Chapter 1

Introduction

Millions of worldwide users enjoy video games, which are large pieces of software that require complex platforms to distribute them, which results in them being generally provided by multinational corporations. However this approach often results in these platforms:

- 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].

1.1 Objectives

Modern web ideas revolve around taking power away from large corporations and having platforms that are built, run and maintained by those who use them. This project aims to produce a proof-of-concept, distributed video game marketplace that will allow developers to continuously release and update their games on a public network such that they can be purchased and downloaded by other users.

1.2 Scope

This project will strictly focus on creating a distributed platform in which users can upload, purchase and share games and 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. This will then be deployed to an Ethereum test-net.
2. A local application to be run by users to interact with the smart contract and allow them to join a peer-to-peer network where they can download and upload games.

This project will not present methods for preventing or stopping the distribution of illegal content or include tangentially related features such as achievements, or message boards.

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

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

[3] The Nintendo eShop closing down prevents users from accessing games [4]

Chapter 2

Background Research

This chapter will consider two popular technologies from Chapter 3 to provide key ideas on how I could use ideas from them to build this project.

2.1 BitTorrent

Without centralised architecture, data will need to be distributed among users where a user will only have the data they are interested in. This section will discuss the popular P2P file-sharing protocol BitTorrent and Section 3.1 will look at other examples

BitTorrent [5, 6] is a protocol for sharing data across a distributed network[1] and is one of the most popular P2P file-sharing protocols active today, being responsible for 3.35% of global bandwidth [10]. As such, it is important to understand what makes BitTorrent so successful and how I could include key ideas in my project. 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 pieces that have the fewest copies to boost availability.
3. **As Soon As Possible** A user with no pieces will try to get a random piece quickly so they can contribute to the network.

Optimistic Unchoking A peer allocates a portion of their bandwidth for communicating with unknown peers. This allows new users to join the network and be able to contribute without being ignored and gives a way for existing peers to seek better peers.

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

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

Blockchain technology will allow for the coordination of a large, decentralised set of users whilst keeping the system transparent, secure and auditable without the need for any central party. The built-in consensus mechanisms are designed to make it difficult for users to act in a way that negatively affects the network.

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 their copy of the global 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, 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 users will offer a higher gas price to have their transaction be 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 [17].

Chapter 3

Literature Review

This section will be used to examine the literature surrounding the distributed storage of data. Initially I will look at various distributed file-sharing protocols and then how blockchain has been used to enhance them.

3.1 P2P File Sharing

One finding from the previous section was that a lot of blockchain-based cloud storage systems rely on an external form of distributed file-sharing. Therefore it is important for me to understand the existing options, their architecture, as well as their pros and cons to determine the best implementation for this project.

Table 3.1 looks at various implementations of P2P file-sharing networks.

System	Description of Solution
IPFS [18]	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 [19] or Kubo [20].
Swarm [21]	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 [22, 23]	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 [24]	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. This protocol is dependant on a central cluster existing and will not function properly without it.

Gnutella [24]	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. A network flood in a large network will result in a significant amount of congestion and and a large amount of *pong* messages going to a single node results in a DDoS attack.

Table 3.1: *Various global distributed file systems.*

Some of the common themes found include:

- **Trust** Nodes are typically anonymous so users have to trust that the data they download isn't malicious.
- **Illegal Content** There are no measures in place to stop or prevent the distribution of illegal content across these networks.
- **Payment** None of these networks natively support payment and thus don't support the access control systems required for payment systems.

It is clear that using blockchain in conjunction with these systems could mitigate a lot of their issues and thus a combination of these two ideas will be used for this project.

3.2 Blockchain-Based Cloud Storage

Table 3.2 shows examples of how blockchain has been used to provide secure, decentralised cloud storage platforms. This is to look at how large-scale data storage has been achieved using blockchain technology and give me examples on how I could apply it to this project.

