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

Thomas Smith, tcs1g20
March 29, 2023

Using Blockchain for Video Game Distribution

Project Supervisor: Leonardo Aniello
Second Examiner: Heather Packer

A project report submitted for the award of
BSc Computer Science

Abstract

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

Statement of Originality

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

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

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

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

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

I have not used any resources produced by anyone else.

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

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

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

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

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

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

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

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

Acknowledgements

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

Contents

Chapter 1

Problem Statement

1.1 The Problem

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

(a) taking a large cut of all revenue,
Steam take a 30% cut [12, 3]
(b) being vulnerable to censorship from governments,
*The Chinese version of Steam is heavily censored [**?**]*
(c) the user's access to their games is linked to the platform.
If the platform shuts down, the user loses all their games

1.2 Goals

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

1.3 Scope

This project will be broken down into two distinct components:

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

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

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

Chapter 2

Background Research

2.1 BitTorrent

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

BitTorrent [8, 15] was chosen as part of my background research as it is one of the most popular P2P file-sharing platforms. In 2013 it was estimated that tens of millions of users used BitTorrent every day [18]. In BitTorrent, users barter for chunks of data by downloading and uploading them in a tit-for-tat fashion, such that peers with a high upload rate will typically also have a high download rate.

Download Protocol

For a user to download data from BitTorrent they would:

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

Availability

One of the most significant issues facing BitTorrent is the availability of torrents, where *'38% of torrents become unavailable in the first month'* [8] and that *'the majority of users disconnect from the network within a few hours after the download has finished'* [15]. This

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

paper [14] looks at how the use of multiple trackers for the same content and DHTs can be used to boost availability.

2.2 Ethereum

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

Smart Contracts

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

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

Example Use Cases

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

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

Chapter 3

Literature Review

3.1 Blockchain-Based Cloud Storage

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

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

Paper	**Description of Solution**
Blockchain Based Data Integrity Verification in P2P Cloud Storage [20]	This paper uses Merkle trees to help verify the integrity of data within a P2P blockchain cloud storage network. It also looks at how different structures of Merkle trees effect the performance of the system.
Deduplication with Blockchain for Secure Cloud Storage [10]	This paper describes a deduplication scheme that uses the blockchain to record storage information and distribute files to multiple servers. This is implemented as a set of smart contracts.
Block-secure: Blockchain based scheme for secure P2P cloud storage [9]	A distributed cloud system in which users divide their own data into encrypted chunks and upload those chunks randomly into the blockchain, P2P network.
Blockchain-Based Medical Records Secure Storage and Medical Service Framework [4]	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 [19]	This solution uses IPFS, Ethereum and ABE technology to provide distributed cloud storage with an access rights management system using secret keys distributed by the data owner.

Blockchain based Proxy Re-Encryption Scheme for Secure IoT Data Sharing [11]	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 [17]*

3.2 P2P File Sharing

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

The main issues involving these networks are:

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

System	Description of Solution
IPFS [2]	IPFS is a set of protocols for transferring and organising data over a content-addressable, peer-to-peer network. Data uploaded to an IPFS network is addressed using it content identfier CID, which is a cryptographic hash based upon its content. IPFS is open source and has many different implementations, such as Estuary or Kubo.
BitTorrent [15]	BitTorrent is a p2p file-sharing system that has user bartering for chunks of data in a tit-for-tat fashion, which provides incentive for users to contribute to the network. More on BitTorrent can be found in Section 2.1.
Swarm [5]	Swarm is a distributed storage solution linked with Ethereum that has many similarities with IPFS [15]. It uses an incentive mechanism, Swap (Swarm Accounting Protocol), that keeps track of data sent and received by each node in the network and then the payment owed for their contribution.
AFS [13, 6]	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 [16]	Napster uses a cluster of centralized servers to maintain an index of every file currently available and which peers have access to it. A node will maintain a connection to this central server and will query it to find files; the server responds with a list of peers and their bandwidth and the node will form a connection with one or many of them and download the data.

