OMNICACHE

By

Mohamad Nizar Daouk

Patrick Balian

Anthony Tabbal

Advisor

Dr. Elie Nasr

Senior Project Submitted to the Faculty of Arts and Sciences

Department of Computer Science

AMERICAN UNIVERSITY OF SCIENCE & TECHNOLOGY

In Partial Fulfilment of the Requirements for the Degree of

Bachelor of Science in

Computer Science

Achrafieh

21/05/2021

We certify that we have attended the presentation, read, and verified this project and that, in our opinion, it is satisfactory in scope and quality as a senior year project for the degree of Bachelor of Science in Computer Science.

Project Committee

Elie Nasr, Ph.D.
Chairperson and Project Advisor,
Computer Science Department, AUST - Beirut

Antoine Aouad, M.S
Director of Sidon Branch,
AUST - Sidon

Charbel Boustany, M.E
Director of Zahle Branch and Project Advisor,
AUST - Zahle

Giuseppe Boschiero, M.S
Lecturer and Game Design/Developer Expert

Acknowledgments

We would like to extend our gratitude to Dr. Elie Nasr; not only did he supervise our work, but also meticulously aided us with technicalities of the research paper; his aid helped us push harder to get better research results throughout the whole project as well as making us more confident in presenting the idea.

We would also like to thank Dr. David Khoury for pointing us in the right direction regarding Peer-to-Peer communication and cryptography concepts.

No words can describe how grateful we are to Dr. Saeed Raheel. He taught us to think programmatically and to solve problems in a short time.

Also, a great thanks goes to Mrs. Nathalie Aouad, who taught us all we need to know about networking, which is a core part of OmniCache.

Finally, we would also like to thank our families for supporting us and for giving us the opportunity to continue our studies.

Abstract

Purpose - This paper/project presents an approach to provide a decentralized, scalable, fault-tolerant, and secure file storage service using Blockchain Technology. Centralized solutions have been plagued by issues of security, longevity, accessibility, in addition to several privacy concerns. From high pricing to low transmission speeds, and large attack surface areas, an attempt is taken to mitigate these issues using Distributed Systems.

Design/Methodology/Approach/Specification - This project, called "OmniCache", aims to provide an answer to these problems. The basic premise is to share a file across a Peer-to-Peer (P2P) network and track the encryption, storage, retrieval, and contract administration using a Blockchain. "Omnies" is the cryptocurrency transacted by lenders and renters of storage, which enables users to pay for storing files on nodes of the same network.

Findings - Based on this approach, a Blockchain-based distributed storage service available as a desktop software and implemented in Python is developed. This project utilizes Go Ethereum™ as the Blockchain platform, Solidity and Remix for developing Ethereum™ smart contracts. PyQT was used to implement the graphical user interface in an effort to make the interface more user friendly.

Practical implications - OmniCache enables users to have access to an anonymous and secure way for individuals to store their files as well as offering an ever-growing remote storage capacity. In addition, users that are perpetually investing resources in the network are given incentive to continue doing so. Users with unused free storage space can profit by renting the space to other peers on the network. Omnies that are granted can be used to store more files on the network.

Originality/Value - The aforementioned approach aims to grant users a safe, trusted and fault-tolerant environment where issues present in centralized solutions are nonexistent, thus creating an ecosystem fueled by the users themself.

***Keywords**: Blockchain; Ethereum™; Peer-to-Peer network; storage; smart contract; decentralized application.*

TABLE OF CONTENTS

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION AND BACKGROUND

The advent of the internet and mobile broadband has empowered billions of people globally by providing access to the world's knowledge, high-fidelity communication, and a wide range of lower-cost and more convenient services.

With cloud-based services becoming increasingly popular, apps such as Dropbox, Box, and services offered by mobile device vendors (i.e., Apple iCloud, Microsoft OneDrive, and Google Cloud Platform) offer remote storage and syncing of user data. These apps are commonly used to quickly share files over multiple synced devices and for data redundancy. From an information security perspective, data contained within these services can be vulnerable due to weak authentication protocols, weak passwords, and improper permissions issues (i.e., inadvertently making private files public).

1.1 - Introduction

As the Internet grew and developed, so did the people's reliance on it, new technologies and protocols were created and used in the production of more complex digital solutions to all kinds of things. One of the side effects of this over reliance on the Internet was the influx of data and concerns regarding its storage and safety. Breaches, data loss, and privacy invasion all became major issues that needed to be resolved, and the centralized cloud approach that major companies adopted did not do much to mitigate these issues. By leveraging the power of Blockchain technology and decentralized P2P networks, it is possible to ensure a higher degree of performance, privacy, security, and redundancy. The rest of this chapter provides sufficient background information and is organized in the following manner: Section 1.2 explores the evolution of storage solutions. Section 1.3 delves into decentralized and centralized networks and the differences between them. Finally, section 1.4 discusses P2P networks and Blockchain technology in-depth and the benefits reaped from choosing them.

1.2 - Storage Evolution

Since the earliest computers, the storage and recording of digital data has always been a major concern. External memory devices such as HDDs and SSDs have been a cornerstone of long-term data storage in modern computers, but due to the limitations that came with having data stored locally, especially in terms of privacy, security, and redundancy, methods to provide backup and disaster recovery came to exist notably RAID, and device mirroring [1].

Newer, remote, forms of data storage became increasingly in demand and of importance, leading to the eventual culmination of cloud storage solutions such as CompuServe in 1983 [2], and much later, Amazon with AWS making cloud storage a household name and the go-to method of storing data [3]. Several years after the emergence of cloud storage, came file sharing services such as Napster in 1999 that went on to grow and popularize P2P technology for sharing data on the internet but due to legal issues that lead to the shutdown of the Napster's centralized server [4], networks such as LimeWire rose to replace it by implementing a decentralized alternative.

With cloud and decentralized P2P now established, a new variation of cloud storage was slowly gaining traction, known as "cooperative storage cloud" [5], it takes a decentralized approach to storing storage remotely on several nodes that may be rewarded and encouraged for their contribution. Blockchain and Ethereum™ came and offered more ways of auditing and offering rewards via cryptocurrency [6].

1.3 - Centralized Vs Decentralized

A centralized network architecture is built around a single server that takes care of all the processing. These networks are controlled by a central authority that makes decisions on behalf of the rest of the network. Less powerful computers connect to the server and submit their requests to the central server rather than performing them directly, like in Figure 1.1 below.

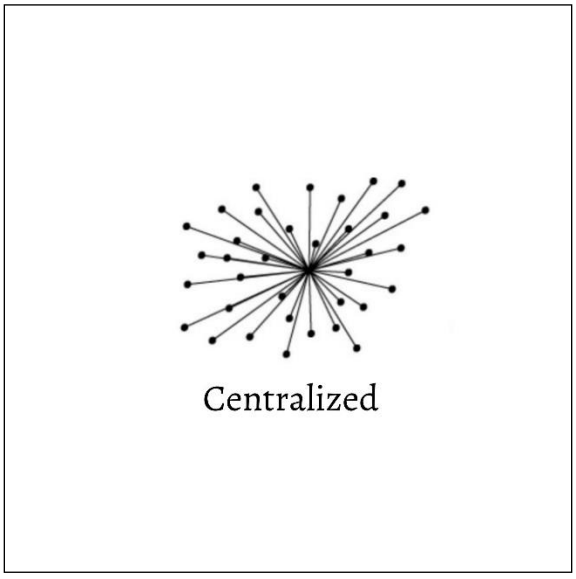

Figure 1.1 - Centralized Network Architecture

A decentralized network distributes tasks among multiple machines, instead of relying on a single server. Decentralized networks offer performance well beyond the needs of most applications, meaning the extra computing power can be used for distributed processing.

Instead of relying on a single centralized server, decentralized networks distribute data, information, and processing workloads across the computers participating in a network and make decisions and perform actions in a democratized way. This allows for greater fault tolerance, if one machine goes down, the network continues to function. The figure 1.2 below depicts Decentralized Networks.

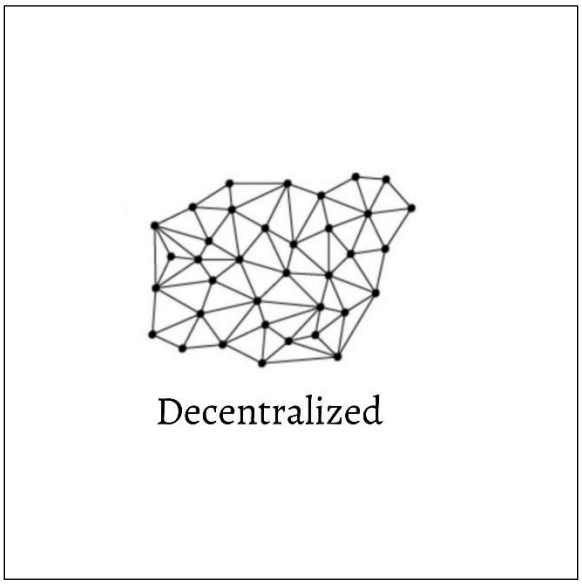

Figure 1.2 - Decentralized Network Architecture

In centralized networks, management over the network is held by a central server. Whereas, in decentralized networks, power is distributed between the computers on the network. Centralized structures make it simpler to upgrade and maintain the network but bring on a weakness: if the main server fails or gets hacked, the entire network is compromised. On the contrary, decentralized structures are very resistant to hardware failure and security breaches but require a larger number of participants to be effective [7][8].

1.4 - Peer-to-Peer Networks and Blockchain

In this section, Peer to Peer networks and Blockchain technology are discussed.

A Peer-to-Peer (P2P) network is a Distributed Systems [9] architecture, in which computers known as “peers” or “nodes” communicate together without a central server to manage the network.

Typically, environments implement a client/server architecture [10], where the server usually provides a service to the client, which is said to “consume” the service. On the other hand, in a P2P network each peer is simultaneously a provider and a consumer.

Many P2P networks and technologies are already well established. Networks like BitTorrent, Skype, DistriBrute, Tiber, Pando, etc. Different P2P networks and technologies have their own stacks and purposes. Some are publicly accessible, and others are not. Also, network scale could be global with millions of nodes, or small work groups of 10 to 50 nodes.

Peer-to-Peer networks implement some form of virtual overlay network [11] on top of the physical network topology, where the nodes in the overlay form a subset of the nodes in the physical network. Three P2P network model are available: unstructured networks, structured networks, and hybrid models.

Unstructured P2P networks do not follow a specific structure on the overlay network by design but are composed of nodes that randomly form connections to each other. (Examples of unstructured P2P protocols Gnutella, Gossip and Kazaa).

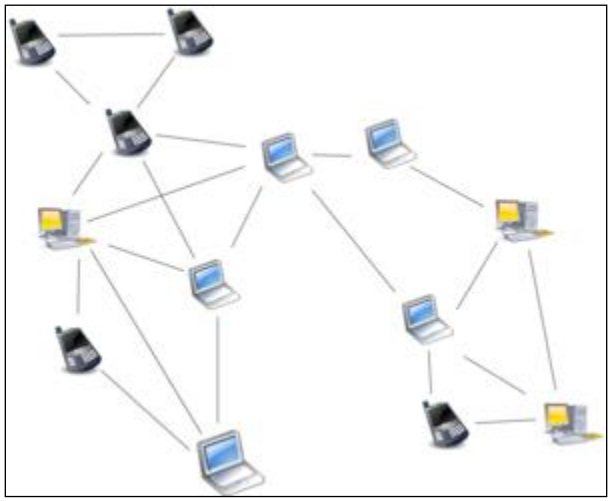

Figure 1.3 - Overlay Network Diagram for an Unstructured P2P Network

Structured networks feature an overlay that is organized into a specific topology along with a protocol assuring efficiency while searching for any resource on the network. Commonly structured P2P networks implement a distributed hash table (DHT) [12] in which a variant of consistent hashing is used to assign ownership of each file to a particular peer.

Hybrid models combine Peer-to-Peer and Server/Client models. Hybrid models consist of a central server that helps users find each other.

Blockchain technology leverages a Peer-to-Peer network to provide a replicated secure ledger. It was introduced by Satoshi Nakamoto in his Bitcoin paper as the technology that solves the cryptocurrencies' double-spending problem without any intermediary third party, like banks, servers, etc. [13]. In a Blockchain network, peers maintain a full copy of the transactions' ledger consisting of a sequence of cryptographically linked blocks. Each block holds a set of transactions and the hash of the previous block as shown in Figure 4.

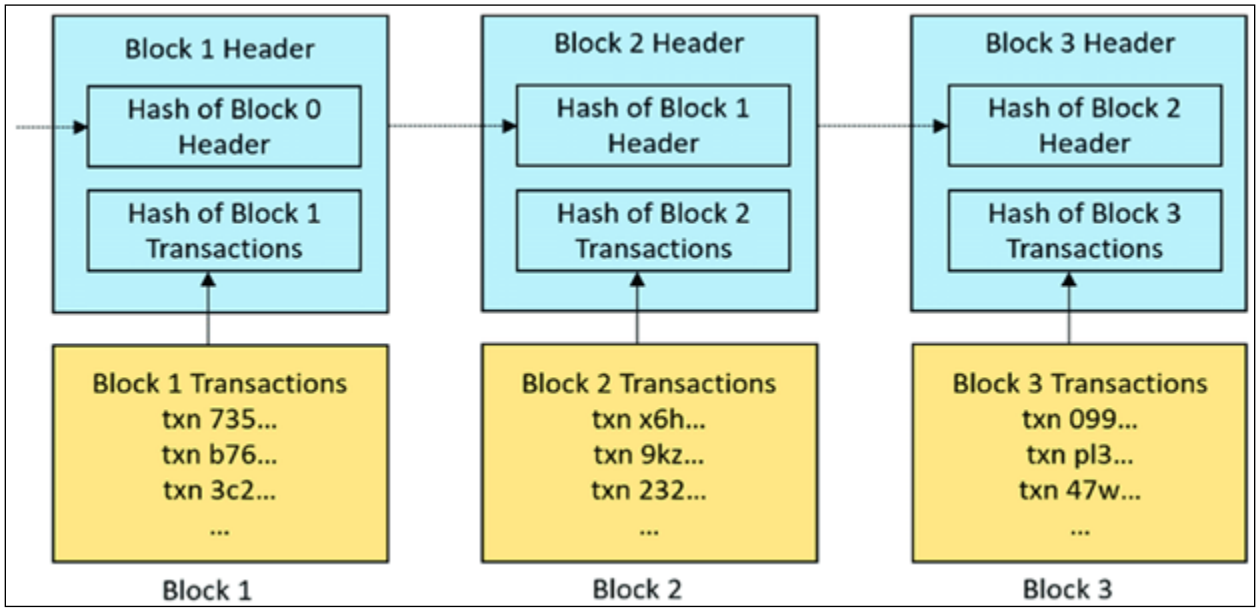

Figure 1.4 - Example Blockchain

The distributed ledger is shared across many peers in the network and is constantly being updated, hence a consensus must be reached as to whether the current state of the ledger is valid or not, across all peers. This is performed using what is known as consensus algorithms. Several of these algorithms exist [14], most notably, Proof-of-Work (PoW), which lets nodes compete by solving computationally intensive mathematical problems, and Proof-of-Stake (PoS), which randomly selects a node to create a block, while staking some of its coins in the network, ensuring the integrity of the transaction's data is preserved among all peers on the network. Another important consensus algorithm, and the one utilized in this project is Proof-of-Authority (PoA), the basic premise is that instead of staking coins like PoS, validators stake their own reputation, this allows it to be more efficient than PoW and more effective and robust than PoS,

especially in a private network [20]. PoA also deters several attack vectors that other algorithms suffer from since validators are thoroughly checked and verified prior. Their identities are kept anonymous and encrypted and only revealed because of negative or malicious behavior on their part. To be able to interact with the Blockchain, the users need to create a wallet which translates to a public/private key pair. Public keys are used to address the wallet and verify its transactions by other users, while the private key is used to sign transaction.

