF-M2	Any user must be able to join and contribute to he network.
NF-M3	Developers who upload games to the network must be publically identifiable.
NF-M4	The data required to download a game must be immutable.
NF-M5	Only the original uploader must be able to make any changes or release any updates to a game.

[1]This project will only test deploying to an Ethereum test network
[2]See Section 4.2.

	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	This application's GUI should be intuitive to use for new users.
	Could
NF-C1	This application could include measures to prevent/stop the distribution of illegal content.
NF-C2	The GUI could include detailed support and or instructions for new users.

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

4.2 Design Considerations

Data

Table 4.3 discusses the different types of data we are going to need to store and where they should be stored based upon their properties.

Data	**Size**	**Location**	**Explanation**
Game Metadata (**F-M1**)	100 – 200B	Ethereum	This data is the minimal set of information required for the unique identification of each game. See Section 4.2. This data is appropriate to store on Ethereum as it is public, small in size, and essential to the correct functioning of the application as all users will need to be able to discover all games.
Game Hash Tree (**F-M12**)	˜15KB	IPFS	This will be the compressed hash tree that will allow users to identify and verify the blocks of data they need to download for a game. The user will download this immediately after purchasing the game. This data is public but its size will make it costly to store on Ethereum at a large scale and given that only a subset of users will actually ever want access to it, it would be wrong to store it on Ethereum. Instead IPFS can be used for reliable and fast access at a very large scale and we can embed the generated CID, from the upload to IPFS, in our smart contract instead.

Game Assets (**F-C2**)	Variable[3]	IPFS	This will represent any promotional material provided for the game that can be viewed on the game's store page. This will typically include cover art and a markdown file for the description and isn't required to purchase or download the game. The user will download this when they first view the game in the store. For similar reasons as the hash tree, this data will be also be stored on IPFS and have its CID stored on Ethereum instead.
Game Data	*avg. 44GB* [4]	Peers	This will the data required to run the game and will be fetched based upon the contents of the game's hash tree. This data is very large and has restricted access so wouldn't be appropriate to store on either Ethereum or IPFS[5]. Therefore, this project will use a custom P2P network for sharing data, which is described in Section 4.2.

Table 4.3: *The different types of data required for each game.*

Swarm [28] was considered as a decentralised storage and distribution platform over IPFS but was decided against as it would couple this project more tightly with Ethereum and the fact that IPFS has a much greater adoption.

Blockchain

Type of Blockchain

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

- being accessible to more users, which will boost both 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-M4**).

Uploading Games

To satisfy (**F-M1**) and (**F-M2**), the data stored on the blockchain will be used for the identification of games. Table 4.4 shows the fields that will stored as part of the smart contract for each game and to manage the whole collection of games. Fields in *italics* are generated for the user and non-italic fields are entered manually.

[3]Some games may include many promotional materials, whilst some could include none. Therefore, it is hard to estimate the expected size.

[4]Calculated based off of the top 30 games from SteamDB [32].

[5]IPFS and similar platforms provide no access control for the data stored there and any encryption based technique would be unviable.

Name	**Description**
	Metadata for each game
title	The name of the game.
version	The version number of the game.
release date	The timestamp for when the game was uploaded.
developer	The name of the developer uploading the game (**NF-M3**).
uploader	The Ethereum address of the developer (**NF-M3**).
root hash	The root hash of the game that uniquely identifies the game and is based upon its contents.
previous version	The root hash of the most previous version of the game if it exists.
price	The price of the game in Wei.
hash tree CID	Required for downloading the hash tree from IPFS.
assets CID	Required for downloading the assets folder from IPFS.
	Managing the Collection of Games
library	A mapping for all games uploaded to the network, where a game's root hash is the key used to find its metadata.
game hashes	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 (**F-M6**).

Table 4.4: *the data to be stored on Ethereum using a smart contract*

Purchasing Content

Users will purchase games from developers over Ethereum by transferring Ether (**F-M5**). The user's address will then be added to a public record, on the smart contract, of all users who have purchased the game (**F-M6**). Upon purchasing a game, a user will broadcast their new library to all of their peers.

Distributed File Sharing

Hash Tree

The hash tree of a given directory is used to represent its structure as well as the contents of its files. Each file is represented by an ordered list of SHA-256 hashes that match a fixed-size block of data. This allows users to easily identify and verify game data (**F-M10**).

Uploading Content

For a developer to upload their game (**F-M1**), they must provide the following:

- the metadata outlined in Section 4.2,
- a hash tree created from the root directory of the game, and
- an assets folder containing a piece of cover art *(cover.png)* and a description file *(description.md)*.

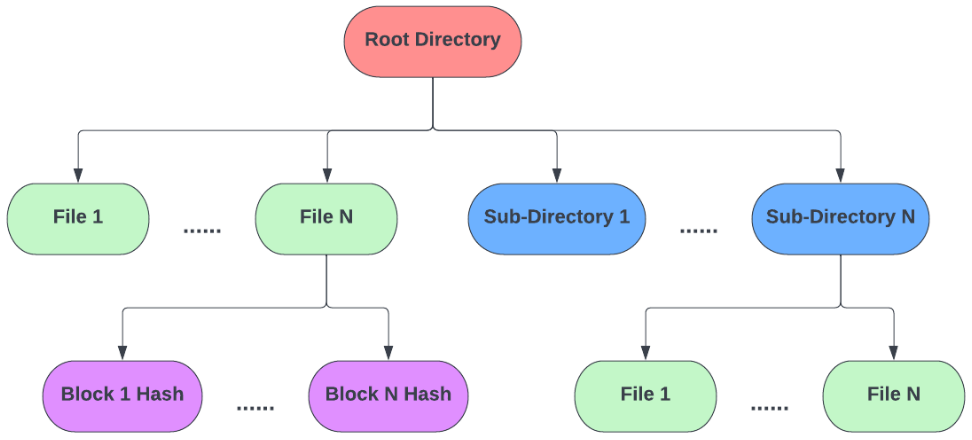

Figure 4.1: *The structure of a hash tree*

The developer should be able to enter the required fields into an upload page of the GUI and have the data generated and uploaded for them (**F-S2**).