Gnutella [16]	Gnutella nodes form an overlay network by sending *ping-pong* messages. When a node sends a *ping* message to their peers, each of them replies with a *pong* message and the *ping* is forwarded to their peers. To download a file, a node will flood a message to its neighbors, who will check if they have and return a message saying so; regardless, the node will continue to flood their request till they find a suitable node to download off of.

Table 3.2: *Various global distributed file systems.*

Chapter 4

Design

4.1 Stakeholders & Requirements

Stakeholders

Game Developers PRIMARY

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

Players PRIMARY

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

Other Platforms SECONDARY

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

Requirements

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

Functional Requirements

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

F-M6	A node must be able to download games in their entirety from nodes in the network
F-M7	A node must be able to verify the integrity of each block it downloads
F-M8	The application should run on the Ethereum network
F-M9	Users must be able to purchase games from developers over the network
F-M10	Users must be able to prove they have purchased a game
F-M11	A user must be able to create and maintain many concurrent TCP connections
F-M12	A user must be able to generate a Hash Tree of a game that describes its contents
	Should
F-S1	Seeders should have a way to prove how much data they have seeded
F-S2	Seeders will only upload content to users who have a valid proof of purchase
F-S3	Allow for the distribution of Downloadable Content (DLC) for games
	Could
F-C1	Allow users to request specific game versions
F-C2	Provide a simple GUI for interacting with the blockchain

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

Non-Functional Requirements

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

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

4.2 Design Considerations

Data

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

Data	Size	Location	Explanation
Game Metadata	100 – 200B	Ethereum	This data is the minimal set of information required for the unique identification of each game. See Section 4.2.1. 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	$\tilde{1}$5KB	IPFS	This will be the compressed Hash Tree that will allow the users to identify and verify the shards of data they need to download for their game. This data would be costly to store on Ethereum for a large number of games and will only need to be accessed by a subset of users. As it is also public data, IPFS is appropriate to store it on, and we can reference the CID within the data stored on Ethereum.
Game Assets	Unkown [1]	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. Similar to the Hash Tree, this will typically be too large to store on Ethereum so, given that it is public and non-essential data, IPFS will be used to store and distribute it.
Game Data	*avg. 44GB* [2]	Peers	This will the data required to play the game and will be fetched based upon the contents of the game's Hash Tree. This data is way too large to store on Ethereum but also isn't public, which means using IPFS would not be appropriate [3]. Therefore, this project will use a custom P2P network for sharing data, which is described in Section 4.2.2

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

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

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

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

Swarm [5] 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. On top of that, IPFS has much greater adoption and is much more mature in terms of working on a large scale.

4.2.1 Blockchain

Type of Blockchain

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

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

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

Uploading Games

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

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

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

Purchasing Content

Users will purchase content from developers over Ethereum using Ether (**F-M9**) and this will be recorded on the blockchain (**F-M10**). Any user can see which other users have purchased the game users can prove this between each other using their public/private keys.

4.2.2 Distributed File Sharing

Hash Tree

The Hash Tree of a given directory is 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 shard of data. This allows users to easily identify and verify game data.