The Ethereum™ Blockchain is an open-source platform providing smart contract scripting functionality [15]. Rather than limiting users to a specific set of transaction types and applications, the platform allows anyone to create any kind of Blockchain application by writing a script and uploading it to the Ethereum™ Blockchain.

A smart contract is a program running on the Ethereum™ Blockchain. It is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum™ Blockchain. Smart contracts, being a type of account, means that they have a balance associated with them. They can send transactions autonomously, as they are deployed to the network and run as programmed. Once it is deployed, user accounts can interact with the smart contract by submitting transactions that trigger a function on the smart contract.

1.5 - Contributions

Big companies have monopolized the remote file storage industry giving themselves access to a huge amount of user data. Having a global centralized solution for file storage creates security risks and privacy issues. Data centers have become a single point of failure that threatens the integrity and privacy of the users' files.

With the proliferation of Blockchain technology, companies have begun to create decentralized solutions for file storage. Some build their solution on top of existing Blockchains like Storj™ and other create their own protocols and proofs like FileCoin™. Decentralized options look to be very promising because they have the following characteristics: fault-tolerant, secure, immutable. However, a lot of different combinations leveraging a wide variety of technologies and their nuances are still unexplored.

This paper studies the combination of a private Ethereum™ network and custom protocols to achieve a decentralized file storage solution that minimizes Blockchain usage by limiting it to

auditing interactions and by diverting network gossip and file transfer to an off-chain Peer-to-Peer network.

In this chapter, the history of storage evolution and the advantages as well as the disadvantages of centralized and decentralized architectures were tackled. Peer-to-Peer networks and Blockchain technologies were also explained, and some of their advantages were also stated. In chapter two, the problem statement is discussed as well as what others have done in the decentralized space.

In chapter three, a new system is proposed and explained in terms of how it differs from the rest. In chapter four, design specifications and diagrams about system architecture, use case, sequence and workflow are presented. In chapter five, implementation details are brought to light and explained in detail with code snippets. In chapter six, security measures are dissected and explained systematically. Chapter seven provides information about testing and verification. Finally, chapter eight constitutes the conclusion and future work.

CHAPTER 2

PROBLEM STATEMENT AND LITERATURE REVIEW

This chapter presents detailed research about the serious problems of cloud file storage and what they entail regarding user's data. It also mentions previous and similar solutions to the one proposed, while covering their advantages and disadvantages.

2.1 - Disadvantages of Cloud File Storage

2.1.1 - Connectivity and Bandwidth

Internet failure results in costly downtime for cloud storage companies. Furthermore, users need to wait a long time to access their remotely stored data if the internet connection is slow. Another potential downside in public cloud storage may be download and data speed (i.e., bandwidth). Although cloud storage has achieved some excellent benchmarks [16], it still falls short with some bottlenecks especially when it comes to data transfer [17].

2.1.2 - Downtime

Cloud service providers look after several customers daily, they can get overwhelmed and even face technical failures. This may lead to a temporary suspension of business processes. If your internet connection is offline, you cannot access any of your cloud data. No cloud provider provides free disconnection. In 2014, the leading cloud storage provider, Dropbox, had a two-day breakdown [18]. For many Dropbox customers, not being able to access their files caused a lot of problems. Moreover, centralized servers may face an abrupt failure of the entire system that might take a lot of time to recover from.

2.1.3 - Security and Privacy

Even though cloud service providers are certified according to safety and industry standards, the storage of critical files at external service providers could still create risks. Cloud storage means the transfer of confidential information by third parties, since using technologies that are built on top of the cloud means that your service provider has access to essential private data. You need full company confidence before attempting to transfer data to a cloud storage provider to keep your data secure. Cloud storage companies have been known to go wrong in the past. As a public service, cloud providers are confronted with routine security challenges. Malicious users can also scan, identify, and exploit system-wide loopholes and vulnerabilities with easy access and procurement of cloud services.

Hypothetically, a hacker could try to break down data belonging to other users and store it in a multi-tenant cloud architecture on the same server, in which several users are hosted.

Privacy concerns are a common talking point when speaking about cloud technologies**.** Some users have cited concerns over privacy because Microsoft has reserved the right to scan files saved on OneDrive to look for what it calls 'objectionable content', such as copyrighted material or explicit images. Apple has a similar policy for its cloud service, Apple iCloud, – but the fact remains, file security cannot be guaranteed if it is deemed objectionable [19].

One of the worst privacy and security breaches occurred in 2017, when hackers stole the private financial records of some 156 million people from servers belonging to Equifax, while the 2018 Facebook-Cambridge Analytica scandal revealed how personal data belonging to up to 87 million Facebook users was harvested without their consent.

2.1.4 - Scalability and Cost

Centralized servers also offer limited scalability since all the applications and processing power are hosted in a central server, the only way to scale the network is by adding even more storage or processing power to the server. This may not turn out to be a cost-effective solution in the long run for the provider, which affects the rental prices of centralized servers. For example, the cost of the first 50TB rental on the world's most popular clouds is listed below:

Hot storage	Price
Amazon S3 Standard	$0.023/GB (first 50TB per month)
Microsoft Azure Hot Blob Storage	$0.0184/GB (first 50TB per month)
Google Cloud Storage Standard	$0.026/GB (first 60TB per month)

Figure 2.1 - Price Comparison of Centralized Cloud Servers

Figure 2.2 - Cooperative Storage Cloud Industry Giants

2.2 - Industry Giants

For the most part, the main players in the industry are very similar in operation and most implementation details. They all build upon and expand on the notion of a digital marketplace where storage is bought and sold, according to the needs of the users. Most of these solutions are built on a Proof of Work Blockchain utilizing mining as computing power for the network. The nodes operating as the storage hosting nodes are paid in tokens special to each of these solutions. Data is stored using multi-region redundancy which means that the files get split, encrypted, and distributed along multiple hosting nodes to prevent a single point of failure.

There are some differences ofcourse, Filecoin™ requires storage nodes to additionally prove that they are continuously replicating the files to pay them, and rewards fast data distribution.

Unlike Storj™, Filecoin™ and Sia™ both implement smart contracts to govern the transactions between nodes, while Storj™ users pay as they go on the storage they use. This does have a disadvantage as it means that if some user deletes the app or stops using it whoever is hosting the node won't be paid.

2.3 - Approach Taken

OmniCache differs itself in two aspects:

Its unique periodic payment model, and its Proof of Authority consensus algorithm.

Unlike the solutions mentioned earlier, users need not invest in an initial sum of money to get up and going as they are generously provided with an initial amount of Omnies on account creation. These Omnies can then be used to purchase storage.

Using a custom protocol, nodes can communicate with each other and offer their storage for rent thus generating additional Omnies. These nodes periodically transact with a smart contract to get paid, avoiding interaction with other nodes.

While Storj™, Sia™, and Filecoin™ all implement a Blockchain based on a proof of work consensus algorithm, OmniCache operates on a Proof of Authority model which saves computing power and circumvents all the issues resulting from mining. By implementing both things differently, OmniCache is able to provide a simple, convenient, and completely free alternative for individuals wanting to make the most out of their unused storage space.

Now that the disadvantages of cloud storage and other competitors have been explained. Chapter three proposes the approach taken in designing the remote storage system along with its multiple features.

CHAPTER 3

PROPOSED SYSTEM

This chapter presents important features of the proposed solution including the different aspects of the system, how they relate, and how they are presented using an intuitive design.

3.1 - Wallet System

After installing the application, a returning user is prompted to provide a private key to retrieve his wallet and associated files, whereas a new user would have a wallet with a private key created for him. A certain amount of space is reserved on each user's machine along with an initial grant of Omnies.

3.2 - Intuitive Design

The homepage displays all the user's necessary information neatly without cluttering the UI. The user is able to navigate and use the program easily thanks to a user-friendly design. Moreover, an account settings page is available for users to manage different aspects of their account. A dark theme is adopted throughout to reduce eye strain in low light conditions by decreasing blue light exposure, which makes using the solution more comfortable [21].

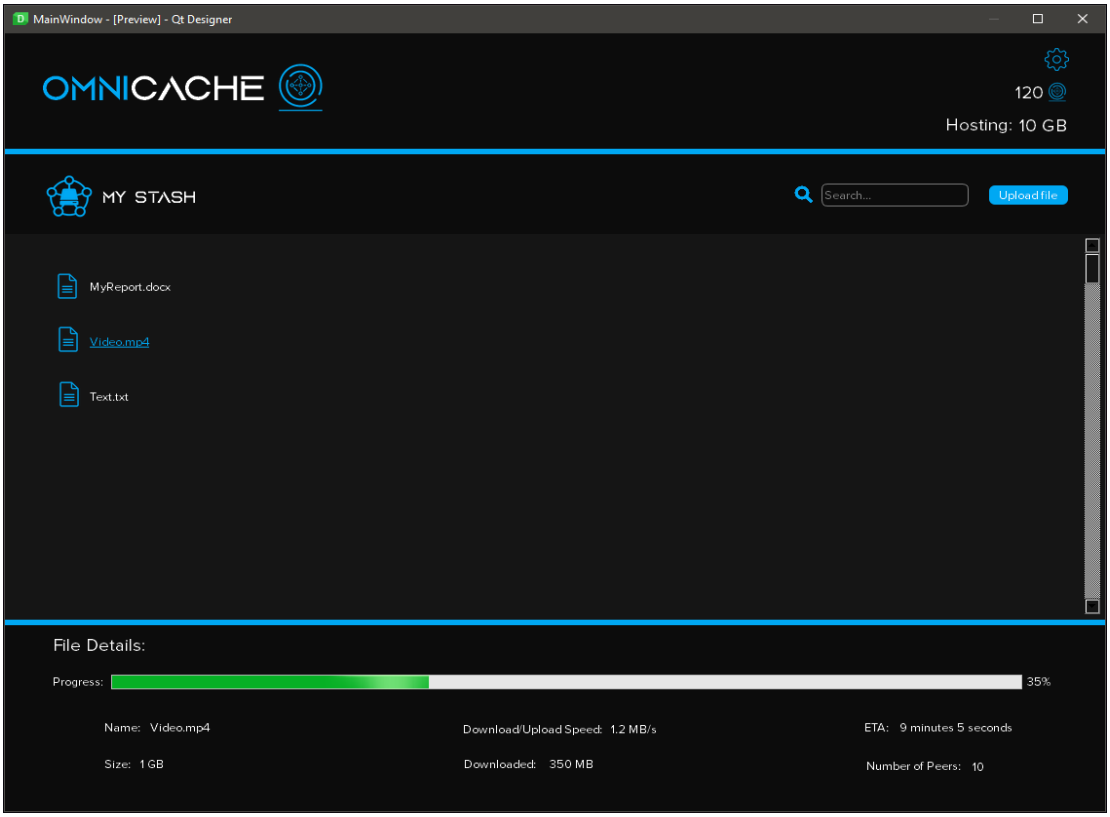

Figure 3.1 - Cooperative Storage Cloud Industry Gants

3.3 - Remote Storage

Once the user joins the network, a vast amount of storage capacity, scattered among the users, becomes instantly available to him. Having access to all those resources enables the user to be able to store files for a fee, these files get divided into chunks and distributed to multiple nodes on the network. Record keeping and auditing is handled by the Blockchain and a transaction is issued every time the user employs remote storage space. Simultaneously, users can also choose to increase the limit of reserved space on their machine allowing them to increase their potential income as more files are stored on their computer.

3.4 - Self-Sustaining Ecosystem

As the number of users joining the network grows so does the storage capacity available to the network. Users wanting to store their files must pay a fee scaling with the file size using Omnies. This cryptocurrency can be acquired in three ways:

1. Periodic payments from hosting files on the user's machine
2. An initial grant
3. Additional compensation for being available, and preserving files hosted on one's machine.

The amount varies according to several factors, the most important being the size of the file. This payment model incentivizes users to dump further resources into the network as well as encourages availability.

3.5 - Reward System

The system distinguishes the good peers from the bad ones, rewarding the former through rewards in the form of Omnies, achievements, and titles. Peers are judged based on their behavior in terms of several factors such as availability, adequate storage of files, and charitability in the amount of data they host, among other things.

3.6 - Validator Check

Validators play a key part in the ecosystem, validating transactions and being trustworthy are their primary roles. Users willing to contribute to the network should submit their full information for

processing. Afterwards, the role of validator may be granted. Users that become validators are to be granted bonus cryptocurrency along with additional achievements.

This chapter introduced the proposed solution, known as “OmniCache”, and the various features and capabilities it introduces. The next chapter details the design specifications behind “OmniCache”.

CHAPTER 4

DESIGN SPECIFICATIONS

Page

This chapter reveals the general architecture of the system including, the different modules and components used, the interaction between them, the sequence of actions performed, and the role of various actors in the system.

4.1 - System Architecture

Users are granted access to the network upon first setup which connects them to an extensive number of interconnected nodes all working to maintain a self-sustaining, fault-tolerant system.

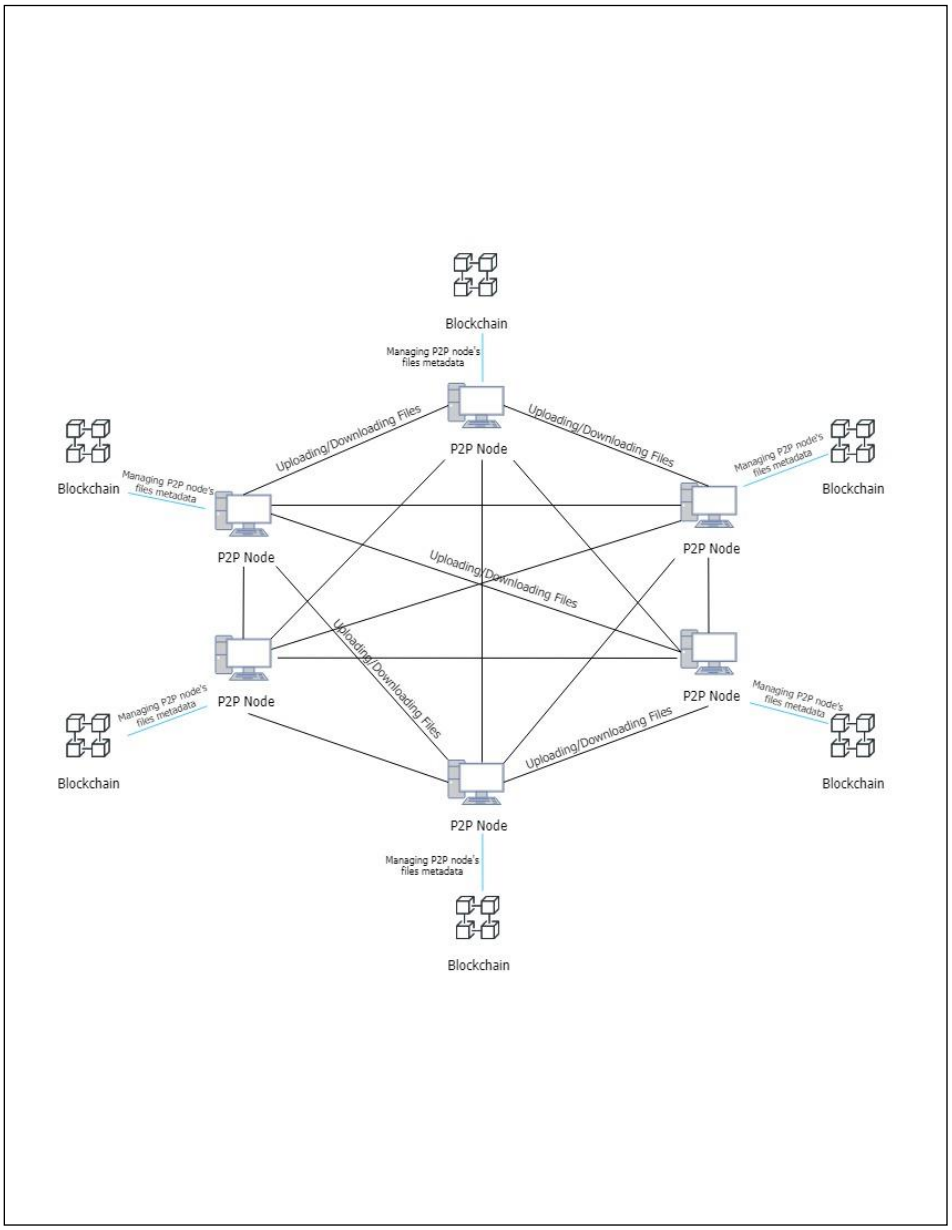

Figure 4.1 - System Architecture

The system is built on a network of nodes, each running the private Ethereum™ Blockchain using Geth. These nodes are interconnected using a custom implementation of a Peer-to-Peer network, all with its own custom-built protocols, in addition to the Blockchain network. The nodes also communicate with each other indirectly by interfacing with a smart contract using Ethereum™ transactions.