Downloading Content

Like mentioned in Section 4.2, it is impractical to store the game's data on the blockchain or IPFS. Instead we will consider ideas from decentralised file-sharing networks, like discussed in Sections 3.2 & 2.1.

Games are content addressable using their root hash field, which will allow users to request data from that game from other users. When a peer seeking data $P_{downloader}$ forms a connection with another peer P_{seeder} they will:

1. Perform a handshake to determine each other's Ethereum address and public key.
2. P_{seeder} will verify that $P_{downloader}$ owns the game by checking the *purchased* mapping on the smart contract (**F-M6**) (**F-S1**).
3. $P_{downloader}$ will send requests for individual blocks to P_{seeder} (**F-M9**).
4. Upon receiving a block, $P_{downloader}$ will verify the contents using the block's hash (**F-M10**) before writing it to disk in the appropriate location.
5. Repeat Steps 3–4 until the entire game has been downloaded (**F-M11**).
6. P_{seeder} may request a signed receipt that details the blocks they uploaded (**F-S3**) to $P_{downloader}$.

Users will be able to connect to many peers at once (**F-M7**) and will send download requests to the subset of their peers who also own the game. Requests will be sent in a round-robin fashion to evenly distribute the requests and prevent overloading a single peer (**NF-S1**). Requests that cannot be completed will be retried when connecting to a new peer or when a peer has a change in library.

See Figure 4.3 for a diagram showing these interactions fully.

Updating Content

To satisfy (**F-M2**), developers will perform the same steps outlined in Section 4.2 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 (**F-M3**). Additionally, this will include the restriction that only the original uploader can upload an update for their game (**NF-M5**).

Each version is considered 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.

Downloadable Content

Downloadable Content (DLC) (**F-C1**) represent optional additions for games that users will buy separately. DLCs will act similarly to how updates are treated. Each DLC will need:

1. **Dependency** The root hash of the oldest version of the game this DLC supports.
2. **Previous Version** (Optional) The root hash of the previous version of the DLC.

Users must own the original game to buy any of its DLC.

Proving Contribution

As a user downloads blocks of data, they will keep track of which users have sent them which blocks. A peer may then request their contributions in the form of a signed message that can be sent to the developer (**F-S3**) in return for some kind of reward. The contents of the reward isn't specified for this project but could include in-game items, digital assets or Ether. This solution assumes that developers have knowledge of which Ethereum address maps to which of their game's users.

4.3 Architecture

This application is structured using the Model-View-Controller MVC pattern to create a separation of concern between its main layers. Figure 4.2 shows a high level overview of the architecture and below I discuss the purpose for each.

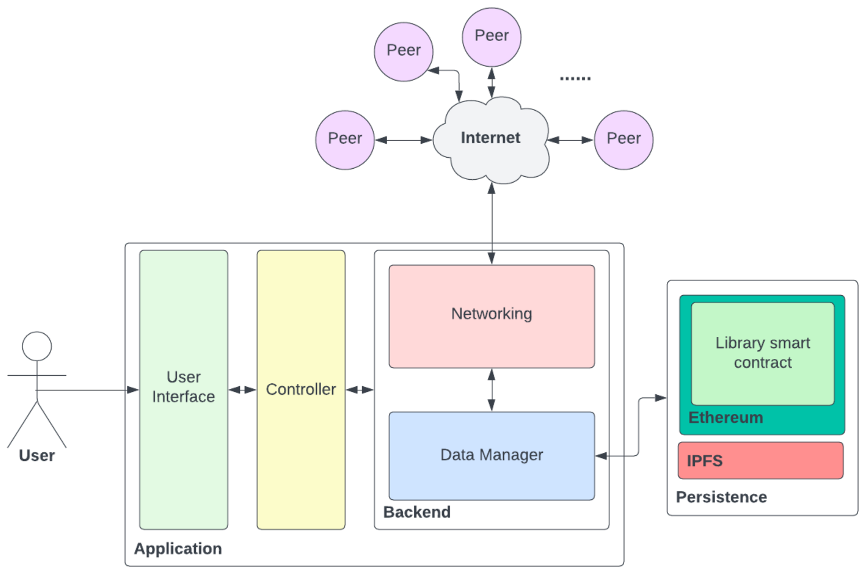

Figure 4.2: The layers of the application

Persistence

The Persistence layer shows how the data for the application is divided across several mediums; namely the **Ethereum Smart Contract**, **IPFS**, and a **P2P Network**. Each component stores a different type of data, which is outlined in Section 4.2.

There are several things to note about using Ethereum as a platform for selling games:

- Ethereum is a less stable currency than most traditional currencies like GBP or USD so games may fluctuate largely in price. This design describes no solution to this issue but an example solution might attempt to map a real-world currency to Ether.
- All write functions to the smart contract will incur a gas fee so uploading or updating data will not be free.
- Users will have to source Ether from elsewhere before being able to purchase games, which may be intimidating to users who are not already familiar with the Ethereum ecosystem. Moreover, many people are sceptical of blockchain technology so may be hesitant to adopt the application.

This layer will also provide interface functions that will allow our backend code to interface with our smart contract. Interactions with IPFS will be very simple uploads or downloads of compressed data.

Backend

The Backend can be broken down into two major components:

- **Networking** The creation and maintenance of network connections with other peers over the internet with the purpose of sharing data.
- **Data Manager** the management of local data and the processing of data received and to be uploaded by the Networking component.

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 (**F-M7**) in the network and will communicate by sending structured messages to each other (**F-M8**). Section 4.3 describes these commands in detail.

Address Verification When two peers connect they will perform a handshake to exchange their Ethereum addresses and public keys by sending signed messages to each other. This will allow a peer to identify what games another peer is allowed access to.