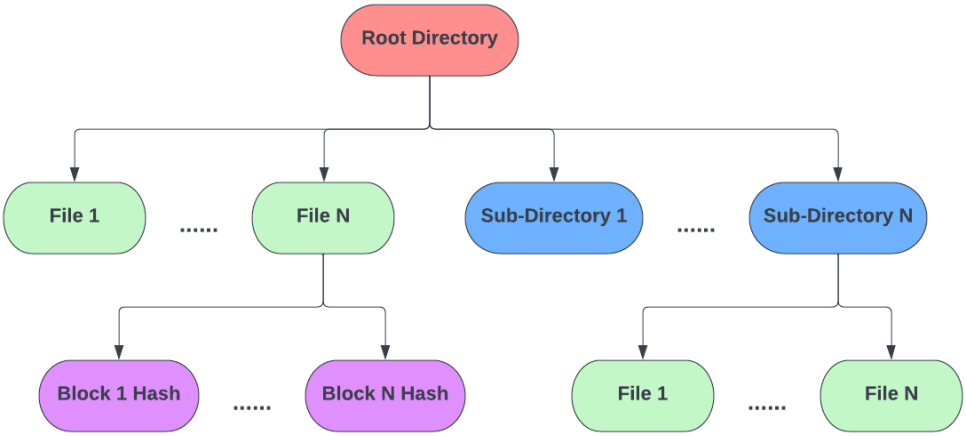

Figure 4.1: *The structure of a Hash Tree*

Uploading Content

For a developer to upload their game (**F-M5**) they must provide the required metadata outlined in Section 4.2.1 as well as the location of the game in storage. A Hash Tree is then generated and all data is sent to its corresponding platform. The game is added to the uploader's library and can now be downloaded from other users who have purchased it.

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 which will allow a user to discover peers who share the same games as them and request blocks of data from them (**F-M3**). When a peer seeking data forms a connection with another peer they will:

1. Perform a handshake to determine each other's Ethereum address and public key.
2. The seeder will verify that the downloader owns the game by checking the *purchased* mapping on the Smart Contract.
3. The downloader will send requests for individual blocks to the seeder (**F-M2**).
4. Upon receiving a block, the downloader will verify the contents using the block's hash (**F-M7**).
5. Repeat Steps 3–4 for an arbitrary number of blocks.
6. The seeder may request a signed receipt that details the blocks they uploaded.

Updating Content

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

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

Downloadable Content

Downloadable Content DLC 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.

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 the developer (**F-S1**) 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.

4.3 Architecture

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

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, which is outlined in Section 4.2

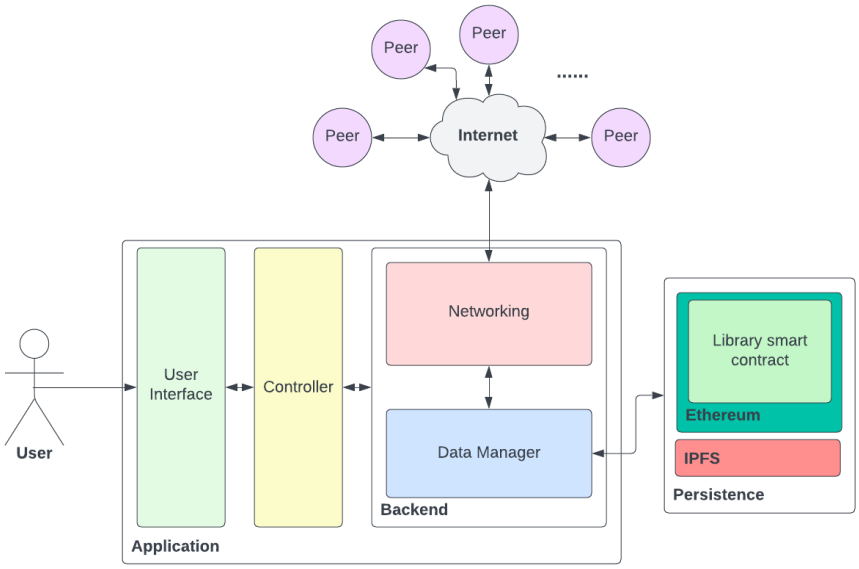

Figure 4.2: *The layers of the application*

Ethereum

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

The Geth go-ethereum package https://geth.ethereum.org/ will allow us to interact with the ethereum blockchain and Abigen https://docs.avax.network/specs/abigen will allow us to compile any smart contracts to Go code. This will allow us to interact with our smart contract on ethereum using a set of Go functions. For development, Ganache https://github.com/trufflesuite/ganache was used to create a local Ethereum instance and Geth was used to connect to an Ethereum test net.