4.2 - Use Case Diagram

A use case diagram is a set of possible sequences of interactions between systems and users in a particular environment and related to a particular goal. It should contain all system activities that are significant to the actors. The below figure shows the system boundaries, along with the different activities that an actor can perform.

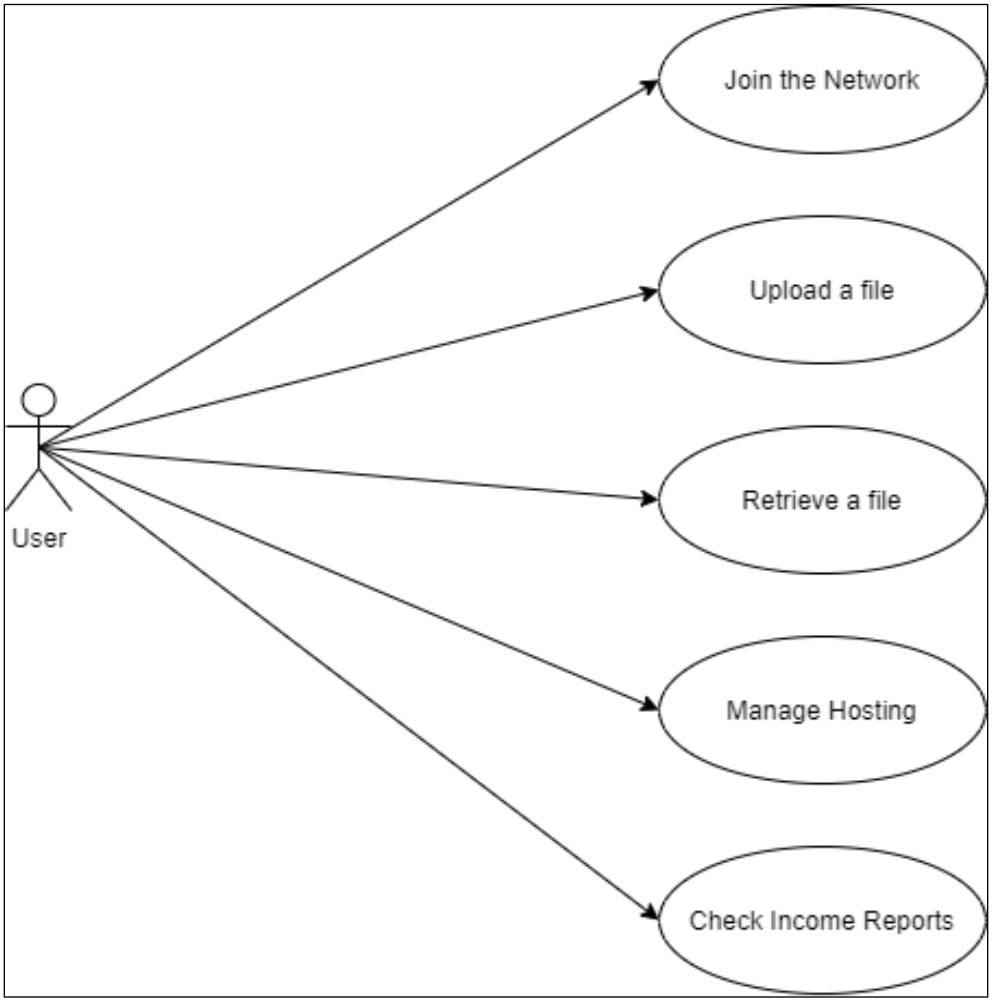

Figure 4.2 - Use Case Diagram

A user running the system can perform several actions. The first step any user has to perform to use the system is joining the Peer-to-Peer and Blockchain networks, respectively. After joining the networks, the user can now begin uploading their files, these files may also be retrieved at any time. A user can change the amount of space hosted on their machine to increase their income of Omnies, this income is logged in payment reports and can be checked whenever required.

4.3 - Sequence Diagrams

A sequence diagram is the collaboration of objects based on time sequence. It is able to capture the sequence of the Interaction between user and the system or between different systems by showing the order of messages sent, the content of each message, and when they are sent.

The process of joining the OmniCache network goes as follows, the new user must first send a join request to a node in the Peer-to-Peer network, this request bounces from one peer to the other until it finally reaches the genesis node. The below figure shows the sequence diagram related to joining the network.

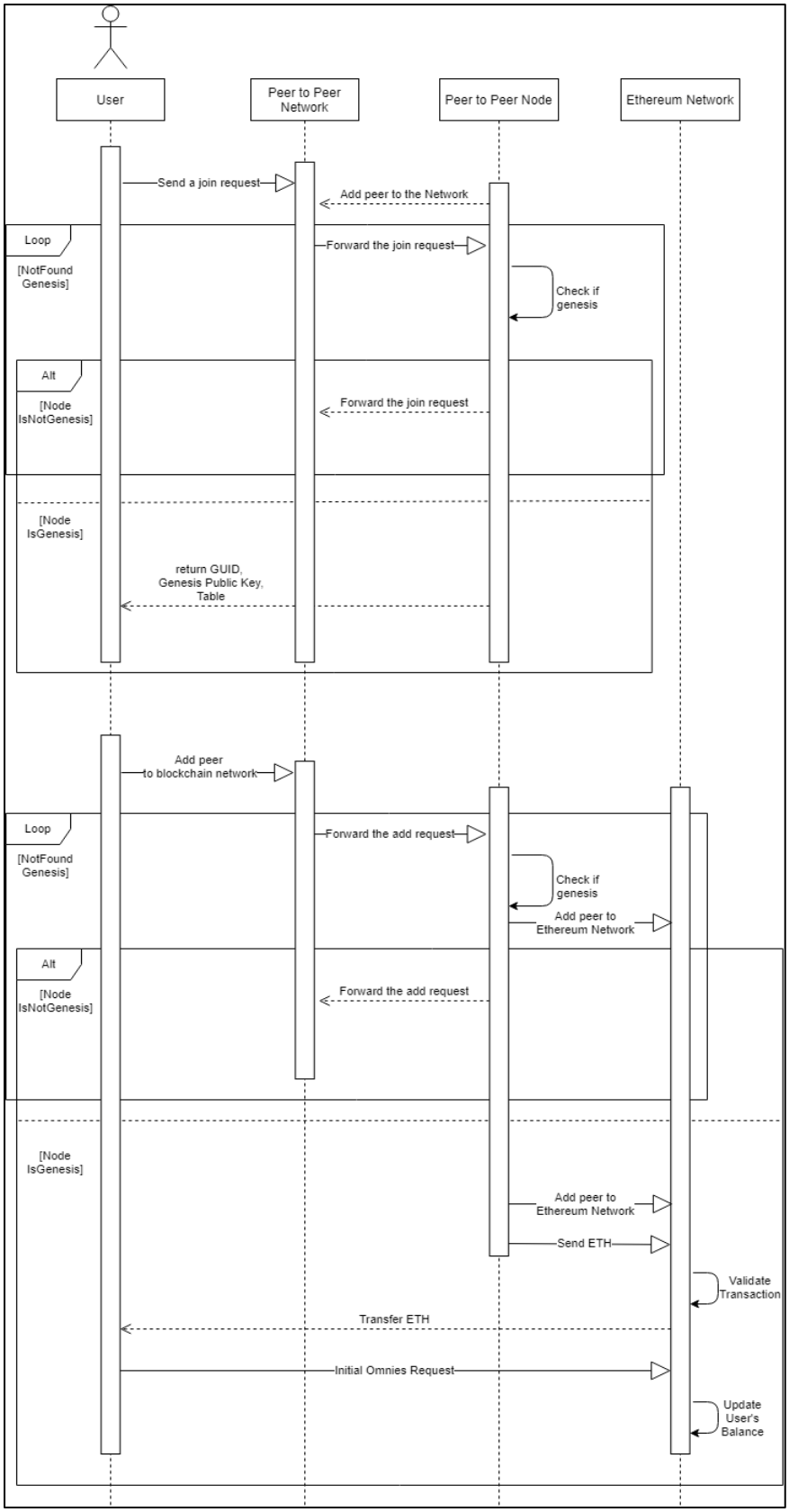

Figure 4.3 - Join Sequence Diagram

In part one, upon reaching the genesis node, it responds to the new user by sending the GUID, genesis public key, and a list of peers to add to their routing table. The user is now officially a part of the Peer-to-Peer network and may now begin the process of joining the Blockchain network as well.

The way this is performed is explained in part two, where the node uses the genesis public key it received earlier to be able to interface with the new network, the node then sends a request to be added by other nodes in the network, this request bounces between the peers whilst adding the node as Blockchain peer on every bounce. Upon reaching the genesis node, it adds the new node as a peer, and sends it some Ether so it can perform transactions.

This Ether transaction gets validated by the signers and the Ether is thus transferred. The new node responds to this by sending a request for an initial amount of Omnies, which gets sent from the genesis.

The following figure represents the sequence diagram pertaining to a user uploading a file. First, the user sends an upload file request to an available node, and the system checks if this node has enough space to handle the file's size.

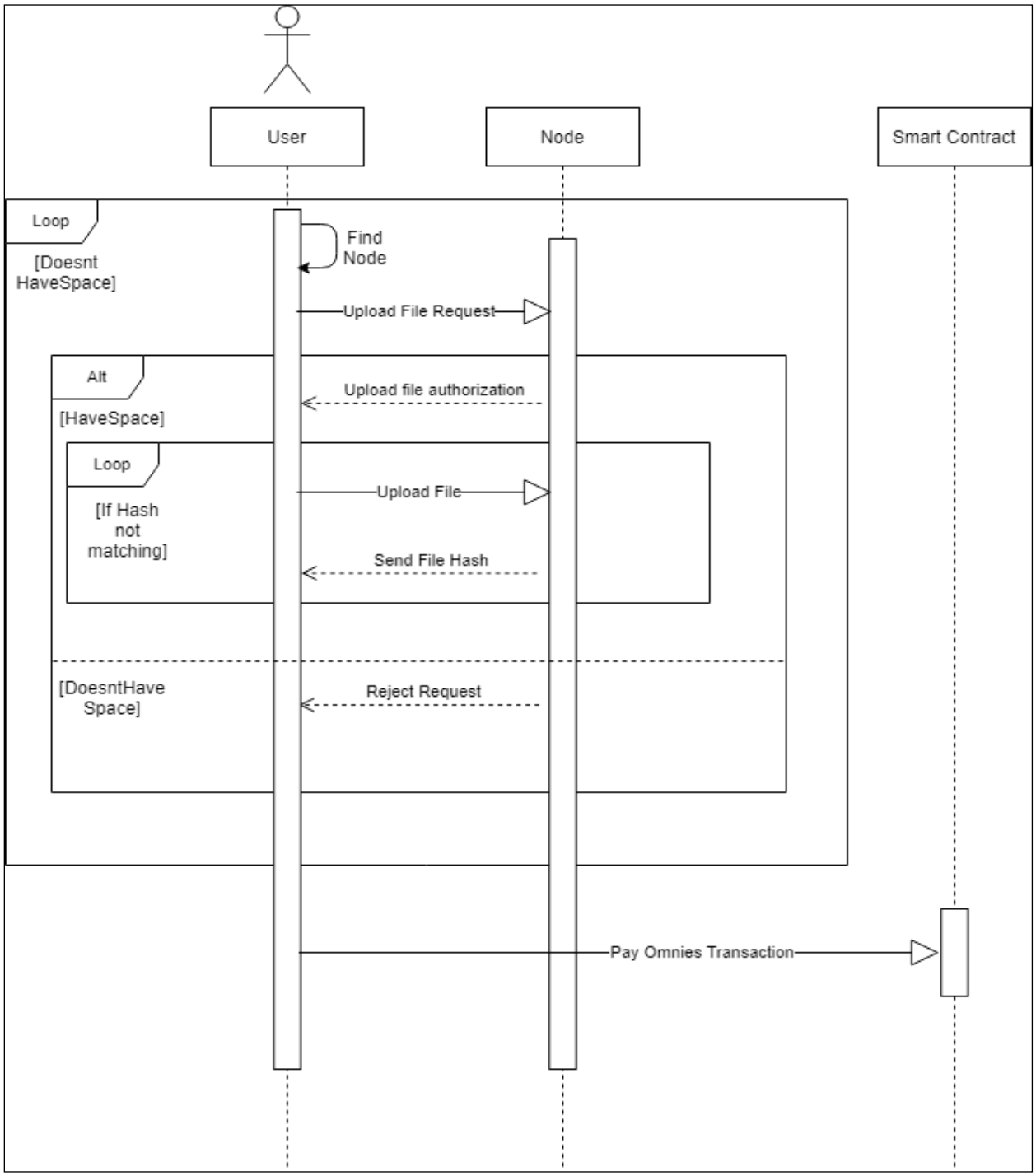

Figure 4.4 - Upload Sequence Diagram

First, the user sends an upload file request to an available node, and the system checks if this node has enough space to handle the file's size. If not, the system automatically tries to find another available node, else, the user uploads the file. After, the node sends the file's hash to the user. The system checks if the hash is matching on both ends. If it is not, the user keeps on uploading the file until both hashes match then an Omnies transaction is issued from the user to the smart contract.

Once uploaded, the file is distributed to multiple nodes which get rewarded for the storage and preservation of the file, the following figure depicts rewarding nodes for their availability and file preservation.