Paper	Description of Solution
Blockchain Based Data Integrity Verification in P2P Cloud Storage [25]	The Ethereum blockchain is used to add trust to a data verification system for a P2P network. It analyses how varying structures of Merkle Trees affect the performance of verification of data stored in the network.
Deduplication with Blockchain for Secure Cloud Storage [26]	This paper implements a deduplication scheme by uploading storage information to Ethereum and uses smart contract based protocols to provide secure deduplication over encrypted data.
Block-secure: Blockchain based scheme for secure P2P cloud storage [27]	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 [28]	Describes a blockchain-based platform that would allow for the secure and immutable storage of user medical records, such that they are independant from any individual medical institution and users have greater control over who can access their personal data.

A Blockchain-Based Framework for Data Sharing With Fine-Grained Access Control in Decentralized Storage Systems [29]	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 an Ethereum smart contract is used to implement a keyword search of the data stored.
Blockchain based Proxy Re-Encryption Scheme for Secure IoT Data Sharing [30]	A 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 3.2: *Examples of blockchain cloud storage systems [31]*

Some common themes found include:

- **Access Control** If these platforms offer access control it is typically up for the owner of the data to distribute access rights.
- **Data Integrity** The blockchain is primarily used to add trust across a distributed network in the form of immutability or providing data integrity services.
- **Data Location** Data stored on the blockchain is commonly metadata that refers to some data stored elsewhere, typically on a form of distributed storage network.

Chapter 4

Design

This section will give a detailed design for this project by first considering the requirements needed to meet this project's goals, then giving a high-level design of how these requirements could be met. This will then be translated into a specific application architecture to be implemented.

4.1 Analysis

This section will look at the key stakeholders for this project and use them to come up with a list of requirements that will, if completed, prove that this project meets the goals it set out in Section 1

4.1.1 Stakeholders

Stakeholders allow us to understand the different types of user by looking at what they will use the platform for, as well as what they want and don't want from it.

Game Developers PRIMARY
Users who upload (and update) games to the platform that will be purchased by other users who want access to it. As the content providers for this platofrm, they are essential in attracting users.

Players PRIMARY
This group will purchase games, then upload and download game data to other other users in the network. They will be responsible for the long-term availability of games and making the platform appealing to developers.

Other Platforms SECONDARY
Other digital marketplaces that serve as the main competitors to this platform. Users will expect a similar experience as found on these other platforms so this project will share several key ideas from them.

4.1.2 Requirements

To determine the requirements of this project, I used a mind-map to show how the primary stakeholders, key pieces of data, and predicted technologies would relate to one another and what constraints would be on those relations. This can be seen in Appendix **??**.

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.
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[1] 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.

[1] See Section 4.2.3.

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 publicly 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.
	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	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

This section will give a high-level design showing how each of the functional requirements can be met by considering key functions for the applications.

4.2.1 Data Types

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	Blockchain	The minimal set of information required for the unique identification of each game. See Section 4.2.2. This data is appropriate to store on the blockchain 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	The hash tree that will allow users to identify and verify 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 makes it costly to store on the blockchain at a large scale. IPFS will be used for fast, reliable access at a large scale and we can store the generated CID in the blockchain instead.
Game Assets (**F-C2**)	Variable[2]	IPFS	Any promotional material provided for the game that can be viewed on the game's store page. This should include cover art and a description file but 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 the blockchain instead.
Game Data	*avg. 44GB* [3]	Peers	the data required to run the game that is 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 the blockchain or IPFS. Therefore, this project will use a custom P2P network for sharing data, which is described in Section 4.2.3.

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

4.2.2 Blockchain

This section will describe how blockchain technology will enable the storage of game metadata and the purchasing of content using a distributed immutable record that can be trusted by any user.

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

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

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

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.

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	A unique fingerprint that identifies the game.
previous version	The root hash of the most previous version of the game if it exists.
next version	The root hash of next update to this 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 of all users who have purchased the game (**F-M6**).

4.2.3 Distributed File Sharing

This section will look at how various types of game data can be shared across a distributed set of peers using a content-addressable hash tree to describe the data.