IPFS

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

4.3.2 Backend

The Backend can be broken down into two major parts:

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

Networking

Users running this application will be a part of a distributed network of peers by creating and maintaining a set of TCP connections with other users in the network and will communicate by sending structured messages to each other. Section 4.3.2 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 The main responsibility of this section is to respond to requests sent by the Data Manager by sending and tracking messages to other peers to fetch the 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 will typically come as part of a request/response pair involving the sharing of information between peers.

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

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

Table 4.5: *The 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 and which aren't.
2. Interact with the Library contract deployed to the Ethereum blockchain to discover, upload and purchase games.
3. Fetch shards of game data to send to other peers.
4. Interact with IPFS to upload/fetch a game hash tree and store assets.
5. Sending requests to the networking layer to find and retrieve blocks of data from peers in the network.
6. Generate hash trees of games and use it for verification of data.

Optimisations

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

Some of the areas this pattern was used include:

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

Ignore File A standard implementation of a .ignore file was included to indicate to the hash tree algorithm which files/folders to ignore. This is useful to ignore temporary or non-static files, which contents will vary by user and thus won't need to be distributed.

4.3.3 Frontend & Controller

Frontend

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

- **Library** The user's collection of owned games, where they can view details of each game owned as well as manage their download status.
- **Store** Where user's can find new games that have been uploaded by other users and purchase them.

- **Upload** Where user's can fill in details about a new game and have it be processed and uploaded to the blockchain.
- **Downloads** Where user's can track all of their owngoing downloads.
- **Peers** Where users can manage their list of connected peers.

Controller

The Controller will be the interface functions used for communication between the frontend and backend. Each function should be used to trigger an action/process or fetch some data. Some example Controller functions will include:

- **PurchaseGame** Purchase a game off of Ethereum and add the metadata to the user's library.
- **StartDownload** Create a download for one of the user's owned games and start looking for blocks of data for it.
- **UploadGame** Upload a game to Ethereum given a set of input parameters.

4.4 Limitations

Social Feature This application will not contain any of the social features found in platforms like Steam. This includes things like friends, achievements, and messasge boards.

Availability Section 2.1 highlights an issue of availability within P2P systems and this application will likely be similar. These could be mitigated by having an active community or good incentives but there is no guarantee when compared to centralised platforms.

Inefficient Updates Updates will be treated as individual games and will this require the user to donwload the entire game again. This is highly inefficient and will result in duplicate data being needed to be downloaded by the user.

Hash Trees The main downside of a Hash Tree is that each file must have at least one block. This means that a large number of blocks may not necessarily correspond to a large amount of data.

Chapter 5

Implementation

5.1 Backend

The backend code for this application was written based upon the design from Section 4.3.2 and was done using the following tools:

Tool	Description & Reasoning
Go	Go was chosen because of its simple syntax, high performance, strong standard library and third party packages for interacting with Ethereum.
go-ipfs-api	A Go package used for interacting with the Kubo implementation of IPFS that gave an easy interface for downloading and uploading data to Kubo.
go-ethereum	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.

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

5.2 Smart Contract

To write and deploy a smart contract that met the criteria specified in Section 4.2.1, I used the following tools:

Tool	Description & Reasoning
Solidity	The language used to write smart contracts for the Ethereum blockchain.
Sepolia	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.
Alchemy	Alchemy provides useful tools for interacting with Ethereum and specifically Sepolia, such as an ETH faucet and an RPC URL.

MetaMask	A browser-based wallet that can easily be conencted to other tools such as Alchemy or Remix.
Remix	A browser-based IDE for writing smart contracts that allows for easy deployment.

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

The contract was successfully deployed to Sepolia and the details of it can be viewed at https://sepolia.etherscan.io/address/0x2899dab55a4a20d698062bbf4d4ce9f1073ce052.

5.3 Other Tools

The following tools were also used throughout development:

Tool	**Description & Reasoning**
Git	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	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	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	Lucidchart was used to create all of the diagrams for this project.