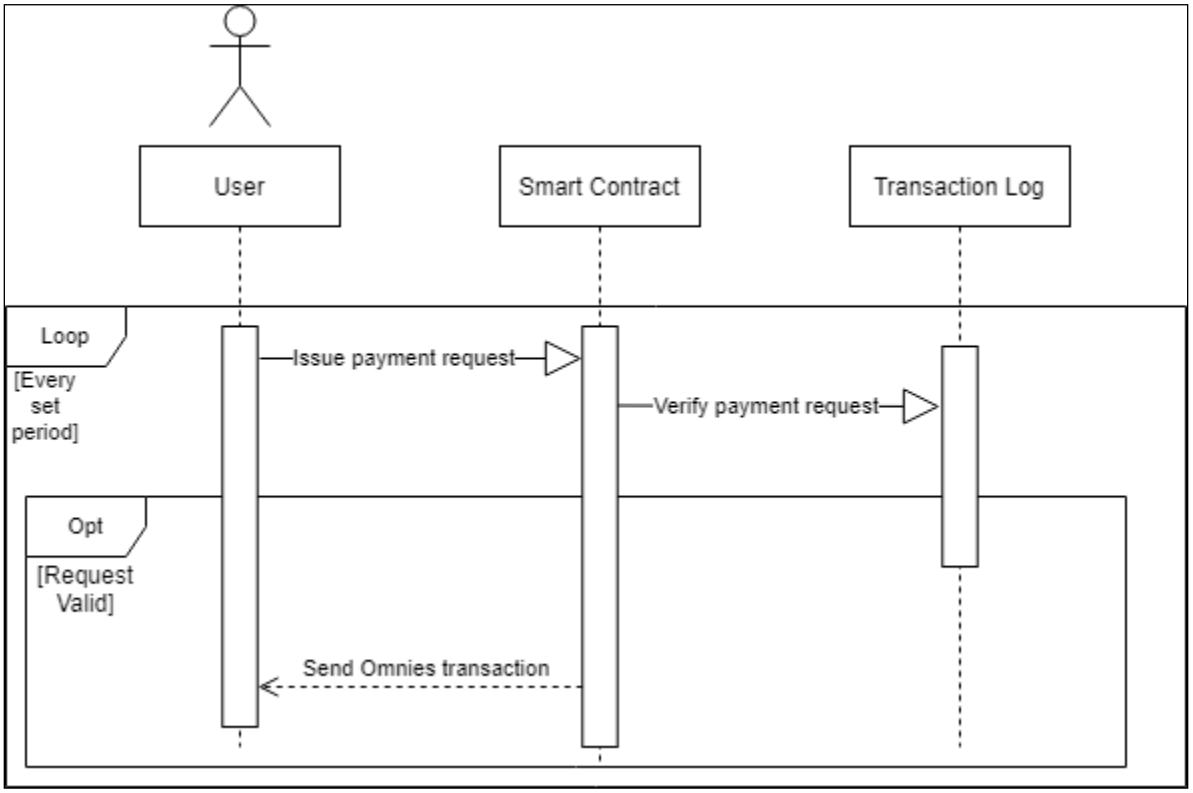

Figure 4.5 - Fund Sequence Diagram

To receive payments in the form of Omnies, the system automatically issues payment requests periodically, these requests are sent to the smart contract which has the job of verifying the requests to make sure they are valid and calculates the amount of Omnies to be paid out, the

verification process relies on utilizing the transaction log of the Blockchain which is untampered. If the request is valid, Omnies are transferred to the user accordingly. Users that stored a file can later retrieve it from the network. The process is described in the following Sequence Diagram.

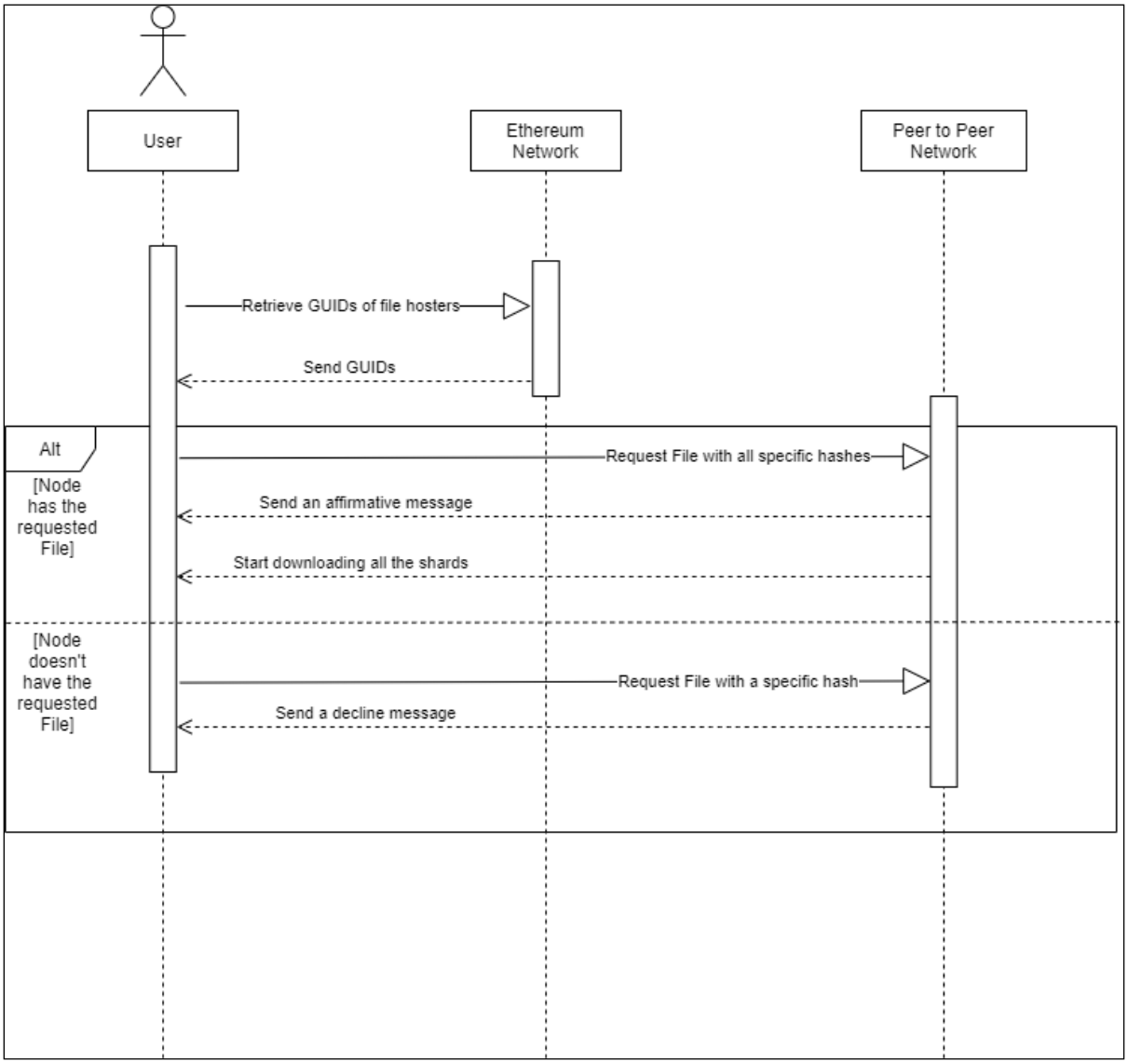

Figure 4.6 - Retrieve Sequence Diagram

First, the user retrieves the GUIDs of the requested file's hosters from the Ethereum™ Network. After that, the user requests the file with specific hashes from the Peer-to-Peer Network. If the hash is not available, the Peer-to-Peer Network sends a decline message informing the user that the file is not available at that moment. If the hash is available, the Peer-to-Peer Network sends an affirmative message to the user that the file is downloading.

4.4 – Workflow Diagram

This diagram presents the way that the solution is going to operate step by step while the user is using it. In the first part, this diagram provides us with the steps that system goes through when a user is joining the network.

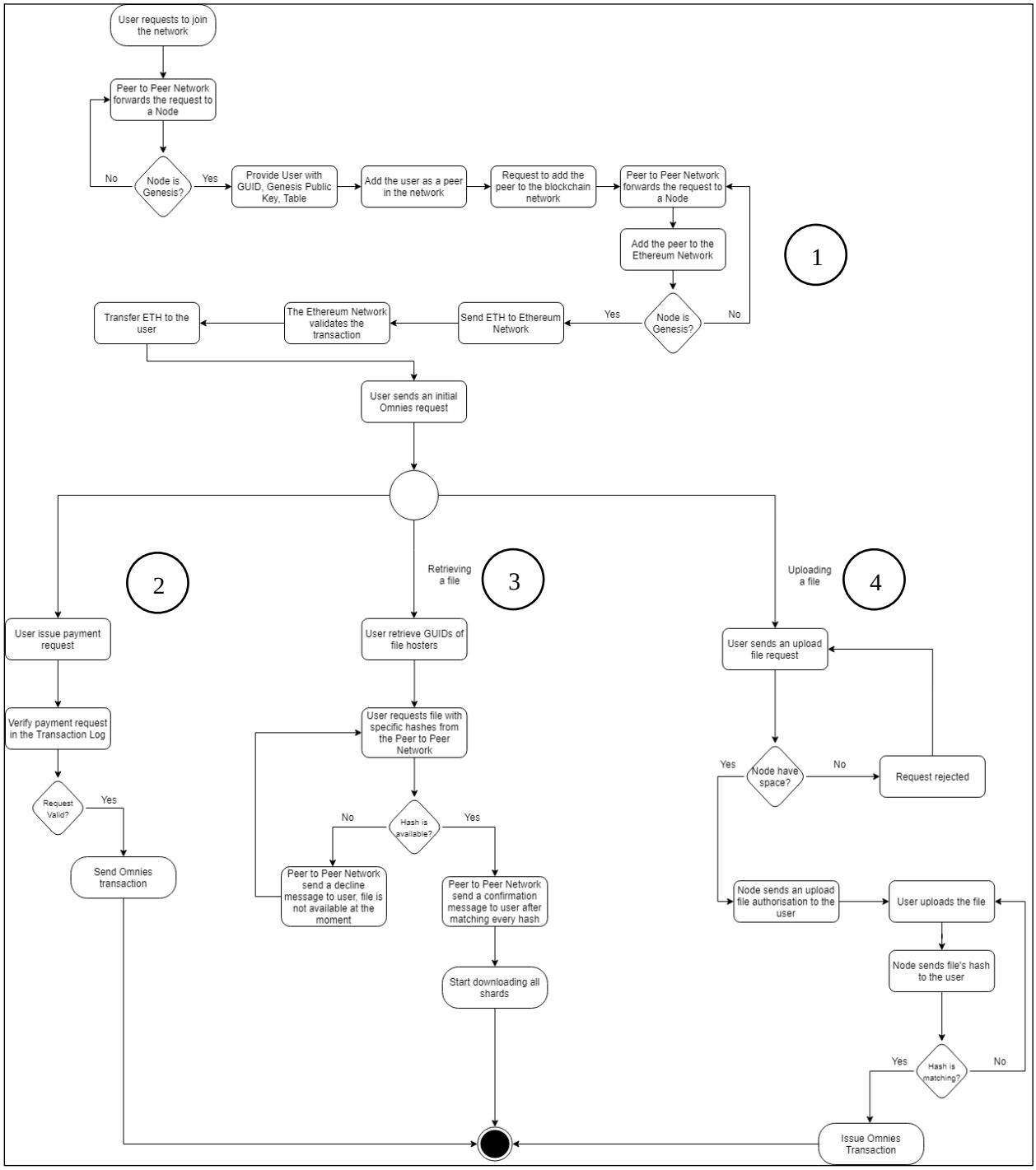

Figure 4.7 – Workflow Diagram

Moreover, this diagram displays the three features that are included in the solution (uploading a file [part four], retrieving a file [part three] and issuing a payment request [part two]) and how each feature is handled by the backend. Further explanation of these three features can be found in the respective sequence diagram along with the following implementation chapter.

This chapter introduced the system architecture of “OmniCache”, including the different use cases of the system, a few sequence diagrams to clarify how some of the system features are implemented and a context diagram to display the communication between the entities in the system. The next chapter talks about the implementation phase.

CHAPTER 5

IMPLEMENTATION

Page

Previously, the system architecture of “OmniCache” was revealed using different diagrams that describe the role of each component, module, and actor. Now, the methods in which the different aspects of the system were implemented are discussed.

5.1 - Peer-to-Peer Implementation

To be able to program a system that consists of multiple parts that need to be reused, an Object-Oriented approach is taken. Nodes in the system are made of two nodes that work together, a Peer-to-Peer (P2P) node and a Blockchain node, the latter being associated by composition with the P2P node. The Python socket library is used to implement the P2P logic.

In the P2P Node the following data members exist:

- bNode -> Blockchain node object
- myip -> current IP address of the local machine (string)
- port -> port number on which the node is listening (int)
- guid -> Global User ID of the node (int)
- peers -> a dictionary mapping guids to the IP and port of peers (guid: [IP, port])
- keys -> a dictionary, mapping keys to guids (int: bytes)
- startTime -> Time at which the node was launched
- chunkSize -> Size of a chunk in bytes (int)
- replicationFactor -> an integer indicating the number of replicated chunks while uploading
- protocol -> a dictionary mapping protocol codes to function pointers (str: pointer)

5.1.1 - Protocol

To enable communication between nodes, a protocol built on top of TCP/IP that can communicate using low-level socket programming had to be designed.
Network protocols are needed because they include mechanisms for devices to identify and make connections with each other, as well as formatting rules that specify how data is packaged into messages sent and received.
The following is the format that used in most of the protocol:
It is worth nothing that exceptions are discussed separately in later sections.
[ProtocolCode] – [Message]
ProtocolCode would be of four bytes

Message 4092 bytes

Here are the protocol codes and their uses:

'JOIN'

The ‘JOIN' request is sent by clients when they want to join the network, if the first recipient of this request is not the genesis node, then the request is forwarded until it reaches the genesis.

[JOIN]-[IP-port]

‘RJON’

The ‘RJON’ request is sent by a returning client that wants to rejoin the network, if the first recipient of this request is not the genesis node, then the request is forwarded until it reaches the genesis node.

[RJON]-[guid-IP-port]

'ADPR'

After a join request is received by the genesis node, in turn it broadcasts an ADPR request to all existing peers in the routing table (peers) with a new GUID, when the peers receive that request, they then add it to their own routing table.

[ADPR]-[guid-IP-port]

'DEFS'

After Broadcasting the ADPR request, the Genesis Node then returns to the joining peer a DEFS message so that the new peer can define itself on the network with the included GUID, routing table and the genesis public key.

Once the message is received by the peer: GUID is set, the routing table is copied and then it starts establishing shared secret keys with the peers in the routing table. After establishing secret keys with all peers using Diffie Helman key exchange algorithm, the peer broadcast an ‘ADBN’ request.

[DEFS]-[guid-GenesisPublicKey-table]

'ADBN'

Once the peer has defined itself, it broadcasts an ADBN request to all peers, containing its public key and Enode, so that it can be added a blockchain peer. When received peers extract the information from the request and add the sender as peer in the Blockchain and the genesis sends them a starting Ethereum™ to start.

[ADBN]-[PublicKey- Enode]

'RSCM'

When establishing secret keys with all peers using Diffie Helman key exchange algorithm, peers send their GUID and three numbers P, G, x, these are discussed further in the chapter to come. Once received these Parameters are extracted and used to generate the shared secret key.

[RSCM]-[GUID-P-G-x]

'PING'

The ‘PING’ request is used to test connections with peers, when sent, time is logged at t0, then when a pong reply is received, time is logged again at t1. Finally, the latency between these two peers can be calculated by performing (t1-t0)

[PING]-[ping/pong]

'UPFL'

When a peer wants to upload a file to the network, the file is split into chunks and sent to random peers, each chunk has an ChunkID associated with it that is used only to acknowledge it being received on the other end. Chunks and their ID are encrypted using the key already established with the peer (more detail on the encryption in the next chapter) resulting from that encryption, a nonce is generated and sent as well. Once received a peer would decrypt the message using the nonce and the established key, extract the chunk and its ID, acknowledge that he received the chunk and then saves the chunk to his hard disk with its hash as its name.

