UNIVERSITY OF PIRAEUS - DEPARTMENT OF INFORMATICS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

MSc «Advanced Informatics and Computing Systems - Software Development and Artificial Intelligence»

ΠΜΣ «Προηγμένα Συστήματα Πληροφορικής – Ανάπτυξη Λογισμικού και Τεχνητής Νοημοσύνης»

MSc Thesis

Μεταπτυχιακή Διατριβή

Thesis Title: Τίτλος Διατριβής:	**Distributed File Storage with User-Hosted Nodes** Κατανεμημένη αποθήκευση αρχείων με κόμβους που φιλοξενούνται από χρήστες
Student's name-surname: Ονοματεπώνυμο φοιτητή:	**Vasileios Argyropoulos** Βασίλειος Αργυρόπουλος
Father's name: Πατρώνυμο:	**Ioannis** Ιωάννης
Student's ID No: Αριθμός Μητρώου:	ΜΠΣΠ2204
Supervisor: Επιβλέπων:	**Efthimios Alepis, Professor** Ευθύμιος Αλέπης, Καθηγητής

July 2024/ Ιούλιος 2024

3-Member Examination Committee

Τριμελής Εξεταστική Επιτροπή

Efthimios Alepis	**Costantinos Patsakis**	**Maria Virvou**
Professor	**Associate Professor**	**Professor**
Ευθύμιος Αλέπης	Κωνσταντίνος Πατσάκης	Μαρία Βίρβου
Καθηγητής	Αναπληρωτής Καθηγητής	Καθηγήτρια

Table of Contents

Table of Figures

Acknowledgments

I would like to express my gratitude to the people who offered me valuable support for the completion of this thesis, which was prepared for the postgraduate program "Advanced Informatics and Computing Systems - Software Development and Artificial Intelligence" of the Department of Informatics of the University of Piraeus

First, I would like to thank my three-member examination committee for their excellent guidance, patience, and support during my research and academic career.

I would also like to express my sincere thanks to my Mother, who has been a source of inspiration, guidance and support to me. Her love, patience, and dedication have given me the strength to continue even in the most difficult times.

Finally, I would like to thank my family and friends who have supported me throughout my academic career.

Ευχαριστίες

Θα ήθελα να εκφράσω της ευγνωμοσύνη μου προς τα άτομα που μου προσέφεραν πολύτιμη υποστήριξη για την ολοκλήρωση αυτής της διατριβής, η οποία εκπονήθηκε για το πρόγραμμα μεταπτυχιακών σπουδών «Προηγμένα Συστήματα Πληροφορικής – Ανάπτυξη Λογισμικού και Τεχνητής Νοημοσύνης» του τμήματος Πληροφορικής του Πανεπιστημίου Πειραιώς

Αρχικά, θα ήθελα να εκφράσω τις ευχαριστίες μου στην τριμελή εξεταστική επιτροπή μου για την εξαιρετική καθοδήγηση, υπομονή και στήριξη κατά την διάρκεια της έρευνάς μου και της ακαδημαϊκής μου πορείας.

Επίσης, θα ήθελα να εκφράσω τις ειλικρινείς μου ευχαριστίες στην Μητέρα μου, η οποία αποτελεί πηγή έμπνευσης, καθοδήγησης και υποστήριξης μου. Η αγάπη, η υπομονή και η αφοσίωσή της μου έδωσαν τη δύναμη να συνεχίσω ακόμη και στις πιο δύσκολες στιγμές.

Τέλος, θα ήθελα να εκφράσω τις ευχαριστίες μου στους συγγενείς και φίλους μου που με στήριξαν σε όλη την πορεία της εκπόνησης της ακαδημαϊκής μου πορείας.

Abstract

This thesis delves into a novel approach to distributed file storage, a system that consolidates storage contributions using self-hosted workers. This innovative system integrates various storage types, including local, cloud, and network-attached storage, into a storage pool. It employs user-hosted nodes managed by a centralized control infrastructure to ensure data redundancy, high availability, and security. By utilizing data encryption and distribution methodologies, the system effectively safeguards data privacy and enhances resilience against potential failures and security breaches.

The architecture of this system strikes a balance between the advantages of centralized and distributed storage models, offering a flexible, scalable, and secure solution for contemporary storage requirements. It triumphs over the limitations of existing storage solutions by enabling the seamless integration of diverse storage media while maintaining optimal performance and efficient resource utilization. Comprehensive performance evaluations demonstrate the system's effectiveness, underscoring its potential as a solution for organizations seeking to consolidate their storage infrastructure through a distributed approach.