Message Handling One of the responsibilities of this component is to respond to requests sent by the Data Manager by sending and tracking messages to other peers to fetch requested data. 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 (**F-M8**) will typically come as part of a request/response pair involving the sharing of information between peers. Command responses are not awaited to remove unnecessary blocking of the connection channel as a user may be responding to many different requests at once by the same peer. Table 4.5 shows the list of commands used bu the application.

Message Format	**Description**
LIBRARY	Request that a peer sends their library of game.
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 peer by sending the root hash of the game and the hash of the block being requested. The response will be a SEND_BLOCK message (**F-M9**).
SEND_BLOCK;[$gameHash$]; [$blockHash$];[$compressedData$];	The user sends a block of data in response to a BLOCK message (**F-M9**). The data is compressed using the *compress/flate* package to reduce message size (**NF-S1**).
VALIDATE_REQ;[*message*]	The user is requesting for a message to be signed using the receiver's Ethereum private key. This is used to verify the receiver's identity and thus their owned collection of games (**F-S1**).
VALIDATE_RES;[*signedmessage*]	The user responds to a VALIDATE_REQ message with a signed version of the received message. From this signature, the receiver can determine the address and public key of the user (**F-S1**).
REQ_RECEIPT;[$gameHash$]	A user will request a RECEIPT message from a peer detailing the data that has been sent by the user for a specific game (**F-S3**).
RECEIPT;[$gameHash$];[*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 from a given game. This will allow for users to prove their contributions to the game developer who could then reward them (**F-S3**).
REQ_PEERS	A user requests the list of peers which the receiver peer is connected to. This will be sent immediately after a peer's identity is validated and will help increase the connectivity in the network (**F-S4**).
PEERS;[p_1*hostname*] : [p_1*port*];...	A user will send a list of their active peers. This will be limited to those peers which they have connected to and thus know the hostname and port of their server (**F-S4**).
ERROR;[*message*]	An error message that can be used to prompt a peer to resend a message.

Table 4.5: *The set of structured messages sent between peers*

Data Manager

The Data Manager has several responsibilities:

1. Track the user's owned games and know which are installed locally and which aren't.
2. Interact with the Library contract deployed to the Ethereum blockchain to discover, upload and purchase games (**F-M1**) (**F-M2**) (**F-M5**).
3. Fetch shards of game data from the disk to send to other peers.
4. Interact with IPFS to upload/fetch a game hash tree and assets (**F-M12**) (**F-C2**).
5. Sending requests to the networking layer to find and retrieve blocks of data from peers in the network.
6. Generate hash trees for games.

Frontend & Controller

Frontend

This application will have a GUI (**F-S2**) (**NF-S2**) where users can interact with the platform. Having a GUI is essential to making the platform as easy to use as possible so that it is accessible to new users. At minimum it will need to include the following pages:

- **Library** The user's collection of owned games, where they can view details for each game as well as manage their download status.
- **Store** Where user's can find and purchase new games that have been uploaded by other users.
- **Upload** Where users can fill in details about their new or updated game and have it be uploaded to the network.
- **Downloads** Where user's can track all of their ongoing downloads and see their progress.
- **Peers** Where users can manage their list of connected peers. Here a user can form new connections, break existing ones and request specific data from their peers.
- **Help** A help page to describe the application and all of its functionality (**NF-C2**)

To satisfy (**NF-M3**), a developer must always be displayed with both their chosen name and their Ethereum address. A developer should publically provide their Ethereum address to ensure their users can identify it.

Controller

The Controller will be represented as a set of interface functions that allow the backend and frontend code to communicate. This can be done to trigger actions such as starting a game download or to fetch data like the list of a user's owned games.

4.4 Benefits

This application presents the following benefits when compared with centralised game marketplaces:

- **Privacy** A user's personal information and usage isn't collected. Traditional platforms require users to enter personal information and will actively collect data about a user's actions through the platform.
- **Ownership** A user's ownership of a game isn't tied to a single platform and use of Ethereum means that a user's ownership is upheld by all computers in the network.

- **Censorship** Similar to the previous point, no one party has control over the platform so it is much harder for third parties, such as governments, to restrict the content uploaded to it.
- **Profits** Developers and gamers communicate directly and this means developer's won't have to pay a hefty fee for a middle-man. This will result in potentially larger profits for the developers.

4.5 Limitations

This application presents the following limitations when compared with a centralised game marketplace:

- **No Social Features** Social features, such as friends or achievements, were not included within the scope of this project.
- **Availability** Section 2.1 highlights the issue of availability within P2P file-sharing systems and it is likely this platform will face similar issues. The use of a contribution system was implemented to help identify those users who have been contributing but there is no automatic rewards system[6].
- **Inefficient Updates** As updates are treated as individual games, they will require users to download the entire game again. This is highly inefficient and results in lots of duplicate data being downloaded.
- **Hash Trees** Modelling data as a hash tree means that each file will need at least one block so a game may have a large amount of blocks for not a lot of data and as each block has to be requested separately this will add a lot network overhead.

[6]An example would be how the micro-payment system works in Swarm [28]

4.6 Sequence Diagram

Figure 4.3 shows how all of the components from this section are combined to download a game off of a single peer in the network.