[UPFL_GUID]-[[chunkID-chunk][]nonce]

'DWFL'

Once the user wants to retrieve a file he uploaded previously, metadata about the chunks pertaining to that specific file are retrieved from the blockchain, then a ‘DWFL’ request is sent to each peer that has a chunk needed with the hash of the file. If the chunk hash matches with one of the chunks hosted on the peer’s computer, he then replies by sending the encrypted chunk with a ‘GTFL’ code. Once the chunk arrives at the other end it is decrypted then saved, when all the chunks are downloaded, they are merged in the correct order to reconstruct the original file.

[DWFL_GUID]-[[chunkHash][]nonce]

[GTFL]-[[chunk][]nonce]

5.1.2 - Upload File

The upload functionality allows users to select a file and upload it to a network of peers. The following procedure is initiated when uploading.

Once the file path has been set, a random fileID is generated to act as a reference for the original file. The file is read in byte mode, one chunk at a time (chunk size is set by default to 3704 bytes), then a chunkID is generated which is used to acknowledge receiving the chunk. Taking into consideration the replication factor X (set to 1 by default) the chunk is uploaded X times to different peers on the network. Peers are chosen at random and before initiating communication with them they are sent a heartbeat message 'PING' to test the connection with them. After a Peer is selected, the chunk is encrypted using the shared key between the two parties and the uploaded to them with the 'UPFL' code once received the receiver replies by sending an acknowledgement message.

After the chunk is acknowledged the chunk's hash, order, fileID, sender and receiver are logged to the blockchain. After all the chunks have been uploaded, the original file size, name, hash and fileID are also logged to the blockchain.

5.1.3 - Download File

Users can download files that were previously uploaded. Supplied by the fileID and name, the information about the chunks pertaining to this specific file are retrieved from the blockchain. ChunkHash, senderGUID, receiverGUID and chunkNumber are then used to contact peers having receiverGUID as their GUID and request the chunk which name is identical to the Chunk Hash extracted from the blockchain. This is done concurrently to speed up the download process with a max of three chunks downloading at the same time. Any missing chunks are re-downloaded so that all the chunks become available locally after which they are merged into a file thus reconstructing the original file having the same hash value.

5.1.4 - Peer Handling

Peers can simultaneously listen on a specific port and connect to other peers. To be able to handle peers randomly connecting and disconnecting, an event driven approach has been implemented so that each connection is handled in a different thread.

When a node is initialized, it starts listening on port 4444 by default, and each time a connection is accepted a thread is spawned to handle the request that came with the new connection.

```
while not self.turnoff:
	try:
		clientSocket, address = inbound.accept()
		thread=threading.Thread(target=self.handlePeer,args=[clientSocket],dae
	mon=True)
		thread.start()
	except:
		self.logging("* error in connectionsSpawner : was not able to spawn a
		connection to {}".format(address))
		continue
```

After Handling the peer's request, the thread is killed, and the connection closed. This way, a peer would not have a large number of threads constantly running at the same time, thus efficiently making use of the node's CPU.

Having protocol codes directly mapped to their actual functions enables us to process or drop peer requests.

```
self. protocol={
		'JOIN':self.join,#JOIN code
		'UPFL':self.upfl,#UPFL code :  upload file
		'ADPR':self.adpr,#ADPR code add peer when new peer joins
		'DEFS':self.defs,#DEFS code define self
		'ADBN':self.adbn,#ADBN code add blockchain node
		'RSCM':self.rscm,#RSCM code establish shared secret key with peer
		'PING':self.ping,#PING request to test connection and latency
		'DWFL':self.dwfl}
```

```
#-------------------------------------------------------------------------------
----
    def handlePeer(self,clientSocket):
#-------------------------------------------------------------------------------
----
        host, port = clientSocket.getpeername()
        self.logging("* Connection to {} has been esstablished".format(str((host,po
rt))))
        '''
        handle peer depending request/ProtocolCode
        '''
        peerconn = PeerConnection(host,port,self.
startTime,sock=clientSocket,Keys=self. keys)

        try:
            protocolCode, data = peerconn.recvdata()
            if protocolCode:
                protocolCode = protocolCode.upper()

            if protocolCode in self.protocol:
                self.protocol[protocolCode](peerconn , data)
            else:
                self.logging("{} code not recognized ".format(protocolCode))
        except Exception as e:
            print(e)
            self.logging("* error while handeling peer {}:{} with code {}".format(h
ost,port,protocolCode))
```

Later, if a peer were to send a ‘PING’ request ([PING]-[ping]), peerconn. recvdata() would yield ‘PING’ as protocolCode and “ping” as data.

Executing: self. protocol[protocolCode](peerconn , data)

would call the following function:

```
def ping(self,peercon,data):

        peercon.sendData('ping','pong')
```

5.1.5 - Cleaning

To make use of the node's hard drive space efficiently, chunks that are not valid anymore need to be deleted. To be able to periodically check for unnecessary files and delete them a thread had to be created to be responsible for comparing local hosted files with the Blockchain logs pertaining to the node's GUID and delete all files that are not valid but hosted locally in the node's storage space. This is done by filtering the logs by the receivers GUID which yields a list of files that should be hosted on the node, after which all files not in the list are deleted.
After cleaning the hosted files, the node requests payment from the contract for the total size of files hosted.

```
  def cleanHosted(self, omniesLabel, dataLabel):

        '''
        Delete files that are no longer valid
        '''
        while True:
            time.sleep(2*60)
            print("cleaning")
            hostedFiles = self.getHostedFiles()
            if not hostedFiles:
                continue
            else:
                toNotDelete = self.bNode.filterByRGUID(self.guid, hostedFiles)#hash
es
                #print(toNotDelete)
                print('started Deleting')
                items = list(set(toNotDelete + self.hold))
                self.DeleteFiles(items)
                self.hold.clear()
                if len(self.getHostedFiles()):
                    self.bNode.requestPayment(len(self.getHostedFiles())*self.chunk
Size)
```

5.2 - Blockchain Implementation

The Blockchain system builds on top of the groundwork set by the P2P node and is a core part of OmniCache. It consists of multiple components that need to work together to achieve the functionalities required including joining, uploading, retrieval, payment issuance, and auditing. The main components being the smart contract, Go Ethereum™, and the bcNode class.

5.2.1 - Smart Contract

The smart contract is written in Solidity and implemented using Remix as an IDE. It takes on multiple roles, primarily acting as the central figure that nodes must interface with for all functionalities.

5.2.1.1 - SafeMath

The contract object “Bank” begins with a using-directive to bind the SafeMath library to uint256 variables. The SafeMath library is implemented in the smart contract with the goal of ensuring no integer overflow occurs when performing arithmetic operations on Omnies.
It consists of two methods that take two uint256 values each.
- sub -> Returns the difference.
- add -> Returns the sum.
The methods are marked as both pure, meaning they do not read or modify the state, and internal to the library:

5.2.1.2 - Bank

The object consists of several data members:
- waitPeriod -> The wait period between payments (constant (uint)).
- enrollPayment -> The initial amount of Omnies to be paid at enrollment (constant (uint)).
- dataRate -> The price per chunk, to be paid to the hosting nodes (constant (uint)).
- fileCost -> The price per chunk, to be paid by the uploading nodes (constant (uint)).
- chunkSize -> The maximum size of one chunk, used in payment calculations (constant (uint)).
- balances –> Omnies balances of all accounts (mapping (address => uint256)).
- isEnrolled -> To check if account is already enrolled (mapping (address => Boolean)).
- lastPaid -> Keeps track of the last time an account was paid in Unix time (mapping (address => uint)).
- owner -> Owner of the smart contract (address).
- totalSupply_ -> Total supply of Omnies (uint256).
Bank also defines three events that are explained later.
The constructor is called upon contract deployment and serves to initialize the totalSupply, the owner, and their balance. This creates the Bank with an initial amount of Omnies.

The rest of the contract consists of public methods that do specific tasks:
Enroll -> If the account has not been enrolled before, enrolls a new account into the Bank, and awards them an initial amount of Omnies.
- giveOmnies -> Performing payment auditing issuance.
- myBalance -> Returns the Omnies balance of the account.
- totalSupply -> Returns the total supply of Omnies.
- uploadFile -> Deduct Omnies and emits event.
- uploadChunk -> Emits event.
- deleteFile -> Emits event.
- getEnrolledStatus -> Check if already enrolled.

Note that some of the methods are marked as view meaning they don't modify the state of the Blockchain and as such can be called without gas.

5.2.1.2.1 - Events

Events are signals that are fired during specific function calls and allow us to log data in the Blockchain in an orderly fashion.
Three events are defined:

```
event logFile(address indexed accountAddress, int linkToOGF, string fileName,
string fileHash, int totalSize);
```

This event logs data relevant to the file being uploaded, including the uploader's public address, a nonce used to link the individual chunks to the file, the name of the file, its hash, and size.
Emitted by the uploadFile function.

```
event logChunk(address indexed accountAddress, int indexed linkToOGF, int
senderGUID, int indexed receiverGUID, string chunkHash, int chunkNb);
```

This event logs data relevant to each chunk getting uploaded, including the uploader's address, the nonce mentioned earlier, the sender and receiver GUIDs, its hash, and number.
Emitted by the uploadChunk function.

```
event logDeletion(address indexed accountAddress, int indexed linkToOGF);
```

This event logs data relevant to file deletion, including the uploader's address and file nonce. Emitted by the deleteFile function.
This concludes the smart contract implementation.

5.2.2 - Go Ethereum™

Go Ethereum™ is an implementation of the Ethereum™ protocol, written in Go, and available as a standalone client called Geth. Geth is used as the foundation and home of the private blockchain network that OmniCache functions on. It is utilized using shell commands issued by the bcNode class.

5.2.3 - BcNode

The main class of all Blockchain nodes.

It is important to first mention that there are two versions of the Blockchain nodes, one relevant to all normal accounts, and one relevant to the genesis alone.

This distinction must be drawn because the genesis does not upload/retrieve any files and is only there to launch the initial creation of the Blockchain. Consequently, there exists two different implementations of bcNode which will be referred to from now on as bcNode_C (normal node implementation) and bcNode_G (genesis implementation).

Several data members exist in common between both nodes:
- pubKey -> Current account's public key (string)
- web3 -> Web3.py class object
- ip -> Current IP address of the local machine (string)

bcNode_C has additional data types:
- enode -> Current account's enode, used for peer addition (string)
- contract -> Smart contract class object
- exists -> Specifies whether the data directories exist or not (Boolean)
- passPhrase -> Temporary storage of the pass used to unlock the account (string)
- proc -> Process object that stores the Geth process that runs the Blockchain

bcNode_G has an additional data type:
- txNonceCount -> Used to internally keep track of transaction counts (int)

5.2.3.1 - On Start

Upon running the program, the directory is scanned to check for existing Blockchain data directories. New bcNode_C nodes are presented with the option to either create a new account or import an existing one using a previously created keyfile. On the other hand, existing nodes are allowed to rejoin. A passPhrase must be supplied in all cases to unlock the account. The entire start-up process of bcNode_C nodes may be summarized using several methods implemented in the class.
These methods are:
- dataDirsExist -> Checks if data directories exist
- createAccount -> Launches the account creation process
- importAccount -> Launches the import account process
- createGenesisJson -> Creates the genesis. json file
- preRunInit -> Performs pre-run initialization of the Blockchain node
- runExistingNode -> Runs the Blockchain node in a Geth instance
- postRunInit -> Performs post-run initialization of the Blockchain node
- validatePass -> Attempts to unlock the account with the supplied pass phrase

A different execution flow exists for each of creating a new account, importing an existing account, and running an existing node with an existing account.

5.2.3.2 - Account Creation

For both types of nodes, account creation begins with creating a new temporary file to store the entered password in. For normal nodes, this file is deleted immediately after account creation and only exists to be passed to Geth.

Following this, a new subprocess is created to run Geth with a specific command.

```
command = 'geth account new --datadir ./ETH/node --password tmpPass'
```

This command tells Geth to create a new blockchain account with /ETH/node as the path to store data in. The temporary file created is passed using the –password option.

The standard output and error are redirected for logging purposes to a log file located in a relative path at /logs/blockchain/initLog.txt.

5.2.3.3 - Account Import

This functionality is performed using the importAccount method. New users may opt instead to import an existing account they had by providing a keyfile generated from a previous account creation. This keyfile begins with "UTC" and is in the keystore folder in the node's data directories and is necessary for imports. The keystore is copied into the new account.

5.2.3.4 - Execution Flow for New Nodes

After creating or importing the account, createGenesisJson, preRunInit, runExistingNode, and postRunInit are executed in that order.

5.2.3.5 - Execution Flow for Existing Accounts

If the node and account already exist, then runExistingNode and postRunInit are called immediately instead.

5.2.3.6 - Genesis Json File

This functionality is performed using the createGenesisJson method. This file is necessary to create, join, and synchronize the Blockchain and as such it needs to be the same across all nodes. It consists of many fields, but the important ones are discussed below.

- "chainId"

This is the network ID and must be set to a number that is not already used by an Ethereum™ network to avoid picking up peers from other networks.

- "clique"

Set to specify the use of a PoA consensus algorithm.

- "extradata"

Set to specify the initial signer's addresses (In this case, it is the genesis's address).

- "alloc"

Used to give the genesis node an initial allocation of Ether.

After creating this file, the node may be initialized with a call to preRunInit.

5.2.3.7 - Node Initialization

After account creation, the public key for new account can now be read using the names of the key files.

The key files are read using:

```
keyFiles = [filename for filename in listdir('./ETH/node/keystore/') if
filename.startswith("UTC")]
```

and the public key is then extracted using:

```
self. pubKey = "0x" + keyFiles[0]. split("--")[2]
```

Now that the public key is obtained, it is time to create the genesis json file.

This file is necessary to create, join, and synchronize the Blockchain and as such, it needs to be the same across all nodes. It consists of many fields, but the important ones are discussed below.

- "chainId"

This is the network ID and must be set to a number that is not already used by an Ethereum™ network to avoid picking up peers from other networks.

- “clique”

Set to specify the use of a PoA consensus algorithm.

- “extradata”

Set to specify the initial signer’s addresses (In this case, it is the genesis’s address).

- “alloc”

Used to give the genesis node an initial allocation of Ether.

After creating this file, the node can now be initialized with Geth using subprocess.

The initialization command is as follows:

```
command = 'geth init --datadir ./ETH/node ./ETH/genesis.json'
```

"init" tells Geth to initialize the node located at the data directory specified by –datadir with the path to genesis. json supplied after.

5.2.3.8 - Running the Node

This functionality is performed using the runExistingNode method. A command to run Geth is supplied in a new subprocess in a new non-daemon thread. This command is be explained later in depth. The subprocess is ran using Popen to save a process object for later use. The object is stored in the proc data member. STDOUT and STDERR is redirected to an instance of the logPipe class. This is explained later as well. This concludes the start-up system for bcNode_C nodes.