Περίληψη

Η παρούσα διατριβή εξετάζει μια νέα προσέγγιση για την κατανεμημένη αποθήκευση αρχείων, ένα σύστημα που ενοποιεί τις συνεισφορές αποθήκευσης χρησιμοποιώντας self-hosted workers. Αυτό το καινοτόμο σύστημα ενσωματώνει διάφορους τύπους αποθήκευσης, συμπεριλαμβανομένων των τοπικών, του cloud και της δικτυακής αποθήκευσης, σε ένα storage pool. Χρησιμοποιεί κόμβους που φιλοξενούνται από χρήστες και διαχειρίζονται από μια κεντρική υποδομή ελέγχου για να διασφαλίσει τον πλεονασμό των δεδομένων, την υψηλή διαθεσιμότητα και την ασφάλεια. Χρησιμοποιώντας μεθοδολογίες κρυπτογράφησης και διανομής δεδομένων, το σύστημα διασφαλίζει αποτελεσματικά το απόρρητο των δεδομένων και ενισχύει την ανθεκτικότητα έναντι πιθανών αστοχιών και παραβιάσεων ασφαλείας.

Η αρχιτεκτονική αυτού του συστήματος επιτυγχάνει ισορροπία μεταξύ των πλεονεκτημάτων των συγκεντρωτικών και των κατανεμημένων μοντέλων αποθήκευσης, προσφέροντας μια ευέλικτη, κλιμακούμενη και ασφαλή λύση για τις σύγχρονες απαιτήσεις αποθήκευσης. Θριαμβεύει έναντι των περιορισμών των υφιστάμενων λύσεων αποθήκευσης, επιτρέποντας την απρόσκοπτη ενσωμάτωση διαφορετικών μέσων αποθήκευσης, διατηρώντας παράλληλα τη βέλτιστη απόδοση και την αποδοτική χρήση των πόρων. Αναλυτικές αξιολογήσεις επιδόσεων αναδεικνύουν την αποτελεσματικότητα του συστήματος, υπογραμμίζοντας τις δυνατότητές του ως λύση για οργανισμούς που επιδιώκουν να ενοποιήσουν την υποδομή αποθήκευσης μέσω μιας κατανεμημένης προσέγγισης.

Introduction

Short description of the Problem

Existing distributed file storage solutions are either fully decentralized, which can pose challenges in terms of control and management, or they do not allow the seamless incorporation of third-party storage providers in a way that is agnostic to the type of storage being used. This presents a significant challenge for organizations that wish to leverage a diverse range of storage options without being constrained by the architecture of their storage solution. The inability to integrate various storage types into a cohesive system means that many organizations cannot optimize their storage infrastructure to meet their specific needs, leading to inefficiencies and potential vulnerabilities.

Consider, for instance, a scenario where a consortium of organizations explores implementing redundant file storage by pooling their available resources. Each organization in the consortium may have different types of storage media at its disposal, such as network-attached storage, local storage, or cloud storage resources. The idea is to contribute these varied resources to a unified storage pool that ensures redundancy and resilience. While the concept is appealing, current distributed storage solutions do not support such diverse integration in a straightforward and efficient manner. As a result, these organizations might face potential barriers to achieving a unified storage solution that can effectively leverage their collective storage capabilities if they adopt this approach.

Approach

The proposed distributed storage architecture offers a range of potential benefits. It can integrate various storage types, presenting them as a unified pool that ensures both redundancy and high availability. This involves coordinating storage contributions from multiple providers while maintaining data consistency and integrity across the distributed network. Such an approach requires advanced mechanisms to manage the heterogeneity of storage types efficiently and securely, ultimately providing a robust and flexible storage infrastructure. Additionally, due to the nature of the data stored, security and privacy are highly important, and strict mechanisms must be implemented to ensure that the data is securely stored and only accessed by authorized parties. The system's ability to seamlessly incorporate different storage media and its focus on data security and privacy make it a promising solution for organizations seeking to optimize their storage infrastructure.

To achieve these objectives, a centralized control infrastructure is designed and implemented. This infrastructure orchestrates the workers deployed by the storage providers and issues commands aimed at efficiently and securely managing the available storage resources. The system ensures optimal performance, reliability, and security in managing diverse storage resources by centralizing control while leveraging storage contributions in many locations.

Goals

The main objective of this thesis is to establish a distributed storage architecture that integrates various storage media into a unified and resilient storage pool. The goals include:

- Orchestrating contributions from different providers while maintaining data consistency and integrity across the distributed workers.
- Securely and efficiently managing heterogeneous storage types across the distributed storage infrastructure.