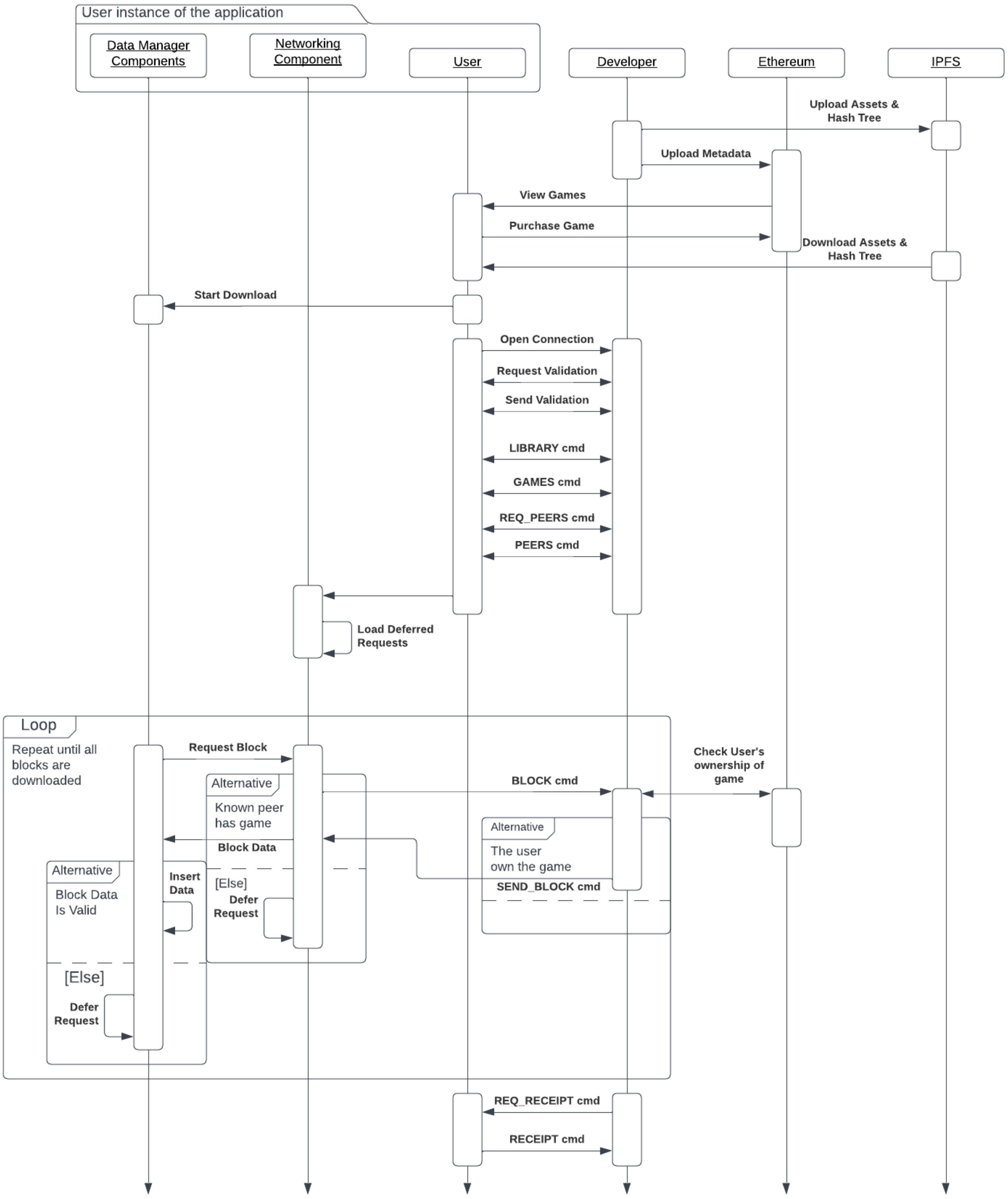

Figure 4.3: *A sequence diagram showing the main interactions needed to download a game from a single peer*

Chapter 5

Implementation

5.1 Backend

Table 5.1 shows the tools to create the backend of my application as it was described in Section 4.3.

Tool	Description & Reasoning
Go [33]	Go was chosen because of its simple syntax, high performance, strong standard library and third party packages for interacting with Ethereum.
go-ipfs-api [34]	A Go package used for interacting with the Kubo implementation of IPFS [27] that gave an easy interface for downloading and uploading data to Kubo.
go-ethereum [35]	A collection of tools used for interacting with Ethereum including an Ethereum CLI client (Geth), and a tool for converting Ethereum contracts into Go packages (abigen). This was essential for interfacing with the smart contract.
Zap [36]	A logging library that performs much better and provides easier customisation when compared to Go's standard library implementation.
Viper [37]	A configuration file management library that helps read, write, and access configuration options written to file. This makes it simple to access global configuration settings and made it easy to configure test profiles.

Table 5.1: *The tools used to develop the backend*

5.2 Smart Contract

Table 5.2 shows the tools used to create and deploy the Library smart contract as it was described in Section 4.2.

Tool	Description & Reasoning
Solidity [14]	The first language used to write smart contracts for the Ethereum blockchain. Most tutorials are focused around Solidity so it made the most sense to learn it for this project.

Sepolia [15]	An Ethereum test-net that used to deploy my smart contract to. One of the main benefits was that it provides a fast transaction time for quick feedback. It was chosen of Goerli [16] as the faucet gave out 5 times as much Ether.
Alchemy [38]	Alchemy provides useful tools for interacting with Ethereum and specifically Sepolia, such as an ETH faucet and an RPC URL.
MetaMask [39]	A browser-based wallet that can easily be connected to other tools such as Alchemy or Remix. This tool can generate new accounts and show you the transaction history of each using an intuitive UI.
Remix [40]	A browser-based IDE for writing smart contracts that allows for easy deployment. I did have an implementation that would deploy using go-ethereum but due to a bug in package it was unable to work for the Sepolia test-net.
Ganache CLI [41]	Ganache CLI was used to create a local Ethereum test-net that I could develop my application with. With one command I could quickly boot a lightweight blockchain with a predetermined set of keys. When compared to Geth, from go-ethereum, Ganache CLI was much more beginner friendly.

Table 5.2: *The tools used for deployment of my smart contract*

The contract was successfully deployed [42] to the Sepolia test-net and can be interacted with by any user.

5.3 Frontend

The frontend code was developed from Section 4.3 to provide a user a GUI to interact with. Table 5.3 details the list of tools used.