5.2.3.9 - Genesis Node Startup

In contrary to normal nodes, the genesis node runs only once and as such has no need for the ability to import a node. The startup process is relatively the same as bcNode_C nodes apart from running the node. To run a bcNode_G using Geth, the following command is executed in a new subprocess (The command is a more complex version of the normal bcNode_C nodes):

```
command = 'geth --datadir ./ETH/node --syncmode=full --cache=2048 --networkid 15 --
port 30305 --nat extip:{0} --mine --unlock {1} --nodiscover --password {2}'.
format(self.ip, self.pubKey, "tmpPass")
```

Geth is told to run the node specified at –datadir with several options.
The important ones are explained below:
--syncmode -> "Full" is chosen instead of the default value "Fast" due to synchronization errors that would generate when the number of blocks mined reached ~100 during testing.
--cache -> 2048 bytes worth of cache is provided to improve performance.
--nat -> Used to relay the IP to other peers
--mine -> To run as a signer, this option is only specified while running bcNode_G
--unlock -> To unlock the signer's account to validate transactions, this option is only specified while running bcNode_G
--password -> To specify the password file for –unlock, this option is only specified while running bcNode_G

--nodiscover -> Used to prevent the nodes from being discovered automatically, peer addition is instead handled manually.

5.2.3.10 - Post Run Initialization

This functionality is performed using the postRunInit method.
Several objects and variables must be initialized following running the node.
After account creation, the public key for new account can now be read using the names of the key files. The key files are read using:

```
keyFiles = [filename for filename in listdir('./ETH/node/keystore/') if
filename.startswith("UTC")]
```

and the public key is then extracted using:

```
self.pubKey = "0x" + keyFiles[0].split("--")[2]
```

What follows is a set of initializations relevant for interfacing with the Blockchain and smart contract.
The web3 object is first initialized using:

```
Web3(Web3.IPCProvider())
self.web3.middleware_onion.inject(geth_poa_middleware, layer=0)
```

This is used to connect to the running Geth instance to interact with it.
Now, if this is bcNode_C, the enode is also initialized with:

```
self.enode = self.web3.geth.admin.node_info()["enode"]
```

The pubKey is now transformed to its checksum value, for both nodes, to send transactions:

```
self.pubKey = self.web3.toChecksumAddress(self.pubKey)
```

What follows now is node dependent:

- For the bcNode_G, the contract needs to be deployed. After compiling the smart contract, its ABI and Bytecode are extracted and hardcoded into the bcNode_G script. This is done since the smart contract is deployed only once. Note that the ABI must be loaded in json form.

Using these two values, the contract object can be initialized:

```
contractInterface = self.web3.eth.contract(abi=abi, bytecode=bytecode)
```

The contract's constructor is then called to deploy the smart contract with a large initial amount of Omnies as an argument:

```
tx_hash = contractInterface. constructor(99999999999999).transact()
tx_receipt = self. web3.eth.waitForTransactionReceipt(tx_hash)
```

The transaction is then waited for to retrieve the contract's address to publish it to bcNode_C nodes.

- For the bcNode_C, the contract address and ABI are provided to initialize the contract object

```
self. contract = self. web3. eth. contract(address=contractAddress, abi=abi)
```

Following this, the program attempts to unlock the account.

5.2.3.11 - Account Unlock

This functionality is performed using the validatePass method.

The default account is then set and unlocked to begin transacting:

```
self.web3.eth.defaultAccount = self.web3.eth.accounts[0]
self.web3.geth.personal.unlock_account(self.web3.eth.accounts[0], self.passPhrase,
0)
```

Finally, the value of passPhrase may be cleared from memory.

5.2.3.12 - Peer Addition

Exists in both node types. Manual peer addition is performed by passing the new peer's enode to the addToNet function with:

```
self.web3.geth.admin.add_peer(enode)
```

5.2.3.13 - Initial Ethereum™ Transaction

New nodes that join the Blockchain network need to receive an initial amount of Ether to send transactions. This is performed by bcNode_G only using the sendETH function.
First, the genesis node account's private key is decrypted by reading the keystore and supplying the preset password.
Then, the txNonceCount variable gets initialized to the transaction count and is used as a nonce in building the transaction to be sent. This transaction is then signed using the private key decrypted earlier and sent:

```
signed_tx = self.web3.eth.account.signTransaction(tx, private_key)
```

Checks are performed to ensure it is received.

5.2.3.14 - Interacting with the Smart Contract

bcNode_C nodes interface with the smart contract using several functions that call different smart contract functions.
- enroll -> Calls the enroll function in the smart contract.
- requestPayment -> Calls the giveOmnies function in the smart contract
- getOmnies -> Calls the myBalance() function in the smart contract
- isEnrolled -> Calls the getEnrolledStatus() function in the smart contract
- logFileUpload -> Calls uploadFile
- logChunkUpload -> Calls uploadChunk
- logDeletion -> Calls deleteFile

5.2.3.15 - Event Filtering

To retrieve the data that gets stored in the event logs of the Blockchain, several functions were implemented that rely on filtering. Filtering is performed using the createFilter function in the Web3.py object. This function is instructed to read all events from the first block. Each of these filters can be also provided with an indexed smart contract variable to filter for specific events only.

Each event has specific variables marked as indexed, to allow for easier filtering. For example, filterByAddress events are filtered with the account's address as a filtered argument.

```
self.contract.events.logFile.createFilter(fromBlock=0,
argument_filters={'accountAddress':self.web3.eth.defaultAccount})
```

The events are then retrieved using get_all_entries() and for each of these events the transaction receipt is obtained and processed to retrieve the logged data.

```
for event in event_filter.get_all_entries():
            receipt = self.web3.eth.getTransactionReceipt(event['transactionHash'])
            result = self.contract.events.logFile().processReceipt(receipt)
```

Then the data can be accessed using:

```
result[0]['args'][‘arg_name’]
```

The filtering functions are as follows:

- filterByAddress -> Read logFile events to retrieve files uploaded by a specific address
- filterByFile -> Reads logChunk events to retrieve chunks for a specific uploaded file
- filterByRGUID -> Reads logChunk events and returns a list of chunks that have been deleted but still exist on the host's machine
- isFileValid -> Reads logDeletion events and checks if the file has been deleted

5.2.3.16 - Synchronization Check

The core of the functionality implemented needs to interface with the Blockchain for its needs. This requires that the chain to be synchronized properly, and checks to be performed beforehand. The checkSyncStatus method takes care of this by checking if the instance is connected to any peers:

```
if self.web3.net.peer_count != 0:
```

Followed by a check on whether Geth started syncing or not:

```
if (not self.web3.eth.syncing) and (self.web3.eth.block_number() != 0):
```

This ensures a proper level of chain synchronization.

5.2.3.17 - LogPipe

This class was created to properly log the Geth output. It creates a daemon thread that acts as a pipe and attempts to take in STDOUT and STDERR from the Geth process to write them to a logfile named runLog.txt using the logging library in Python. Since Geth outputs indefinitely as long as its running, a rotating file handler was used to manage the log file's size.

```
handler=RotatingFileHandler('./logs/blockchain/runLog.txt', maxBytes=8192,
backupCount=1)
```

The argument maxBytes determines the max size of the log file in bytes.

5.3 - Driver

The driver file launches all the necessary GUI Classes, P2P and Blockchain nodes, and handles the on-click events for all the buttons of the GUIs that are connected to functions that trigger all required functionalities from the P2P and Blockchain classes. The driver files contain a class called SystemTrayIcon that is used to minimize the software to the system tray to keep it running the background, so the user is able to use his computer in parallel while hosting other users' files.

5.3.1 - System Tray

```
class SystemTrayIcon(QtWidgets.QSystemTrayIcon):
	"""
	CREATE A SYSTEM TRAY ICON CLASS AND ADD MENU
	"""
	#-------------------------------------------------------------------------
	def __init__(self, icon, parent=None):
	#-------------------------------------------------------------------------
		QtWidgets.QSystemTrayIcon.__init__(self, icon, parent)
		self. setToolTip(f'OmniCache')
		menu = QtWidgets.QMenu(parent)	#Create a Qt widget with tray icon
	parent
		open_app = menu.
	addAction("Open OmniCache")	#Adding Open OmniCache in tray menu
		open_app.triggered.connect(self.open_omnicache)
		exit_ = menu.addAction("Exit")
		exit.triggered.connect(lambda: sys.exit())	#Adding Exit in tray men
	u

		#Created an instance of Join Network UI and Homepage UI
		self.Homepage_UI = None
		menu.addSeparator()
		self.setContextMenu(menu)
		self.activated.connect(self.onTrayIconActivated)
```

In PyQt, a widget called QSystemTrayIcon is used to create a system tray component for the software. Then, a QMenu, which is a tray menu holding a button that is open by right-clicking on the system tray on the computer, is utilized. The menu contains an “Open OmniCache” button and an “Exit” button that closes the software completely.

5.3.2 - P2P Node Creation

Upon launching the software, a GUI class called UI_JoinNetwork is created. The user needs to input the IP address of another node in the P2P network which is validated by regular expression. After clicking on Join Network button, a function called initNode is called. initNode creates a P2P node for the user by taking the target IP address from the input above and it creates a thread to handle the node connection spawner in parallel. After which, the Homepage UI appears.

```
#Initiliazing node on Join network click
#------------------------------------------------------------------------
def initNode():
#------------------------------------------------------------------------
    global Homepage_UI

    ip = Join_Network_UI.ipaddress_input.text()
    port = 4444
    bNode = bcNode("172.29.133.188")    #Initializing Bc Node
    node= Node("172.29.133.188",port,bNode, npeer=10)   #Initializing P2P Node
    tosend='-'.join(["172.29.133.188",str(port)])
    thread=threading.Thread(target = node.connectionSpawner, args = [], daemon = Tr
ue)
    thread.start()
    node.connectAndSend(ip, port,'join', tosend, waitReply=False)

    Homepage_UI = Ui_homepage(node)    #Creating Homepage UI with node as arg
    tray_icon.show()
    HomepageListeners()

    return Homepage_UI
```

A function was added in the driver that handles all the on-click events of all the buttons in the homepage UI called HomepageListeners().

5.3.3 - GUI

5.3.3.1 - Splashscreen UI

Ui_splashscreen is the first GUI class that is called upon launching the software. It shows the OmniCache logo for 2 seconds until the application is loaded.

5.3.3.2 - Join Network UI

Ui_JoinNetwork is the GUI class that is called when the user is launching the software for the first time on a specific computer. In this GUI, the user must input a target IP address which is the IP address of another node in the P2P Network, so that a node for his machine can be created.

Figure 5.1 - Join Network_UI

5.3.3.3 - File Item UI

Ui_file_item is a widget class, and not a window. An instance of this class is created for each item in the files' list in the home page. Each widget contains the file's name, download file button, and remove file button.

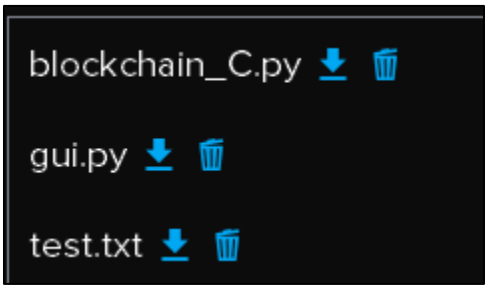

Figure 5.2 - Item_File_UI

5.3.3.4 - Homepage UI

Ui_homepage is a GUI class, and it is basically the main page of the software. First, it shows the users' Omnies balance, and the size of the files hosted for other users and includes a settings button for the user account management. Moreover, it displays a list of his uploaded files with an option to search by their names. At the bottom, it shows the selected file details (File's name and size).

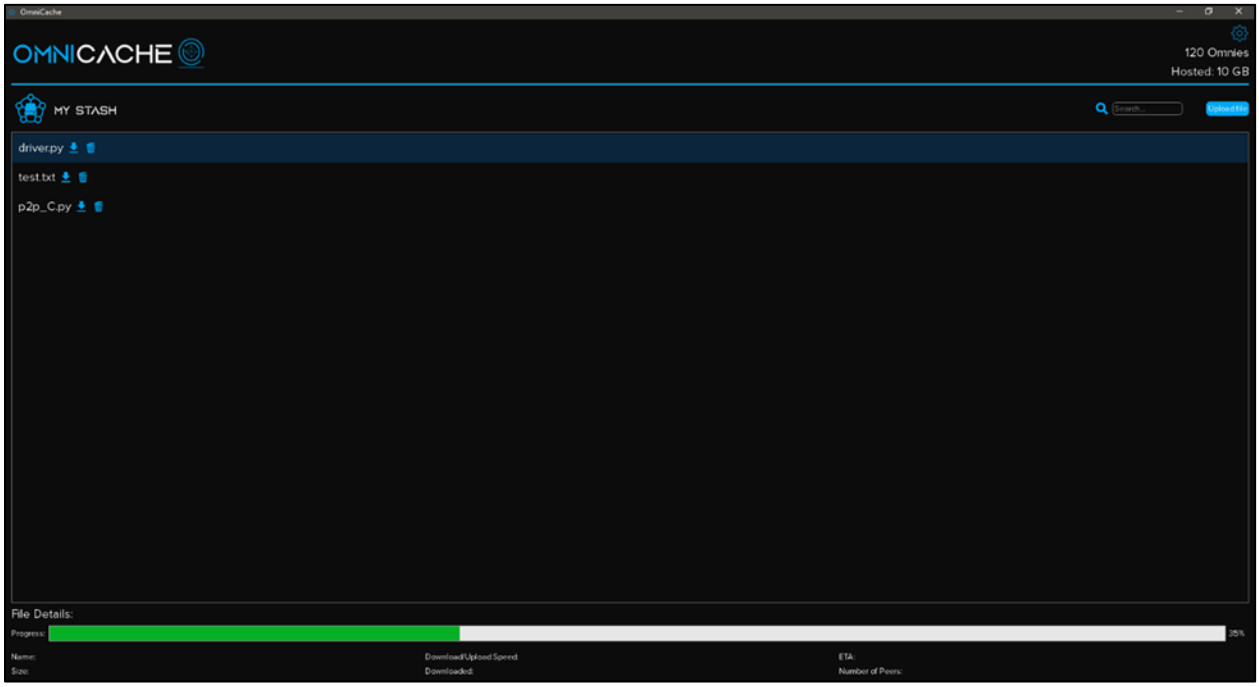

Figure 5.3 - Homepage UI

In this class, important functionalities in the software are implemented, like the upload function, and adding an item to the list after uploading it. First, by clicking on upload, a File Dialog opens, and the user needs to select the file that he wants to upload from his computer. It is required to run the upload task in a different thread to avoid a freezing GUI, and keep the application running. For this purpose, a thread class called UploadTask is created:

```
#Upload Task
class UploadTask(QtCore.QThread):
    #Task thread finished event
    finished = pyqtSignal(object)
    #-------------------------------------------------------------------------
    def __init__(self, node, filepath):
    #-------------------------------------------------------------------------
        QtCore.QThread. __init__(self)
        self.node = node
        self.filepath = filepath

    #When task thread starts
    #-------------------------------------------------------------------------
    def run(self):
    #-------------------------------------------------------------------------
        filename , _ = self.node.sendChunks(self.filepath)
        self.finished.emit(filename)
```

So, after getting the file path from the file dialog, a thread is created and the file path as well as the node instance are passed as an argument to the UploadTask constructor. In the UploadTask class, the run function is called after starting the thread. The "run" function is calling another P2P function sendChunks that is responsible of splitting the file to multiple chunks and uploading them to other users. After the task done, a pyqtSignal "finished" is emitted, which is an event used to inform the system that the uploading task is finished.

```
#On-Click Upload Button
    #-------------------------------------------------------------------------
    def upload_onclick(self):
    #-------------------------------------------------------------------------
        #To-do on clicking Upload Button

        filename = ""
        browseFile = QFileDialog()   # creating a File Dialog
        filename = browseFile.getOpenFileName(self,"Select File","",)   #Receiving a string
        from the file dialog

        #Extracting file name from filepath
        ntpath.basename("a/b/c")

        uploadtask = UploadTask(self.node, Path(filename[0]))    #Creating a thread
        uploadtask. finished. connect(self.addItemtoList)    #After thread is finished
        self.threads.append(uploadtask)
        uploadtask.start()
```

Upon finishing the uploading task, addItemtoList() function is called to add a widget item to the files' list.

```
#------------------------------------------------------------------------
    def addItemtoList(self, filename):
    #------------------------------------------------------------------------
        # If filedialog open is clicked
        if filename != "":
            #Adding an item to the QListWidget
            myQCustomQWidget = Ui_file_item()
            myQCustomQWidget.label.setText(filename)
            myQListWidgetItem = QListWidgetItem(self.listWidget)
            myQListWidgetItem.setSizeHint(myQCustomQWidget.sizeHint())
            self.listWidget.addItem(myQListWidgetItem)
            self.listWidget.setItemWidget(myQListWidgetItem,myQCustomQWidget)
```

5.3.3.5 - Settings UI

UI_settings is a GUI class that display the settings page as a popup.