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**). Users should be able to specify which folders/files to exclude when generating a hash tree.

Hash trees will be stored on IPFS with the CID being kept with the other game metadata on Ethereum.

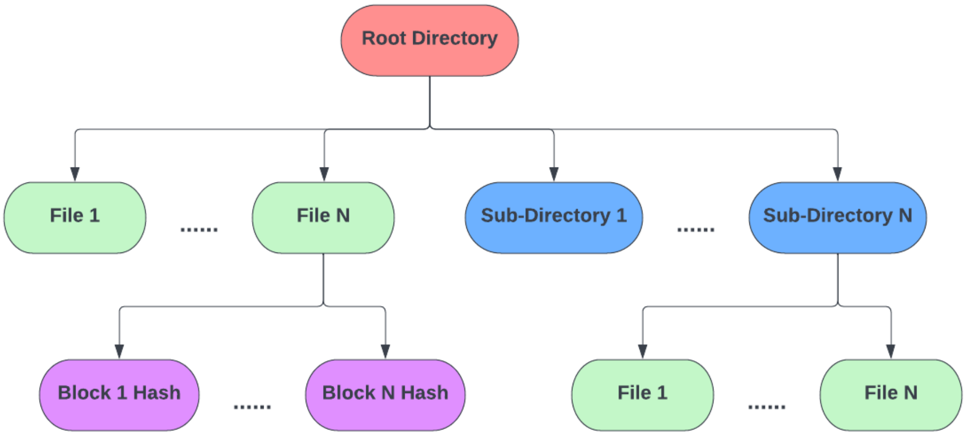

Figure 4.1: *The structure of a hash tree*

Downloading Content

Using the hash tree, a user will select blocks of data and send requests for them to 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. $P_{downloader}$ will send requests for individual blocks to P_{seeder} (**F-M9**).
3. Upon receiving the first request, P_{seeder} will verify that $P_{downloader}$ owns the game by checking the *purchased* mapping on the smart contract (**F-M6**) (**F-S1**).
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 and send requests to many peers at once (**F-M7**). Requests will be sent in a round-robin fashion to evenly distribute the requests and prevent overloading a single peer (**NF-S1**). Incomplete requests will be retried periodically or when connecting to a new peer.

Updating Content

To satisfy (**F-M2**), developers can upload updated versions of their games by providing the root hash of the most previous version of the game, where only the original uploader will be allowed to publish an update to an existing game (**NF-M5**). Any users who own a previous version of a game will be allowed access to all future versions (**F-M3**).

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 section will detail the architecture and implementation details for an application based upon the design considerations given in the previous section.

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.

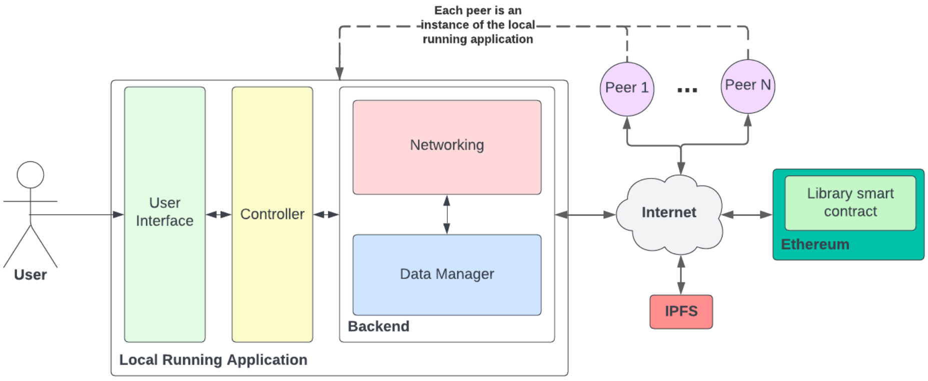

Figure 4.2: *The components of the application*

4.3.1 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 4.2.1.

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 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.
- 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.

A peer's behaviour is described by the Backend in Section 4.3.2, which will be part of the application run locally by the user.