Tool	**Description & Reasoning**
Wails [43]	Allows you to add a webkit frontend to a Go application, so that you can use a modern web framework. This allowed me to easily create a reactive UI using tools I was previously familiar with. Wails allows you to implement a controller using functions written in that can be called from the frontend and can emit events that trigger actions in the frontend.
Vue.js v3 [44]	A reactive, component based web-framework that allows me to create reusable components that react to changes in state and can trigger events at different points in a components lifecycle. The Vue Router [45] package was used to add multiple pages to the application and markdown-it [46] was used to render markdown files. Vue was chosen over other frameworks (like React or Angular) because it is my personal favourite and the one that I have had the most experience working with.

Pinia [47]	A state management tool for Vue.js that boosts the reusability of components and reduces the overall complexity of the frontend by allowing state to be accessed globally through stores. Pinia integrates directly with Vue's composition API and is very beginner friendly and well documented, especially when considered to similar tools from other frameworks.
SASS [48]	An extension of CSS that is used to style DOM elements. This was essential in making the UI look nice and be accessible without having styling that was hard to maintain and search through.

Table 5.3: *The tools used to develop the application's GUI*

5.4 Other Tools

Table 5.4 shows the other tools used for the design, development and write-up of this application.

Tool	Description & Reasoning
Git [49] GitHub [50]	A version control system used in conjunction with GitHub. Creating periodic commits meant I always had a recent backup available and could easily backtrack to help find issues. Use of a GitHub Actions helped remind me that not all of my tests passed at all times :(.
LaTeX [51]	Used for the write-up of this document. LaTeX was useful in creating a large document and has many packages that help with referencing and design.
VSCode [52]	My code editor of choice for this project as it allowed me to seamlessly work on both my frontend and backend code at once.
Lucidchart [53]	Lucidchart was used to create all of the diagrams for this project. Lucidchart offers a better user experience when compared to alternatives like Draw.io but locks several features behind a paywall.

Table 5.4: *General purpose tools used for this project*

Chapter 6

Testing

6.1 Overview

My approach to testing will consist of the following principles:

1. **Test Driven Development** [54] 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 running tests.
3. **Documentation** Test cases should be well documented and grouped contextually such that they are easy to maintain and add to. See Appendix **??** for an example of the documentation and structure expected for each function test.

Tools

In Table 6.1 I detail the different tools I used to write automated tests for my application.

Tool	**Description & Reasoning**
Go Testing	The testing package included with Go's standard library was sufficient to produce most of the test cases required for this project.
testify [55]	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 6.1: *The tools used for testing my project*

6.2 Unit Testing

Unit tests were written alongside the code they were testing to ensure my code was robust and responded appropriately to all inputs. Generic functions like getters or setters were ignored and I aimed for a 70% test coverage to ensure a significant amount of the application was tested. Table 6.2 shows a breakdown of test coverage by package.

Package	**Coverage**	**Tests Written**
model/manager/hashtree	75.1%	69
model/manager/games	61.4%	107
model/manager/ignore	90.6%	13
model/net/tcp	71.6%	27
model/net/peer	50.4%	111
model/persistence/ethereum	81.7%	20
model/util	61.2%	24
Total	70.3%	371

Table 6.2: *Code coverage by package. Missing entries do not have code in them for example model/manager is only a wrapper for its child packages.*

Automated tests were not written for the Controller functions as these were usually very simple and would only call a backend function and translate the output. Equally, UI components were not tested either due to the complexity of writing such tests.

6.3 Integration Testing

Profiles

A profile is used to mimic a specific type of peer that a user may find through this application. This is to allow me to test my application under different conditions and see how it reacts. Table 6.3 details the used profiles.

Name	**Purpose**
Listen Only	A peer who will listen and respond to all requests perfectly but will never request anything. This is useful for testing when we want a lightweight client to just download data off of. This will also be able to upload a game to the locally running smart contract.
Sender	This peer will respond to requests but will also periodically send requests. This can be used to show how my application reacts to more realistic peers.
Unreliable	This peer will pseudo-randomly not respond to messages or send incorrect data in response. This is used to show how my application recovers from faults sent by other users.
Selfish	This peer will send requests but will never respond to any. This is used to show how my application handles expired requests.

Table 6.3: *The different profiles used to simulate real-world peers.*

The main outcome from these different profiles shows the need for a reputation system where a user can distinguish between peers that reliably respond to requests and peers that don't. This would help mitigate some of the overhead of timed-out requests or receiving incorrect data.

6.4 Acceptance Testing

To ensure this application meets the requirements as described in Section 4.1.2, each requirement will be part of at least one user walkthrough. In Table 6.4 we give a high level description of each user walkthrough and any comments or results found from it. See Appendix **??** for a detailed example.

Id	Requirements	Description	Success
1	(**F-M1**) (**F-M5**) (**F-M12**) (**F-S2**) (**F-C2**) (**NF-M1**)	1. P_1 uploads a game G_1. 2. P_2 finds G_1 on the store. 3. P_2 purchases G_1. 4. P_2 shows G_1 added to their library.	YES
2	(**F-M6**) (**F-M8**) (**F-M9**) (**F-M10**) (**F-M11**) (**F-S1**) (**F-S2**) (**F-S3**) (**F-S4**) (**NF-M2**)	1. P_1 and P_2 own game G_1. 2. P_1 connects to P_2. 3. P_1 and P_2 exchange Ethereum addresses. 4. P_1 will get a list of P_2's peers and attempt to connect to them. 5. P_1 starts a download for G_1. 6. P_1 sends requests for blocks to P_2. 7. P_2 queries the smart contract to verify that P_1 owns G_1. 8. P_2 will respond to P_1 with the requested data. 9. P_1 will verify each block of data received using its hash. 10. P_1 will have full downloaded G_1. 11. P_2 will request a contributions receipt.	YES
3	(**F-M7**) (**F-M8**) (**NF-S1**)	Refer to the benchmark tests described in Section 6.5.	YES
4	(**F-M2**) (**F-M3**) (**F-M6**) (**NF-M5**)	1. P_1 is the original uploader of G_1. 2. P_1 uploads an update to G_1, G_2. 3. P_2 owns G_1 and thus G_2. 4. P_2 will successfully download G_2.	YES

Table 6.4: *The set of user walkthroughs used to prove the completeness of this project's requirements.*

6.5 Benchmarking

This project uses benchmarking to determine the overall performance and scalability of the application, whilst also being useful in discovering any bottlenecks or bugs.