Table 5.3: *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** Tests should be written alongside the code to reduce the risk of bugs and improve robustness.
2. **Fail Fast (Smoke testing)** Automated tests should be ran in a pipeline where the fastest tests are always ran first to reduce the time spent ranning tests.
3. **Documentation** Test cases should be well documented and group contextually.

Tools

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

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

Table 6.1: *The tools used for testing my project*

6.2 Unit Testing

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

6.3 Integration Testing

Profiles

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

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

Deployed Implementation

Distributing Instances

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

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

Ethereum Test-Net

The Library Smart Contract, from Section 4.3.1, will need to be deployed to an Ethereum test-net to allow instances of the application, that are distributed over the

6.4 Acceptence Testing

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

The following acceptence tests will be included:

Id	Requirements	Description

1	(**F-M1**) (**F-M9**) (**F-M10**) (**F-C2**) (**NF-M4**) (**NF-S2**) (**NF-C1**)	A user walkthrough that shows a user uploading a game and that game being visible on the store to a separate user. A separate user will then purchase that game. Unit tests are also provided to show the correct functionality of the Smart Contract.
2	(**F-M2**) (**F-M5**) (**F-M6**) (**F-M7**) (**F-M10**) (**F-S2**) (**F-C2**) (**NF-C1**)	A user walkthrough that shows a user connecting to a peer and downloading a game off of them and those files being successfully downloaded. This assumes that the user has successfully purchased the game. Integration tests show the processing for verifying a user and data sent over the network.
3	(**F-M6**) (**F-M7**) (**NF-S1**)	Benchmark tests to show the scalability of downloading a game by varying given conditions. See Section 6.5.
4	(**F-S2**)	A user walkthrough in which a user does not validate their Ethereum address when connecting to peers. The user attempts to download blocks off of their peers but is rejected due to not being verified.
5	(**F-M4**) (**NF-S2**) (**NF-M3**) (**NF-M4**)	User walkthroughs showing two attempted uploads for a given game that already exists on the network. The first by the owner showing the process of selecting their previous game and uploading an update and another by a non-owner attempting the same thing. Unit tests for the Smart Contract can also show functionality.

6.5 Benchmarking

Benchmarking is being used to determine the overall performance and scalability of the application

Benchmarking is being used in this project to determine the overall performance and scalability of the application, whilst also being useful in identifying any bottlenecks or how the end user can optimise their inputs. The key benchmark being assessed is related to how the application scales downloading games by varying the following factors:

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

To ensure the consistency and correctness of results, all benchmarks will be ran on the same machine, running the same OS, and be completed multiple times. The test data is a collection of pseudo-randomly generated files that meet the criteria specified in each benchmark.

Number of Peers

This benchmark allows us to observe how the application scales when dealing with many peers at the same time and how if affects the overall performance of the application.

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

	Runtime (s)			
Peer Count	**1**	**2**	**3**	**avg.**
1	56	65	63	61.3
2	65	64	60	63
4	66	65	65	65.7
8	66	62	60	62.7

Table 6.3: *How varying peer count affects download speed*

Taking into account a degree of error, it is clear that increasing the number of peers does not reduce the download time. This indicates that a bottleneck may exist elsewhere in the application such as inserting received data. This is supported by the observation that downloads got much slower towards the end and would often take $\tilde{1}5$ seconds for the last 10%; however, this could also be caused by the file verification that is ran once an entire file is downloaded.

However, this result also shows that many peers can be supported without an impact on performance. However, maintaining TCP connections will be expensive for a very large number of peers so a UDP implementation should be a consideration moving forward.

Game Size

This benchmark will be useful in discovering an optimal strategy for determining the directory structure of games uploaded to the network to allow developers to optimise

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

their uploads to give the greatest download speed.

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

File		Runtime (s)			
Count	**Size**	**1**	**2**	**3**	**avg.**
200	200				
100	400				
50	800				
25	1,600				
5	8,000				
1	40,000				

Table 6.4: *How varying file count and size affects download speed*

Shard Size

This benchmark will be useful in determining an optimal shard size to use that maximises download speed.