4.3.2 Backend

The Backend containts all functionality required for a peer to use and contribute to the platform and 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 locating and sharing data. See Section 4.3.2.

- **Data Manager** The management of local data and the processing of data received and to be uploaded by the Networking component. See Section 4.3.2.

Both sections will interface with Ethereum as appropriate to complete their functionality.

Networking

This component will connect users to the distributed P2P file-sharing network, where it will create and maintain 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 4.3.2 describes these commands in detail.

Peer Identification Peers are identified by their Ethereum addresses, which will allow us to view which games they've purchased and are allowed access to. Upon forming a connection, each peer will request the other return a signature for a generated message, from which we can derive their address and public key.

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 by 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**) and if it isn't received after a given amount of time then it is resent.

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**).
SERVER;[$hostname$] : [$port$]	When we form a connection with a peer we send them the address of our server that peers use to connect to us. This allows that peer to share our address through the PEER command so we are more easily discoverable.
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

This component is responsible for interacting with local storage and managing the user's collection of owned and installed games. It will interact directly with Ethereum to discover, purchase (**F-M5**), and upload (**F-M1**) (**F-M2**) games and use IPFS to upload and distribute game assets (**F-C2**) and hash trees (**F-M12**).

Download Order Each download will have a downloader thread that selects the order at which blocks are to be fetched. Using the hash tree, it will queue whole files at a time and to verify the whole file once all blocks have been fetched.

4.3.3 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. Users should be able to view both old and new versions of a game and check for any new updates.
- **Store** Where user's can find and purchase new games that have been uploaded by other users. Games should be searchable by their root hash.
- **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 contribution data from their peers.
- **Help** (**NF-C1**) Contains some questions and answers about the application.

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 Sequence Diagram

Figure 4.3 is a sequence diagram that shows a game being uploaded to the platform and then being purchased and downloaded by another user.