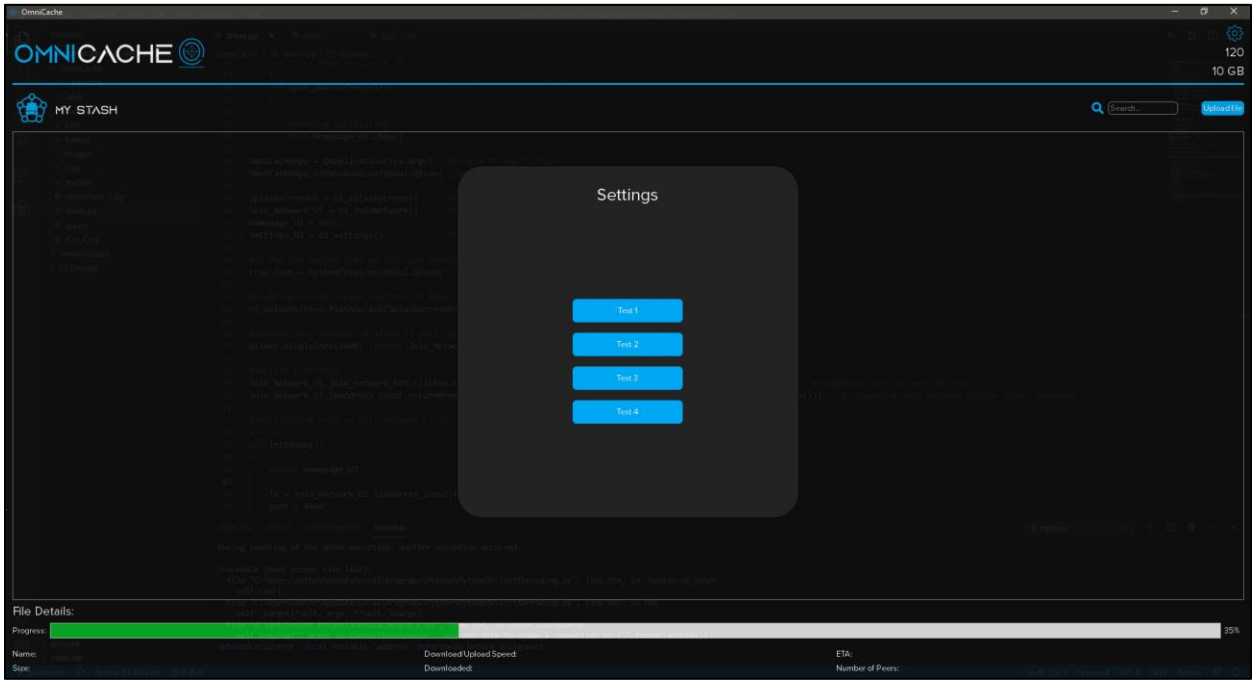

Figure 5.4 - Settings UI

CHAPTER 6

SECURITY

Page

Previously, the way in which different modules in the system were implemented was explained. In this chapter, there is a focus on the security aspects that can be used to protect data being handled on an unsecure network.

6.1 - Secure Communication

When a node joins the network the first thing it does is: generateKeys() which, as the name suggests, generates shared secret keys with each peer on the network. This is done using the Diffie-Hellman algorithm with some minor modifications. The algorithm can generate a shared secret key between two parties and is best suited when the two parties are on a public network where malicious individuals might be eavesdropping.

6.1.1 - Diffie-Hellman Algorithm

Elliptic Curve Cryptography (ECC) is an approach to public-key cryptography, based on the algebraic structure of elliptic curves over finite fields. ECC requires a smaller key as compared to non-ECC cryptography to provide equivalent security (a 256-bit ECC security has an equivalent security attained by 3072-bit RSA cryptography).
For a better understanding of Elliptic Curve Cryptography, it is very important to understand the basics of Elliptic Curve. An elliptic curve is a planar algebraic curve defined by an equation of the form:

$$y^2 = x^3 + ax + b$$

Where ‘a’ is the co-efficient of x and ‘b’ is the constant of the equation the curve is non-singular; meaning its graph has no cusps or self-intersections (when the characteristic of the Co-efficient field is equal to two or three). In general, an elliptic curve looks like as shown below. Elliptic curves could intersect almost three points when a straight line is drawn intersecting the curve. As one can see, the elliptic curve is symmetric about the x-axis, this property plays a key role in the algorithm.

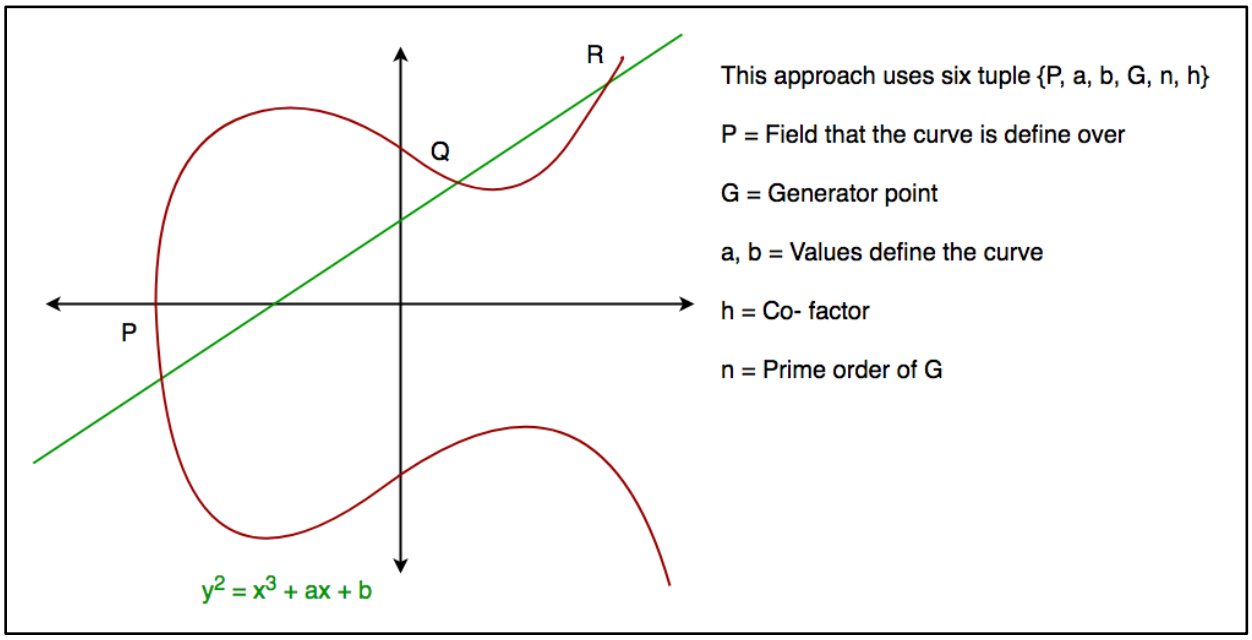

Figure 6.1 - Elliptic Curve

The Diffie-Hellman algorithm is being used to establish a shared secret that can be used for secret communications while exchanging data over a public network using the elliptic curve to generate points and get the secret key using the parameters.

- For the sake of simplicity and practical implementation of the algorithm, only four variables are considered, one prime P, G (a primitive root of P), two private values, and b.
- P and G are both publicly available numbers. Users (say, Alice and Bob) pick private values a and b and they generate a key and exchange it publicly, the opposite person received the key and from that generates a secret key after which they have the same secret key to encrypt.

Alice	Bob
Public Keys available = P, G	Public Keys available = P, G
Private Key Selected = a	Private Key Selected = b
Key generated = x = GamodP	Key generated = y = GbmodP
Exchange of generated keys takes place	
Key received = y	key received = x
Generated Secret Key = ka = yamodP	Generated Secret Key = kb = ybmodP
Algebraically it can be shown that ka = kb	
Users now have a symmetric secret key to encrypt	

Figure 6.2 – Establishing Shared Secret Key Between Two Parties

```
    def establishSecComm(self,target):
    #-------------------------------------------------------------------------------
--------
        '''
        Uses Diffie Helman key exchange to setup a shared secret key between the 2
parties
        '''
        P=getPrime(10)
        G=random.choice(self.primRoots(P))
        #print("P = {}, G = {}".format(P,G))
        #a for bob
        a= random.randint(2,1000) #private
        x=pow(G,a)%P #public
        #share/send P, G and x
        #then receive y
        tosend='-'.join([str(self.guid),str(P),str(G),str(x)])
        try:
            ip , port = self.peers[target]
            code, y=self.connectAndSend(ip,port,'rscm',tosend)
            #shared secret
            if code=='yscm':
                ka=pow(int(y),a)%P
                key=hashlib.sha256(ka.to_bytes(10,byteorder='big')).digest()
                self.keys[target]=key
                return True
            else:
                return False
        except:
            self.logging("* Could not connect to establish secure coms with {} ".fo
rmat(target))
            return False
```

```
    def rscm(self,peercon,data):
    #--------------------------------------------------------------------------
--------
        try:
            guid,P,G,x=data.split('-')
            guid=int(guid)
            P=int(P)
            G=int(G)
            x=int(x)
            b= random.randint(2,1000) #private
            y=pow(G,b)%P #public
            #send y
            peercon.sendData('yscm',str(y))
            kb=pow(x,b)%P
            key=hashlib.sha256(kb.to_bytes(10,byteorder='big')).digest()
            self.logging("Generated a key with peer : {}".format(guid))
            self.keys[guid]=key #store it with in table
        except:
            self.logging("Could not generated a key with peer : {}".format(guid))
```

6.1.2 - Encryption

Now that secret keys have been generated between the node and all the peers in its routing table, the key can be used to encrypt data before sending it on the network. To achieve this, it was decided to use the AES encryption with a 256-bit key. The National Institute of Standards and Technology selected three “flavors” of AES: 128-bit, 192-bit, and 256-bit. Each type uses 128-bit blocks. The difference lies in the length of the key. As the longest, the 256-bit key provides the strongest level of encryption. With a 256-bit key, a hacker would need to try 2256 different combinations to ensure the right one is included. This number is astronomically large, landing at 78 digits total. The three AES varieties are also distinguished by the number of rounds of encryption. AES 128 uses 10 rounds, AES 192 uses 12 rounds, and AES 256 uses 14 rounds. The more rounds, the more complex the encryption, making AES 256 the most secure AES implementation. It should be noted that with a longer key and more rounds comes higher performance requirements.

```
  def encrypt(self,key,to_encrypt):
    #---------------------------------------------------------------------------
--------
        cipher=AES.new(key,AES.MODE_EAX)
        nonce = cipher.nonce
        ciphertext, tag = cipher.encrypt_and_digest(to_encrypt)
        return (ciphertext,nonce,tag)

    #---------------------------------------------------------------------------
--------
    def decrypt(self,key ,nonce,to_decrypt):
    #---------------------------------------------------------------------------
--------
        cipher=AES.new(key,AES.MODE_EAX,nonce=nonce)
        plaintext=cipher.decrypt(to_decrypt)
        return plaintext
```

Most of the background calculation is done by the Crypto library Crypto.Cipher.AES.

6.2 - Blockchain

Blockchain technology refers to a decentralized database system containing cryptographically linked blocks of digital assets. Blockchain technology refers to a peer-to-peer network database ruled by a decentralized system. Blockchain defines three core strategies: Cryptography, Decentralization, and Consensus. All these measures make it difficult to tamper the blockchain technology with an individual record. It is because the hacker would be required to change the entire block containing the blockchain records. The network participants also have their security keys assigned to the ultimate transactional key that acts as a personalized digital signature. Under any circumstances, if the record is altered, the digital signature gets invalid, and any attack gets detected by the corresponding network right away.

Blockchain has several characteristics that make it secure:

Immutable, all the blockchain transactions in the Blockchain-based technology are immutable. Even the code encryption is done for the transactions, effectively covering the participants' date, time, and information.

Consensus-Based, the transactions in blockchain technology are executed only when the parties present on the network anonymously approve the same. One may even choose to alter the consensus-based regulations to match the circumstances.

Digital Signature, this Blockchain technology streamlines the exchange of translational values with unique digital signatures that depend on public keys. In Blockchain technology, private key codes are known only to the key owners to develop ownership proof. It is a critical feature that avoids any fraud in blockchain record management.

Persistent, Blockchain technology invalid transaction detected by the consensus-based system. It is not easy to roll back the transactions once they are integrated into the Blockchain ledger. Cryptographically, Blockchain blocks are created and sealed inside the chain, making it difficult for the hackers to edit or delete already developed blocks and put them on the network.

6.3 - Secure System Design

The system leverages Blockchain technology, Diffie-Hellman algorithm, AES encryption alongside sharding and replication to provide a high level of security. When a user chooses to upload a file, it is split into chunks, each chunk is encrypted, using the shared secret key (256-bit) and AES, it is then sent to a peer on the network. LinkToOGF is generated to act as reference to the original file, that way it is possible to know that the following chunk is part of a specific file. After receiving acknowledgment that the chunk has been received successfully, a transaction is made with the smart contract logging LinkToOGF, ChunkHash, ChunkOrder, senderGUID and receiverGUID. This means that every chunk's whereabout is logged to the Blockchain, this in turn applies all the technologies advantageous characteristics to the data that is being stored on it. Even in extreme scenarios where a malicious file was to be uploaded, it wouldn't even be considered as a threat because it is sharded into multiple small shards and distributed on the network according to a specified replication factor (One, Two or Three). Retrieving chunks is done securely now that ChunkHashes are stored and replicated on the private Blockchain, all a user's client needs to do is fetch the chunks logs from the chain and contact each receiver to ask him about a ChunkHash if available it is downloaded to the node's local storage to reconstruct the original file.

In the next chapter, the testing and validation phase of OmniCache is discussed, during which, several issues were faced, but have been successfully handled using the solutions stated in chapter seven.

CHAPTER 7

TESTING AND VALIDATION

Page

To properly test features that require network communication, a virtual private network (VPN) had to be setup and configured, so that communication between different nodes can happen as if they belonged to the same local area network.

Throughout the testing phase, a debugging version of the client had to be made, having GUI elements omitted and logging information every step of the way. To separate the two clients, two branches were created, a GUI branch and the main branch, whereas the name suggests GUI branch was used to version control the client with GUI elements and the main branch stored the core components of OmniCache pertaining to the P2P and Blockchain functionality. That way testing was easy, fast, and deprived of unnecessary exception that could hinder the debugging of network related features.

7.1 - Joining the Network as a Blockchain Node

As previously mentioned, one of the first steps is joining the network as a Blockchain node but while tackling this task a problem arose. Geth having an option to automatically discover nodes and add them to the network, produced unwanted behavior, as random nodes began joining the private network. That is why another method was utilized where, the system has to manually add specific nodes that want to join the private network. To achieve that, an additional protocol code had to be crafted to broadcast the joining node's Enode so that each existing node can add it manually as a peer.