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

	Runtime (s)			
Shard Size (bytes)	**1**	**2**	**3**	**avg.**
1,048,576				
2,097,152				
4,194,304				
8,388,608				
16,777,216				

Table 6.5: *How varying the shard size of the hash tree affects download speed*

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 may ignore some of the SHOULD or COULD requirements.
Laptop damaged or lost	3	1	3	All work is stored using version control and periodic backups will be made and stored locally and in cloud storage. I have other devices that could be used to continue development.
The application is not finished	2.5	4	10	This project is quite large in scope and will require a significant effort to implement and test. Using Agile development [7], I will incrementally increase the scope of the project and ensure I have a minimum viable application. A cut off point will be used to finish my implementation regardless of progress to focus on final testing and the report write up.

Table 7.1: *The risk assessment of this project*

7.2 Sprint Plans

For each sprint we will detail the planned requirements, whether they were completed or not, as well as any general comments about that sprint.

7.2.1 Sprint 1

The P2P distribution network was expected to be one of the most complex parts of the application so Sprint 1 was focused on getting the basics of it working.

Req.	Description	Complete	Evidence
(**F-M2**)	Downloading blocks from peers	YES	Acceptence & benchmark tests
(**F-M5**)	Upload blocks to peers	YES	Acceptence tests
(**F-M6**)	Download the entire game	YES	Acceptence tests
(**F-M7**)	Verify downloaded data	YES	Acceptence & unit tests
(**F-M11**)	Maintain TCP connections	YES	Unit tests
(**F-M12**)	Generate Hash Trees	YES	Unit tests

7.2.2 Sprint 2

Sprint 2 was largely spent on two main aspects: creating the Smart Contract to store game metadata and start working on a user interface to show to the user. This sprint had a slower start due to me not being familiar with writing Smart Contracts and having difficulty in finding and learning a suitable GUI framework.

Req.	Description	Complete	Evidence
(**F-M1**)	Store data on Ethereum	YES	Smart Contract unit tests
(**F-M8**)	Connect to the Ethereum network	YES	Smart Contract unit tests
(**F-M9**)	Purchasing games	YES	Smart Contract unit tests
(**F-M10**)	Proving a game purchase	YES	Smart Contract unit tests
(**F-C2**)	Provide a simple GUI	YES	See Appendix A for screenshots
(**NF-M3**)	Uploaders are publically identifiable	YES	Smart Contract unit tests
(**NF-M4**)	Game data cannot be changed	YES	Data on Ethereum is effectively immutable
(**NF-S2**)	Only the original uploader can release an update	YES	Smart Contract unit tests
(**NF-C1**)	Intuitive GUI	YES	See Appendix A for screenshots

7.2.3 Sprint 3

Req.	Description	Complete	Evidence
(**F-S1**)	Proving the amount of data seeded	YES	
(**F-S2**)	Validate a user's ownership of a game before sending data	YES	

7.2.4 Incomplete Requirements

There were several requirements, which were left incomplete due to a variety of reasons.

Req.	Description	Reasoning
(**F-M3**)	Discover peers based upon what games they own	This requirement was left incomplete intentionally due to the complexity it would have brought to the project. A solution could include using trackers, similar to BitTorrent, or a neighbour discovery algorithm.
(**F-S3**)	Allow for the distribution of DLC	DLC was left out of scope due to a combination of time constraints and an inefficienies within the update system that I felt should be resolved before implementing the DLC system.
(**NF-S3**)	Only the original uploader can release DLC for their game	

Chapter 8

Evaluation

8.1 Project Organisation

Agile Development

The Agile Methodology [7] is a continuous development cycle that allows for developers to react to change in terms of the scope or requirements for a project and was used throughout this project to organise my time.

Sprints

One useful advantage of dividing requirements into sprints is that it gives you a greater understanding of how requirements should be prioritised in terms of producing a minimally viable product.