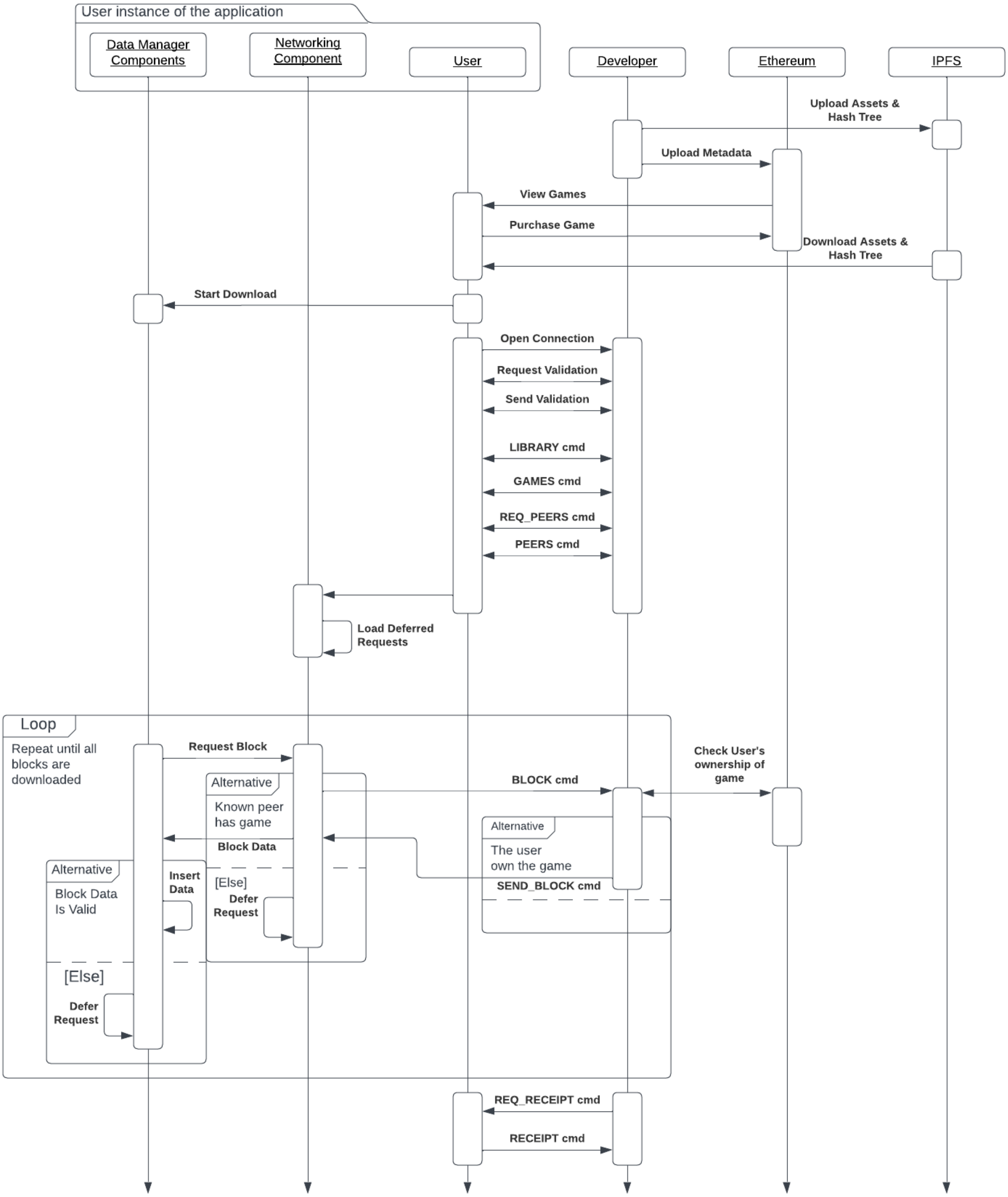

Figure 4.3: *A sequence diagram showing the main interactions needed to download a game from a single peer*

4.5 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.
- **Concurrent Downloads** Many downloads can be completed in parallel whereas a centralised platform would do them sequentially.

4.6 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[4].
- **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.
- **Illegal Content** Currently there is no policy, automated or otherwise, to stop the distribution of illegal content. This is an incredibly complex problem to tackle and any obvious solution would risk violating the anti-censorship goal of this project.

[4] An example would be how the micro-payment system works in Swarm [21]

Chapter 5

Implementation

The implementation was completed according to the design specified in Chapter 4 using the collection of tools specified below. See Section 7.2 for a breakdown of how sprints were used to structure the implementation.

5.1 Backend

Table 5.1 shows the tools to create the backend of my application as it was described in Section 4.3.2.

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 [20] 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 4.2.2.

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 4.3.3 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 due to me already having lots of experience using it.

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.
Zotero [54]	A reference manager used to help me organise all of the documents cited in this paper. This includes a browser extension to easily add new references.

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

CHAPTER 6

Testing

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 running tests.
3. **Documentation** Test cases should be well documented and grouped contextually such that they are easy to maintain and extend.

Tools

Table 6.1 shows the different tools used to write automated tests for my application.

Tool	Description & Reasoning
Go Testing [55]	The testing package included with Go's standard library was sufficient to produce most of the test cases required for this project.
testify [56]	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*

6.1 Unit Testing

Unit tests were written alongside the code they were testing to ensure my code was robust and responded appropriately to all inputs. 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	75.2%	69

model/manager/games	61.3%	107
model/manager/ignore	90.6%	13
model/net/tcp	71.9%	27
model/net/peer	55.8%	111
model/persistence/ethereum	81.7%	20
model/util	61.2%	24
Total	71.1%	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 also written for the smart contract functions and tested locally using Ganache CLI [41]. Tests were not written for generic functions like getters and setts; this includes the Controller functions that were typically wrappers around Model functions. Equally, automated UI tests were omitted due to time constraints and that UI components were partially tested by the user walkthroughs.