7.2 - Offline Nodes

A core part of OmniCache is that it is fault-tolerant, in other words, nodes need to be able to adapt to random peers disconnecting and reconnecting to the network. While testing, abnormal behavior started appearing because nodes that were previously online have become offline and contacting them would surely raise exception in the code. To mitigate such behavior, periodic heartbeat messages had to be implemented and added to the protocol, simply checking the online status of the peer before initiating communication is enough to avoid unwanted exceptions.

7.3 - Encapsulation and Decapsulation of Encrypted Messages

Encapsulating an encrypted chunk of size 3704 bytes inside a payload of 4096 bytes had to be decapsulated then decrypted on the other end of the communication line. Having the encrypted

bytes concatenated to the nonce while being separated by a dash created complication downline and was very hard to figure out what was the cause of the bug. It turns out that the AES encryption replaces some characters by a dash and that was troublesome, because when splitting the bytes on the dash delimiter, the resulting list had more than three elements which was not supposed to happen. That is why replacing the dash delimiter by a sequence of characters that won't appear after encrypting was the solution to the problem, example: ("[]").

7.4 - Running the Node

During initial testing phases, several issues would pop up regarding synchronization with the Blockchain.

When mining around 100 plus blocks, any new nodes attempting to join the network would run into synchronization issues and fail in completing the process of adding peers and transacting with the rest of the nodes. It was quickly discovered that the problems were stemming from the synchronization mode of the Geth client. By default, Geth runs nodes with a synchronization method known as "Fast", where it downloads the state rather than the entire block data. An attempt to mitigate the synchronization difficulties came in the form of asking Geth to use a larger internal cache than the default 1024 MB.
Even with this, the problems persisted, and the only way to solve it was by making sure the synchronization mode was in "Full" instead. It is suspected that the issues may have been due to clashes with the PoA consensus algorithm utilized but these suspicions need to be investigated further. With the new flags set, Geth runs the node correctly every time and the problems were successfully eliminated.

7.5 - Transacting with the Genesis

A core part of joining the Blockchain network is successfully receiving ETH for transaction gas from the genesis. If two nodes were attempting to join simultaneously, one of the transactions would fail and a node would have to restart the process of joining the Blockchain network. A race condition existed due to the way the transaction is sent. The transaction requires a nonce, to distinguish it from others, this nonce is used by retrieving the transaction count in the current Blockchain, so when two nodes were requesting ETH at the same time, the nonce would be the

same for both transactions and the latter one would fail. To counter this, a basic form of semaphore was used.

7.6 - Minimizing Interaction with the Geth Client

Geth features a command line program that is used to run and manage different aspects of the Blockchain nodes. During early prototypes, a major portion of OmniCache required manual usage and entering of commands to successfully create and run nodes, and a command prompt with Geth output was always visible. For the sake of simplicity and intuitive usage, a huge effort was undertaken to hide the command prompts and automate or mask most of the input. At the same time, proper logs were required to ensure that stability and support was not compromised. Several issues came to be during this phase, as threads would refuse to terminate, and log files would grow to immense sizes. Using properly placed daemon threads, rotating file handlers, and creation of a class known as LogPipe to redirect running node output and error, correct logging and automation were achieved.

7.7 - Ensuring Proper Synchronization

It would occur sometimes that whenever a node would attempt to enroll by transacting with the smart contract, that the transaction would fail, and a restart would be required. It was discovered that this was because either the Geth client would have taken too long to run the node or synchronization would not have ended. Many steps were taken to try and resolve this issue, but most have failed. It was not until a proper function to check synchronization status was created, this function consisted of checking the node for correct addition of peers, along with proper block number checks and whether the node was still in sync. By utilizing this, OmniCache would wait for proper synchronization status before attempting to transact with the smart contract for enrollment purposes.

7.8 - System Tray

OmniCache did not have any way to let the users keep the application running in the background and keep on hosting the files of other peers. For the sake of redundancy and reliability, by creating a system tray class that lets the application continue to function while it is minimized in the operating system’s tray, the issue was solved and OmniCache users can host files properly while using their devices for other tasks.

7.9 - Node Referencing

To be able to use the functionalities of the P2P node, OmniCache requires to have a reference for the node created upon joining the network in most of the GUI classes.

During the development process, a persistent problem was that the node did not have any reference in the GUI classes, and the only way to do it was with a click of a button called “Join Network”. By creating a function named initNode() in the main driver class, this function handled the creation and the connection of the node to the P2P network once the user clicks on the “Join Network” button and sends the node object reference to the main homepage and other classes.

7.10 - Multithreading

The first time testing the upload functionality, the main GUI thread froze while uploading until it was finished. After research, it turns out that functionalities like uploading, downloading, and deleting files should be running in a different thread. By using QThread, which is the central class of the Qt threading system, P2P or Blockchain functions called from the GUI classes are handled in separate threads to avoid any performance or freezing issues in the software.

7.11 - Node Preparation

After implementing the upload and download files features and testing them, they were not working immediately upon launching the software since the node was not ready to handle the user’s requests. A thread was created named “NodeReady” that begins running at the loading screen page, this thread handles the preparation of the node and returns a confirmation that the software is ready to be used.

7.12 - Dummy File Item

It is important to inform the user that the file has started uploading. To accomplish this, a file item is added to the list of files on the homepage, but the item is unclickable because the file is not uploaded yet and not ready for use. After facing some issues, steps were taken to counteract them including, creating an unclickable dummy widget class of the original file item widget while the file is uploading and replacing it with the original widget class after the uploading process is finished.

7.13 - IP Input Regular Expression

Having the user's input as an invalid IP address caused a major issue for the creation of the node. To force the user to input a text as an IP address form, a regular expression is created and used in validating the user's input to ensure a proper IP address.

7.14 - User Passphrase Validation

Passphrase validation is handled while creating the Blockchain node which is handled in a separate thread. The passphrase input string is stored in a variable from QInputDialog which a Qt widget popup that asks the user for an input. The issue was that Qt widgets cannot be moved to another thread; they must stay in the main GUI thread. By relocating the input dialog popup in the main GUI thread and dividing the Blockchain creation thread into two different threads, one before the user's input and one afterward, the issue was solved.

In the next chapter, a basic feasibility study is performed to show the hours put in for each step of the project, also an estimated cost was calculated based on the working hours of the team.

CHAPTER 8

FEASIBILITY STUDY

Page

Thorough study of the feasibility regarding any system is of immense importance, and it should be carefully considered before proceeding with the system's development and deployment. Many attributes contribute to the cost versus benefit analysis such as licensing fees, hardware requirements, the number of hours spent on planning, research, analysis, design, implementation, and testing.

8.1 - Software Requirements

- Python as a programming language with free libraries available to download and install:
- Web3 to interface with the Blockchain
- Crypto to encrypt and decrypt using AES
- PyQt5 for the graphical user interface
- Geth which is the go implementation of the Ethereum™ Blockchain, it can be downloaded for free.

8.2 - Total Number of Working Hours

The total number of working hours consists of the summation of the hours spent on corresponding phases: planning, research, analysis, design, implementation, and testing. The number of hours spent on the planning phase was eight hours per week by three people for two weeks, making the total time spent on planning 48 hours. The number of hours spent on the research phase was approximately 10 hours per week by three people for two weeks making a total of 60 hours. The number of hours spent on the analysis phase was around seven hours per week by three people for two weeks for a sum of 42 hours. The Design phase took 10 hours per week for two weeks, adding up to 20 hours. The number of hours spent on the Implementation phase was almost 20 hours per week by three people over the course of eight weeks, adding up to 480 hours. As for the testing phase, the number of hours spent was 10 hours per week by three people for six weeks, adding up to 180 hours. Thus, the total number of hours spent is 830 hours. The cost per hour of the development efforts presented by every individual can be estimated to be around the average hourly earnings of a software developer, estimated to be $14.5 per hour.

8.3 - Overall Cost of the Project

The overall cost of the project is the total price of all the resources spent to achieve the final product.

Software requirements are free of charge since all the frameworks used were open source or free, and the total number of working hours that the development team have contributed to develop this system.

The following table shows all the costs related to this project.

Table 1 - Costs Table

Expense Type	Time Spent (Hour)	Cost (Time Spent x avg Salary in $)
Planning	48	696
Research	60	870
Analysis	42	609
Design	20	290
Implementation	480	6960
Testing	180	2610
Software	0	0
Hardware	0	0
Total	830	12035

The cost of each expense type was calculated by multiplying the number of hours spent on the corresponding type with the cost per hour.

8.4 - Making Profit from OmniCache System

OmniCache provides the user with a huge remote storage space in exchange its cryptocurrency. By reserving a storage space on the user's local machine for file hosting, users can generate Omnies to use later as payment when storing files. On the other hand, users will have the option to buy directly Omnies by converting normal currency through a website, which will be implemented in the future.

Lastly in chapter nine, the conclusions and future work are presented with key factors for the success of OmniCache.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

This final chapter summarizes the achieved results from this project. It also presents paragraph the learning outcome of the team, the relevance of the work and discusses where the project will be heading towards in the future.

9.1 - Achieved Results

Using P2P communication and Blockchain technology, OmniCache can achieve similar results to big companies implementing decentralized file storage. But unlike other solutions, OmniCache tends to the everyday user who has unused storage space. The following features were implemented successfully:

An intuitive and user-friendly interface.

The deployment of a private Ethereum network alongside a P2P network.

Encrypted end-to-end communication.

Upload, Download, and deletion of sharded files.

Auditing and recording transactions using a Blockchain.

9.2 - Acquired Knowledge

During the seven months of work, a great deal of knowledge was acquired about new technologies. Blockchain technology, smart contract scripting, P2P networks and communication, end-to-end encryption, and the QT library were all new addition to the team's skillset. Additionally, remote collaboration using GitHub and version control solutions proved an invaluable skill required to be learned in order to be able to work properly during the outbreak of the COVID-19 pandemic.

9.3 - Relevance of the Work

Infinitely scalable, OmniCache offers users a secure and private alternative to traditional remote file storage while allowing users with unused storage space to employ it and generate Omnies which can then be spent on additional storage, thus creating a self-sustaining ecosystem.

9.4 - Future Work

The current version of OmniCache implements only the core features that enable decentralized file storage. As future work, the implementation of the following should be considered: Achievements and unlockable titles, routing table management, a website through which validator data can be

submitted for processing, the ability to directly purchase Omnies and finally, gamifying the user experience while incentivizing them to stay online and contribute to the network plays a key factor for the self-sustainability of OmniCache.

References

[1] "RAID 0, 1, 5, 6 & 10" Accessed: Oct. 23, 2020. [Online]. Available: https://www. prepressure. com/library/technology/raid

[2] Michael Banks, "Cloud computing? Been there. Done that. ", November 24, 2008. [Online]. Available: https://www. techrepublic. com/blog/classics-rock/cloud-computing-been-there-done-that/

[3] David Vellante, "Cloud Computing 2013: The Amazon Gorilla Invades the Enterprise", March 28, 2014.[Online] Available: http://wikibon.org/wiki/v/Cloud_Computing_2013%3A_The_Amazon_Gorilla_Invades_the_Enterprise

[4] Guy Douglas BA (Hons), LLB (Hons), PhD, "Copyright and Peer-To-Peer Music File Sharing: The Napster Case and the Argument Against Legislative Reform", March 2004. [Online]. Available: http://www. murdoch. edu. au/elaw/issues/v11n1/douglas111. html

[5] Lucas Mearian, "Start-up unveils cloud storage co-op" September 2009. [Online]. Available: https://www. computerworld. com/article/2527804/start-up-unveils-cloud-storage-co-op. html

[6] Al Shehhi, A. , Oudah, M. , & Aung, Z. "Investigating factors behind choosing a cryptocurrency. " December 2014. [Online]. Available: https://www. researchgate. net/publication/290814876_Investigating_factors_behind_choosing_a_cryptocurrency

[7] SolarWinds MSP, "Centralized Networks vs Decentralized Networks", November 30, 2018. [Online]. Available: https://www. solarwindsmsp. com/blog/centralized-vs-decentralized-network

[8] Loki Admin, "Centralized vs Decentralized Networks", Accessed: Oct. 24, 2020. [Online]. Available: https://loki. network/2019/12/05/centralized-vs-decentralized-networks/

[9] Van Steen, M. , Tanenbaum, A. S. A brief introduction to distributed systems. *Computing* 98**,** 967–1009 (2016). [Online] Available: https://doi. org/10. 1007/s00607-016-0508-7

[10] Saifulazmi Tayib,Client-Server Model, Accessed: Oct. 23, 2020. [Online]. Available: http://norlizakatuk. weebly. com/uploads/2/6/6/0/26606863/saiful_azmi. pdf

[11] Galuba W. , Girdzijauskas S. (2009) Overlay Network. In: LIU L. , ÖZSU M. T. (eds) Encyclopedia of Database Systems. Springer, Boston, MA. Accessed: Oct. 23, 2020. [Online]. Available: https://doi. org/10. 1007/978-0-387-39940-9_1231

[12] Michael Dufel. Distributed Hash Tables And Why They Are Better Than Blockchain For Exchanging Health Records. Dec 26, 2017. [Online]. Available: https://medium. com/@michael.dufel_10220/distributed-hash-tables-and-why-they-are-better-than-Blockchain-for-exchanging-health-records-d469534cc2a5

[13] S. Nakamoto. (2008). BitCoin: A Peer-to-Peer Electronic Cash System. Accessed: Oct. 23, 2020. [Online]. Available: https://bitcoin. org/bitcoin. pdf

[14] D. K. Tosh, S. Shetty, X. Liang, C. Kamhoua, and L. Njilla, ‘‘Consensus protocols for Blockchain-based data provenance: Challenges and opportunities,’’ in Proc. IEEE 8th Annu. Ubiquitous Comput. , Electron. Mobile Commun. Conf. (UEMCON), Oct. 2017, pp. 469–474

[15] G. Wood, ‘‘Ethereum™: A secure decentralized generalized transaction ledger,’’ Yellow Paper, 2014. Accessed: Oct. 23, 2020 . [Online]. Available: https://Ethereum. github. io/yellowpaper/paper. pdf

[16] D. Idilio, B. Enrico, M. Marco, S. Herman, P. Aikos ,Benchmarking Personal Cloud Storage, Accessed : November. 11,2020 [Online] . Available : https://research. utwente. nl/files/5511217/cloud_storage. pdf

[17] Seth Noble ,6 hidden bottlenecks in cloud data migration , Accessed: November. 11,2020 [Online] . Available : https://www. infoworld. com/article/3268954/6-hidden-bottlenecks-in-cloud-data-migration. html

[18] Jon Brodkin, Accessed: November. 11,2020 [Online] . Available : https://arstechnica.com/information-technology/2014/01/dropbox-messed-up-os-upgrade-caused-two-days-of-downtime

[19]: Rosa Golijan, Accessed: March. 15, 2013 [Online] . Available : https://www. nbcnews.com/technolog/your-cloud-drive-really-private-not-according-fine-print-1C8881731

[20] Mariia Rousey·September 1, 2019: [Online] . Available : https://changelly.com/blog/what-is-proof-of-authority-poa/#Pros-and-cons

[21] Eye Health, Eye Safety, “Is dark mode better for your eyes?”, January 30, 2020. [Online]. Available: https://rxoptical. com/eye-health/is-dark-mode-better-for-your-eyes/#:~:text=Dark%20mode%20can%20reduce%20eye,light%2Don%2Ddark%20theme