To ensure consistency in these results, I've included the following constraints:

- All benchmarks should be ran on the same hardware using the same OS,
- All tests should be ran three times,
- Test data should be pseudo-random, and
- All projects should aim for the target size of 40GB to match the average game size given in Section 4.2.

All graphs made with Matplotlib [].

Number of Peers

For each run, we will create a project with 500 files, each of size 80MB and a shard size of 4MiB. We will then run N peers locally to simulate a perfect network connection.

	Runtime (s)			
Peers	**1**	**2**	**3**	**avg.**
1	1,513	1,511	1,511	1,512
2	777	775	777	776
4	397	397	396	397
8	227	223	224	225

Table 6.5: *Raw data from peer count benchmark*

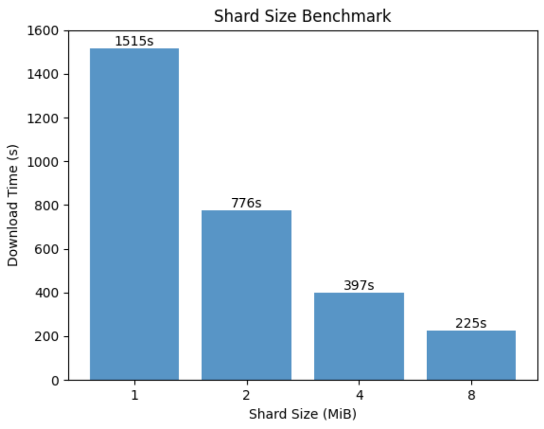

Table 6.6: *Graphical version of Table 6.5*

This result shows us that we can massively increase the download speed by increasing the number of peers we are connected to. This shows that my application can correctly handle many concurrent peers (**F-M7**) and uses the increased connections to boost performance (**NF-S1**).

Game Size

For each run, we will create a project of F files of size SMB, such that $F \times S = 40GB$. We will download the game off of 4 peers that are running locally on the same machine.

File		**Runtime (s)**			
Count	**Size (MB)**	**1**	**2**	**3**	**avg.**
200	200				
100	400				
50	800				
25	1,600				
5	8,000				

1	40,000				

Table 6.7: *How varying file count and size affects download speed*

From these results, we can see that distributing data across a larger pool of files results in a greater download speed. This occurred for several reasons:

1. A file can only be written to by one process at one time so having less files will reduce the potential for parallelisation.
2. For each block downloaded: we open a writer to a file, write a single block, and close the writer. This adds a lot of overhead for files that have many blocks.

Chapter 7

Project Management

7.1 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 decide on any changes my project might need. I could also use online documentation or forums for support.
Personal illness	3	2	6	Depending on the amount of lost time, I will have to choose to ignore some lower priority requirements. Use of effective sprint planning will help ensure I can produce at least a minimal viable product.
Laptop damaged or lost	3	1	3	Thorough use of version control and periodic backups to a separate drive will ensure I always have a relatively recent copy of my work. I have other devices available to me at home and through the university to continue development.
The application is not finished	2.5	4	10	Effective use of agile development and requirement prioritisation will ensure that even if I do not complete the project I will have the most significant parts of it developed. It is important to consider a cut off point for development, where I will have to purely focus on the write-up and final testing.
Lack of large-scale testing infrastructure	2	5	10	Local benchmarks can be used to determine theoretical upper limits on my application or could be tested using a variety of hardware owned personally. Other tests may show it working on a smaller scale over the internet but it would be difficult and expensive to obtain the hardware to test it at a large scale.

Table 7.1: *The risk assessment of this project*

7.2 Sprint Plans

Sprints were used in accordance of the Agile Methodology [56] to help me incrementally build on my project through prioritisation of the most important requirements.

Sprint 1

I anticipated that the P2P game distribution network would be the most complex and time consuming set of requirements in this project so I decided to focus on it for this first sprint. Table 7.2 shows the requirements included for Sprint 1 and whether they were completed or not.

Req.	**Complete**	**Evidence/Reasoning**
(**F-M7**)	YES	Unit tests for the model/net/tcp package and the peer count benchmark tests.
(**F-M8**)	YES	Unit tests for the model/net/peer/message_handlers file test the handling of structured messages and the structured responses sent back.
(**F-M9**)	YES	All benchmark tests show the downloading of data to a large scale.
(**F-M10**)	YES	Unit tests to show incorrect messages being rejected.
(**F-M11**)	YES	User walkthrough shows the download of a game in its entirety.
(**F-M12**)	STARTED	The algorithm to generate a hash tree and the using of it to download data was implemented but no way to upload it anywhere.
(**NF-M2**)	YES	User walkthrough 2 shows that any user can establish a connection with any other user.
(**NF-S1**)	STARTED	Users will form many connections concurrently and optimisations were made using the producer/consumer pattern to complete actions like inserting data, or requesting data.

Table 7.2: *Requirements included for Sprint 1*

Sprint 2

Sprint 2 was about increasing the scope of the application by focusing on two main aspects:

1. The integration with Ethereum using a Smart Contract, and
2. Allowing users to interface with the application via a GUI.

This sprint had a much slower start compared to the first one as I was largely unfamiliar with smart contract development and the related packages needed to interface with them. On top of this, I considered several UI framework's before settling on my final choice which increased the length of this sprint.

Table 7.3 shows the requirements pitched for Sprint 2 and whether or not they were completed.