6.2 Integration Testing

The purpose of integration tests were to evaluate how the application fared when interacting with various types of peer. Each *profile* will mimic a type of behaviour that could be expected in a real-world deployment and the different types are detailed in Table 6.3.

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.

6.3 Benchmarking

To assert that my application is scalable (**NF-S1**), I will be using a benchmark to show the download speed of a game varies depending on the number of peers a user is connected to.

Each test case will:

- connect to N **Listen Only** peers.
- dowload the entirety of the generated test data. This data will be of size 40GB to match the average game size mentioned in Section 4.3.
- be run three times to ensure consistency.

Table 6.4 shows the results of the benchmark.

	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 6.4: *Download times for the same piece of data by varying the number of connected peers.*

These results show that the download speed we can massively be increasedbby connecting to more peers. This shows that my application can correctly handle many concurrent peers (**F-M7**) and will distribute requests among many different connections to increase scalability (**NF-S1**).

6.4 Acceptance Testing

A user walkthrough is a series of steps to take that, if completed, prove the completeness of a set of requirements. All user walkthroughs use a contract deployed to the Sepolia test-net [42], satisfying (**F-M4**), (**NF-M2**) and (**NF-M1**), and all transactions completed can be seen publicly.

Table 6.5 describes all user walkthroughs and Appendix **??** shows the evidence for their completeness.

Id	**Requirements**	**Description**	**Success**
1	(**F-M1**) (**F-M5**) (**F-M12**) (**F-S2**) (**F-C2**) (**NF-M3**)	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**) (**NF-M2**)	1. P_2 connects to P_1. 2. P_1 and P_2 exchange Ethereum addresses. 3. P_2 starts a download for G_1. 4. P_2 sends requests for blocks to P_1. 5. P_1 queries the smart contract to verify that P_2 owns G_1. 6. P_1 will respond to P_2 with the requested data. 7. P_2 will verify each block of data received using its hash. 8. P_2 will have full downloaded G_1. 9. P_1 will request and receive a contributions receipt for G_1 from P_2.	YES
3	(**F-M2**) (**F-M3**) (**F-M6**) (**NF-M5**)	1. P_1 is the original uploader of G_1 and P_2 has already purchased G_1. 2. P_1 uploads an update to G_1, G_2. 3. P_2 will hit the 'check for updates' button and see G_2 in their library.	YES
4	(**F-S4**) (**F-M7**) (**NF-M2**)	1. P_1 is connected to P_2. 2. P_3 forms a connection with P_1. 3. P_3 requests a list of P_1's peers and P_1 responds with the details for P_2. 4. P_3 forms connections with P_2.	YES

Table 6.5: *The set of user walkthroughs used to prove the completeness of this project's requirements.*

Chapter 7

Project Management

7.1 Risks

This section will describe any anticipated risks for this project that were considered at the start of the project and then discuss whether any occurred and how effective my mitigation strategies were.

Risk Assessment

Table 7.1 shows the risks that were considered at the start of this project.

Risk	Loss	Prob	Risk	Mitigation
Difficulty with blockchain development	2	3	6	I will dedicate time before implementing the blockchain section to learning the necessary tools using online guides, documentation, and forums to ensure I am able to complete it.
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*

Risk Evaluation

Table 7.2 looks at which risks occurred during the this project and how effective my mitigation stratgeies were.

Risk	**Explanation**
The application is not finished	Several requirements were not met, as shown in Section 7.5, and this can be attributed to the project simply being too large for the time frame I had. However, this risk was highly anticipated and by using sprints I was able to focus my attention to the most important requirements and ensure that the application I had was still successful in terms of the goals of this project.
Lack of large-scale testing infrastructure	The lack of devices available to me made this application difficult to test in a *real* environment but the mitigations suggested allowed me to prove that my application worked correctly in a test-network and would can scale effectively.
Difficulty with blockchain development	Being new to this field meant I faced several setback when working with blockchain technology. However by dedicating time to learning and practicing before incorporating it into my project, I was able to successfully complete that section.