Sprints also gave me fixed time windows in which to complete and write tests for certain parts of the code....

Test Driven Development

Test Driven Development [1] TDD worked extremely well with Agile development as it meant that as I was completing requirements I was also testing their functionality. This meant that, in the long run, it was much easier to locate bugs and ensure the functionality of existing code through regression tests.

Tests were also useful in helping with me reason with my code. For example, if a new feature broke a lot of tests then the code might be too tightly coupled and should be considered for a refactor.

One disadvantage is that writing tests are time consuming and, when working with distributed systems and or concurrent code, can be quite difficult to make. This made writing my application tedious.

8.2 Outcome of the Application

8.3 Limitations and Future Work

8.3.1 Limitations

8.3.2 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.

Optimisations

New Commands Extending the command set from Section 4.3.2 with more complex ones (such as batch requests) could potentially reduce the number of interactions two peers would need to make and remove a lot of overhead gained from exchanging lots of commands.

UDP Over TCP Currently each user manages a set of TCP connections with their peers and this creates a lot of overhead from having to maintain channels of communication that may not necessarily be used. A UDP approach would allow for faster communication with less network overhead at the expense of reliability and greater complexity.

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.

New Features

Automated Rewards Currently there is no system in place to automatically reward users for contributing. A micro-payment system like present in Swarm [5] could be used to give a guaranteed incentive to users.

Chapter 9

Conclusion

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, I can say that this project was successful in providing a proof-of-concept application t