Req.	Complete	Evidence/Reasoning
(**F-M1**)	YES	Unit tests for the Library smart contract and user walk-through 1 show the ability to upload game metadata to Ethereum.
(**F-M2**)	YES	Unit tests for the Library smart contract and user walk-through 4 show the ability to upload an update to an existing game to Ethereum.
(**F-M3**)	YES	Unit tests for the Library smart contract show users of an existing game being given ownership of an updated version.
(**F-M4**)	YES	The smart contract was successfully deployed the Sepolia test-net [42]. All user walkthroughs will form connections to this smart contract.
(**F-M5**)	YES	Unit tests for the Library smart contract and user walk-through 1 show the successful purchase of a game.
(**F-M6**)	YES	Unit tests for the Library smart contract show a user being added to a mapping containing all users who have purchased the game.
(**F-M12**)	YES	Hash trees are now uploaded to IPFS and the CID is stored on Ethereum.
(**F-S2**)	STARTED	Basic pages were added according to Section 4.3. These pages had little styling or reactivity but could perform the required basic functions. See Appendix **??** for screenshots of the final versions.
(**NF-M1**)	YES	The use of the Ethereum blockchain means that no single user can control what is uploaded to the network.
(**NF-M3**)	YES	Developers can be uniquely identified using their Ethereum address. This should be made publically verifiable by the developers.
(**NF-M4**)	YES	Data stored on Ethereum is inherently immutable.
(**NF-M5**)	YES	Unit tests for the smart contract show the restriction that only the original uploader can release an update.

Table 7.3: *Requirements included for Sprint 2*

Sprint 3

This sprint was about extending the minimum viable application reached by the end of Sprint 2 with some necessary additions. Table 7.4 shows the list of requirements for this sprint and whether or not they were completed.

Req.	Complete	Evidence/Reasoning
(**F-S1**)	YES	Users will validate each other's Ethereum address after forming a connection and unit tests for the model/net/peer/message_handlers file show this being performed.
(**F-S2**)	YES	The UI was overall improved to improve the user experience.

(**F-S3**)	NO	Users will track the blocks sent to them by each of their peers but this application has no mechanism for redeeming these. Due to time constraints, I was unable to implement a sufficient solution. Moreover, I felt that a micro-payment system, like present in Swam [28], would be a much better implementation.
(**F-S4**)	YES	Users will exchange the REQ_PEERS/PEER commands to discover neighbouring peers. However a better implementation might have the developer of the game be able to provide a list of peers who have the game. This would allow a user to easily find peers who are interested in the same content.
(**F-C1**)	NO	Due to time constraints I was unable to implement this at all.
(**F-C2**)	YES	Game assets are uploaded to IPFS and the CID is stored with the game metadata on Ethereum.
(**NF-S1**)	YES	Benchmark tests show the scalability of my application by varying certain parameters and that the target file size can be downloaded within an acceptable best-case.
(**NF-S2**)	YES	Changes to the UI made it more interactive and easier to navigate. Designs were inspired by pages from existing platforms to make the UI feel familiar. See Appendix **??** for screenshots of the final versions.
(**NF-C1**)	NO	Completing this requirement would be incredibly complex and was decided against being completed. Preventing the distribution of illegal content is an important consideration moving forward to help keep the platform safe.
(**NF-C2**)	YES	A help page was included answering some questions that new users may have about the application.

Table 7.4: *Requirements included for Sprint 3*

7.3 Gantt Chart

A Gantt chart was used to give a high-level overview of my project to give myself a realistic timetable of when different aspects had to be completed by. Figure 7.1 shows the complete version.