Table 7.2: *The risks which occurred during this project.*

7.2 Sprint Plans

By dividing my implementation into sprints, I was able to incrementally build upon my project by completing requirements according to their priority and expected difficulty. Separating my implementation into 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.

The following sections will detail each of my sprints, including a breakdown of the requirememnts that were and weren't complete.

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 7.3 shows the requirements included for Sprint 1.

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	The benchmark test 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 2 shows the download of a game in its entirety.
(**F-M12**)	STARTED	The algorithm to generate hash trees and the ability to use them to download data was implemented. Unit tests for the *model/manager/hashtree* and *model/manager/games* packages show this. Uploading this to distributed storage was planned for Sprint 2.
(**NF-M2**)	YES	User walkthrough 2 shows that any user can establish a connection with any other user.
(**NF-S1**)	STARTED	Users can perform many concurrent connections and channels are used at component boundaries to allow for multiple producers/consumers of data. This requirement was a consideration throughout all sprints.

Table 7.3: *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 7.4 shows the requirements pitched for Sprint 2.

Req.	**Complete**	**Evidence/Reasoning**

(F-M1)	YES	Unit tests for the Library smart contract and user walkthrough 1 show the ability to upload game metadata to Ethereum.
(F-M2)	YES	Unit tests for the Library smart contract and user walkthrough 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 testnet [42]. All user walkthroughs will form connections to this smart contract.
(F-M5)	YES	Unit tests for the Library smart contract and user walkthrough 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 4.3.3. 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 Library smart contract show the restriction that only the original uploader can release an update.

Table 7.4: *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 7.5 shows the list of requirements for this sprint.

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 [21], 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**)	YES	A help page was included answering some questions that new users may have about the application.

Table 7.5: *Requirements included for Sprint 3*

7.3 Gantt Chart

A Gantt chart was used to give a visual, 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.

7.4 Comparison to Interim

Changes were made to the plan, submitted in the interim report (Appendix **??**), early in the implementation to reflect a more realistic timetable for myself. These include:

1. The testing phase was used to focus on deployment and acceptance testing through user walkthroughs and a benchmark.
2. Sprints were extended and sprint reviews were removed to allow me more time to focus on the implementation itself and ensure my code was tested. This included an overflow period and a cut-off where extra time was anticipated.
3. The evaluation period was changed to consider a wider range of topics.