Appendix A

Screenshots

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

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

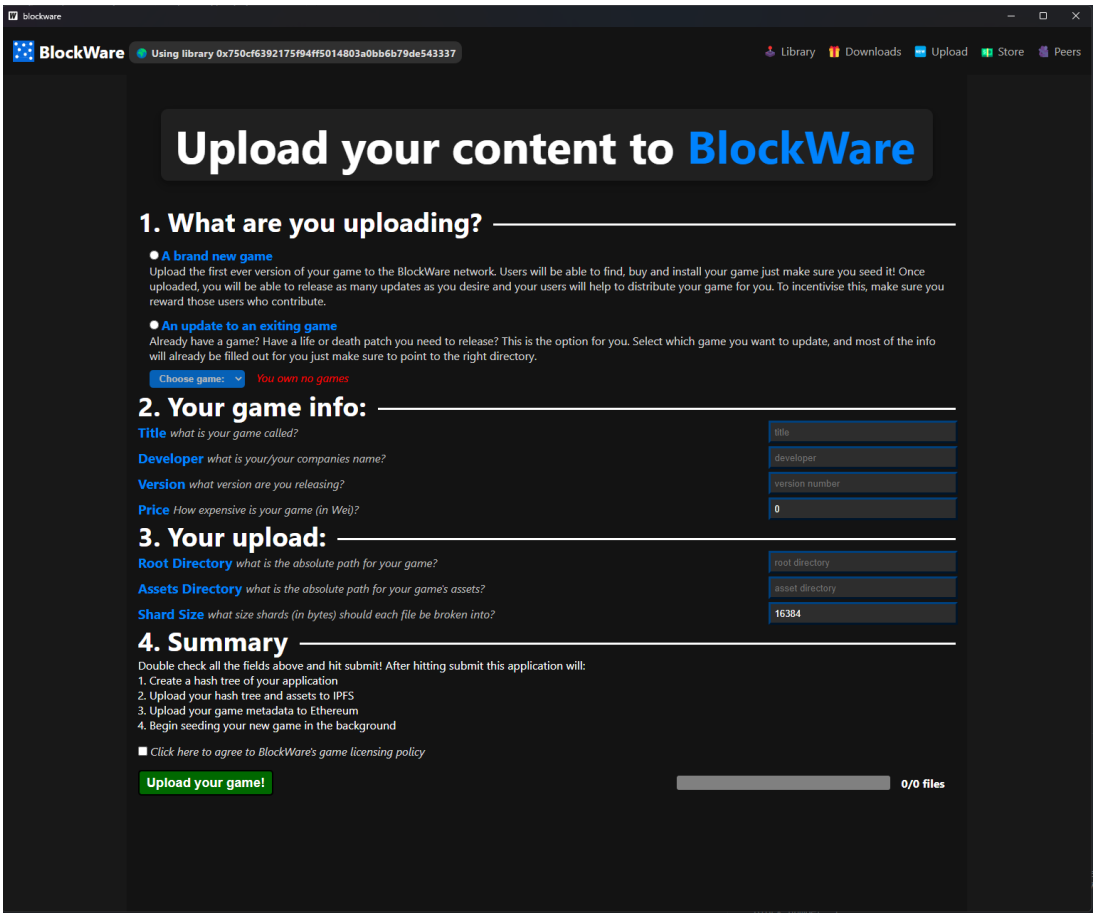

Figure A.3: *The page where users input the details about their game and can upload it to the Ethereum network.*

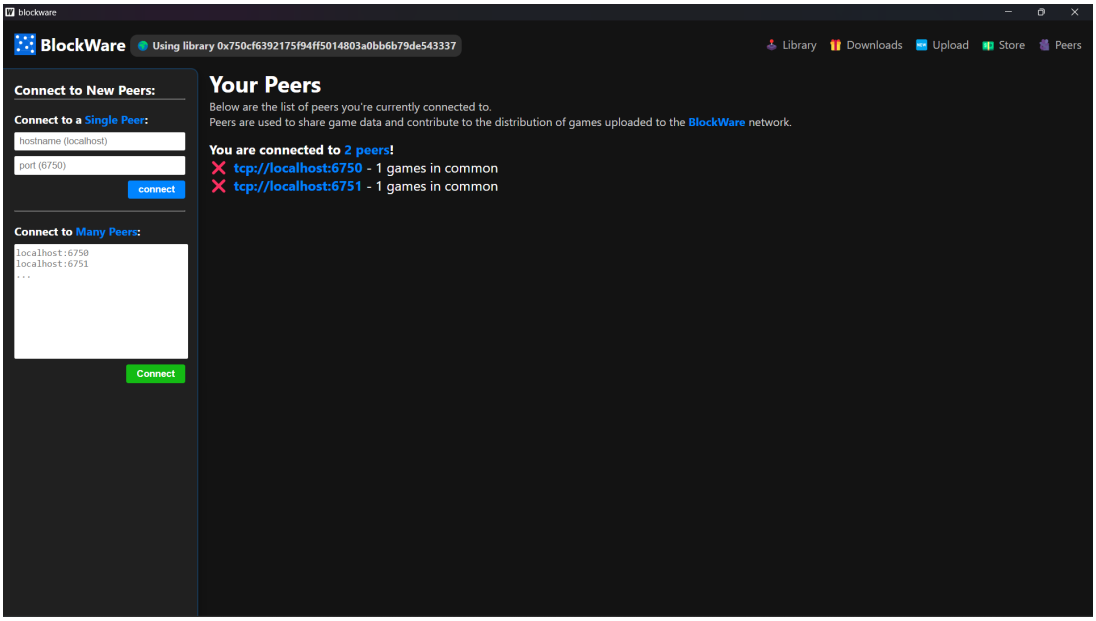

Figure A.4: *The page where users can manage their connections to peers with whom they will download game data off of.*

Bibliography

[1] Beck, K. *Test-driven Development: By Example*. Addison-Wesley Professional, 2003. Google-Books-ID: CUlsAQAAQBAJ.

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

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

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

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

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

[7] Ilieva, S., Ivanov, P., and Stefanova, E. Analyses of an agile methodology implementation. In *Proceedings. 30th Euromicro Conference, 2004.* (Sept. 2004), pp. 326–333. ISSN: 1089-6503.

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

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

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

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

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

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

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

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

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

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

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

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

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