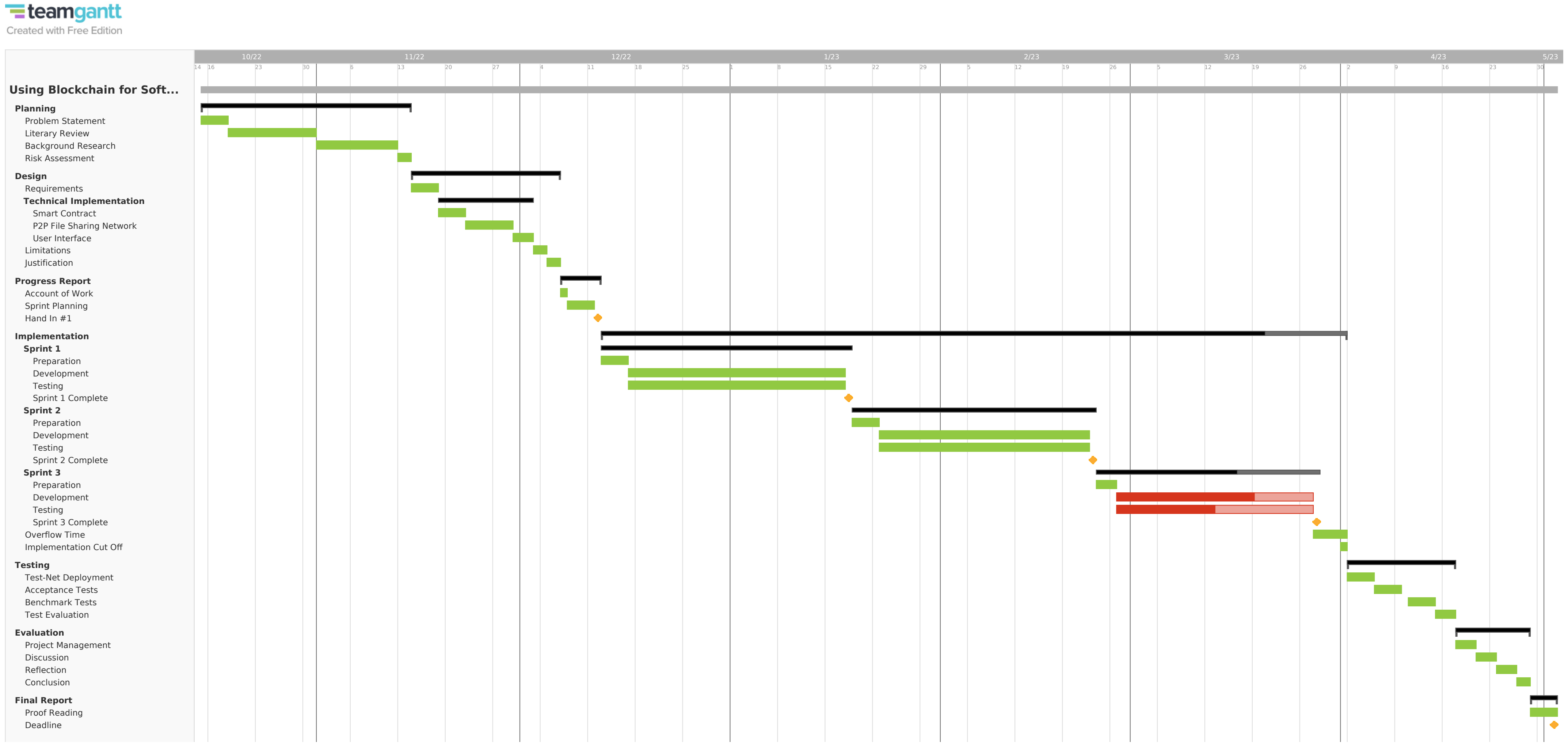

Figure 7.1: *The Gantt chart showing a breakdown of my project by task and milestones, where the red areas incidicate incomplete work. Created with TeamGantt [].*

Chapter 8

Evaluation

8.1 Discussion

Due to the scale and complexity of the platforms this project was aiming to replace, this project was never going to be more than a proof-of-concept as to what a distributed games marketplace could look like. However, this application does have several benefits over its centralised competitors, which are outlined in Section 4.4

This project presents interesting ideas surrounding how games, and other proprietary software, can be distributed without the need for a middle-man. This follows modern web ideas of taking away reliance on large data-centres and having a system built and maintained by the people who use it. Moreover, this type of project is usually open source, which can help attract community contributions to improve the security and user experience of the application.

8.2 Reflection

Risk Assessment

Difficulty with blockchain development As expected there were several difficulties with blockchain development:

- A decent amount of the documentation is minimal or outdated. As blockchain development is a smaller field there are little in the ways of forum postings.
- Some of the libraries used had bugs in them that hindered development. For example, I could deploy the smart contract to the Sepolia testnet from Remix easily but not through Geth.

Despite the problems mentioned above, I was able to complete the blockchain aspect of the project successfully. Effective use of sprint planning meant that I had considered this to be a potential issue and had allocated time to being able to deal with it.

The application is not finished Section 7.2 shows that not all of the requirements were finished and Section 4.5 discusses the limitations with my current implementation. It was expected that this project would not be entirely finished but a lot of the major requirements were met.

The application ended up being incredibly large and took a lot more time to implement and test than expected. Table 8.1 shows a breakdown of the source code. Agile development could only help me so much so by having a hard cut-off point for the implementation I was still left with plenty of time to complete the report.

Language	**Files**	**Comments**	**Code**
Go	45	826	5087
Go Tests	29	791	3205
Vue.js Components	18	147	2855
JavaScript	19	74	209
Solidity	1	30	51

Table 8.1: *The lines of code written for this project calculated using CLOC [57]*

Lack of large-scale testing infrastructure Testing distributed applications is challenging due to many factors, such as having to source homogenous devices, having access to networks that would allow me host a peer, and more. However, several useful results were produced from the benchmarks (Section 6.5) and acceptance tests (Section 6.4) that can be used to improve the application. If this project were to move forward then testing it on a large-scale would be essential.

Using Agile Development

The use of an agile methodology alongside test-driven development meant I could incrementally expand the scope of my application and ensure that existing code was tested sufficiently. Separating my implementation into three sprints benefitted me by:

- having a smaller set of requirements to focus on at once helped me to feel less overwhelmed,
- working on the most important aspects first to ensure I was able to produce a minimum viable product, and
- taking time in-between sprints to take a break from the project and prepare for the next sprint.

On top of this, using a Gantt chart (Figure 7.1) allowed me to have a clear overview of my project timeline so I could realistically gage just how much time I would need for each section.

Chapter 9

Conclusion

9.1 In Conclusion

This project set out to demonstrate how video game distribution could be migrated to a distributed platform with the aim of reducing the risk of censorship, improving ownership and increasing profits for developers.

By researching related topics and reviewing the literature around key areas of this project, I was able to combine many modern ideas and techniques to develop a functional proof-of-concept application. The heavy use of automated testing allowed me to continuously write robust and correct code.

As most of the requirements set out in Section 4.1.2 were met, I can say that this project was successful in providing a proof-of-concept decentralised video game marketplace.

9.2 Future Work

Content Discovery

Creating a fully functioning store page, where users can search for and discover new games, would be a vital next step for this application. Some of the techniques used to achieve this could be:

- **Indexing** Periodically generate an index of all games uploaded that allows users to easily search the store without having to make large amounts of requests to the blockchain. Use of ranking algorithms could allow for users to be shown the most relevant and useful results.
- **More Metadata** Adding more metadata to games would allow for them to be more easily searchable and indexable. For example, each game might be given a set of tags that can be searched for.

This would help flesh out our store page to make the user experience even better as user's will be able to search for games through this application.

New Features

There are some important features that were deliberately left out of scope that would be excellent extensions to this project. These include:

- Game reviews and message boards to allow for greater community integration,

- A fully functioning store page,
- Greater personalisation to the application, and
- Searchable and customisable user profiles.

Optimisations

There are several optimisations we could add to improve the overall performance and scalability of the application:

- **New Commands** By extending the commands from Section 4.3 we could reduce the amount of network overhead required to communicate between peers.
  This could include batch block requests, or only forming connections with peers who we share some games with.
- **Block Selection** Currently blocks are not ranked in any way but by considering ideas from Section **??** we could improve the efficiency, availability and throughput of our network.