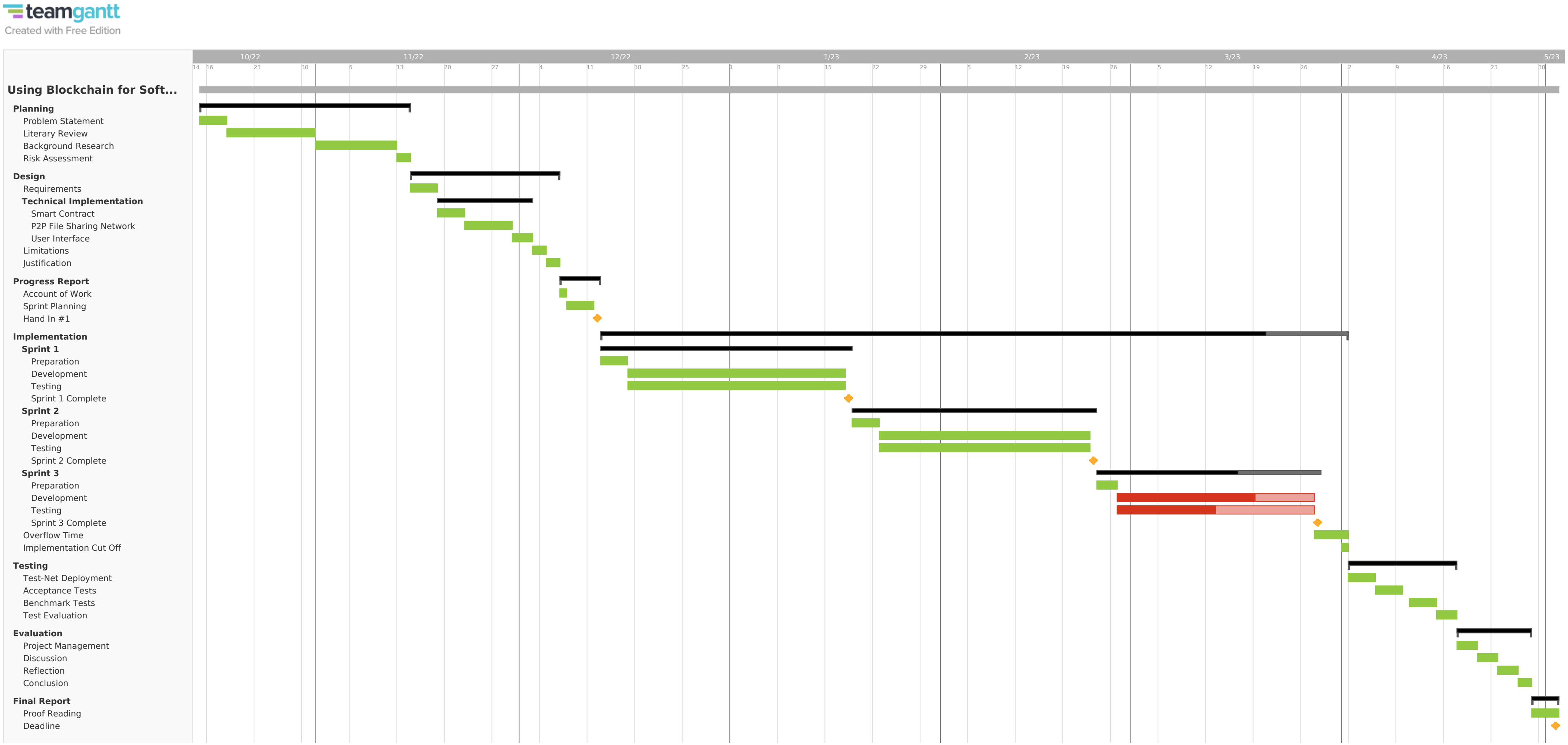

Figure 7.1: *The Gantt chart showing a breakdown of my project by task and milestones, where the red areas incidicate incomplete work. Created with TeamGantt [57].*

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 4.5

This project presents interesting ideas surrounding how games, and other proprietary software, can be distributed between its users without the need for an intermediary. Furthermore, making this project open source would encourage community development through updates or extensions, and improve transparency.

8.2 Reflection

As a newcomer to blockchain technology, I am pleased with what I've been able to achieve over the course of this project. I have genuinely enjoyed working on it and have had to stop myself from adding in new features after the implementation cut off.

However, this project was very large and took months of daily programming to get it to this stage, which left me very burnt out going into my final semester. If I were to redo this project I would reduce the scope to make it more manageable

Chapter 9

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 modern ideas and technologies to develop a functional proof-of-concept application. The heavy use of automated testing allowed me to continuously write robust and correct code and the successful deployment to an Ethereum development meant I was able to demonstrate the correctness of my implementation.

As most of the requirements set out in Section 4.1.2 were met, I can say that this project was a success.

9.1 Future Work

New Features

Some features that I would priortise if I were to continue would be:

- Fleshing out the store page will make it easier for users to discover new content. Games could be given extra metadata to help make them more searchable.
- Allowing users to post reviews or posts to message boards for individual games will allow for greater community integration.
- Add customisable user profiles and allow for users to add each other as friends. Friends would auto-connect as peers to help maintain a network.

Optimisations

I would also like to work on several optimisations, that includes:

- **Peer Reputation** By ranking peers based upon their reliability we can improve the success rate and latency of requests we send. An optimistic unchoking algorithm like seen in BitTorrent could also be used.
- **Block Selection** Currently blocks are not ranked in any way but by considering ideas from Section 2.1 we could improve the efficiency, availability and throughput of our network.