- Having the ability to horizontally and vertically scale, along with the ability to orchestrate a large number and size of available storage infrastructure.

The solution is realized in a novel software system that meets these requirements. This system employs techniques to manage many storage types, ensuring that the distributed worker-based storage infrastructure operates efficiently and securely. The design process and implementation of this system are detailed in the following analysis, which explores the architecture and functionality in depth. All accompanying diagrams are based on the C4 model for visualizing software architecture, providing a clear and structured representation of the system components and their interactions.

Design Assumptions

The proposed solution makes the following design assumptions:

- The system should be able to integrate with multiple storage types and make them a storage pool.
- Any file must be always available for retrieval and stored redundantly.
- The storage contributions by the storage providers infrastructure must not be fully trusted with the data that they hold.
- The system must be able to efficiently recover from worker failures and disconnects.
- The files will be added to the system encrypted, and they will be decrypted only on the client side by the authorized and authenticated entities.

Threat assumptions

The proposed solution makes the following threat assumptions:

- The network infrastructure is susceptible to attack vectors such as and denial-of-service attacks:
  - **Eavesdropping:** This involves unauthorized interception of data traveling across the network. Attackers can passively listen to the communication, capturing sensitive information such as credentials or private messages, leading to privacy breaches.
  - **Denial-of-service (DoS) attacks:** These attacks aim to disrupt the normal functioning of a network or service by overwhelming it with a flood of illegitimate requests. This can render the service unavailable to legitimate users, causing significant downtime and potentially leading to data loss or delayed processing.
- Individual storage nodes or workers cannot be fully trusted and may be compromised or abused:
  - **Compromise of storage nodes:** Malicious actors may gain unauthorized access to storage nodes, either through exploiting vulnerabilities or insider threats. Once compromised, these nodes can be used to access, alter, or delete stored data, jeopardizing data integrity and confidentiality.
  - **Abuse of storage nodes:** Even legitimate storage nodes can be misused by trusted parties or turn malicious over time. This abuse can involve unauthorized access to data, tampering with stored information, or denial of service by selectively withholding data or resources.
- Authorized entities may attempt unauthorized access to other files:
  - **Unauthorized access attempts by authorized users:** Users with legitimate access to the system might try to access files beyond their permission level. Such insider threats can lead to data breaches, where sensitive information is accessed,

copied, or distributed without proper authorization, compromising data privacy and security.

- The system will encounter periodic worker failures and disconnections:
    - **Worker failures:** These can occur due to hardware malfunctions, software bugs, or operational errors, leading to loss of connectivity or functionality. Such failures can disrupt the availability and reliability of the system, potentially resulting in data loss or delayed access to stored information.
    - **Disconnections:** Network issues or maintenance activities can cause workers to disconnect from the system. These disconnections can temporarily lose data accessibility, affecting the system's performance and user experience. Repeated or prolonged disconnections can also strain the system's redundancy and failover mechanisms.

Design

This section outlines the design considerations and decisions. First, a high-level overview of the system is explained, and then each component and its design considerations are analyzed.

Bird’s eye view of the system

A distributed architecture with a central control point is proposed to fulfill the functional requirements that were previously outlined. This topology comprises distributed nodes - called workers - deployed and hosted by the storage provider, connecting to the provided storage resources. These workers receive control instructions from a centralized infrastructure responsible for coordinating file distribution across the network. Additionally, this centralized infrastructure manages authentication, authorization, and cryptographic keys generated and utilized by the client application with which the user interacts. The frontend application served to the end user also provides a user-friendly interface for system use and content management.

Because this design relies on the workers interacting with the storage provider’s storage infrastructure, the computational load of the operations is distributed along the centralized control infrastructure and the workers, thus enabling more efficient scalability as the computational and storage resources become available. This distribution of tasks ensures that as more users and storage providers join the network, the system can handle increased demand without compromising performance or reliability. Furthermore, the architecture's flexibility allows it to adapt to varying storage and computational power levels, ensuring optimal performance across different scenarios.

The design of the worker and service architecture was driven by the need to create a distributed system for file storage homologation. The approach ensures that the worker nodes, which handle storage tasks, can operate efficiently and independently across various environments. This independence allows them to scale with the addition of new user-hosted resources. Meanwhile, the central service is designed to manage and coordinate these distributed workers. The central service also integrates authentication, authorization, and key management, ensuring that security and user access control are maintained centrally without needing the worker nodes. This design ensures that the system can scale while maintaining performance and security.

User-Hosted Worker

The design process of the Workers focused on a balance between autonomy, scalability, and semi-trust. The main goal was to create a component that could live as part of a distributed architecture but handle its storage tasks independently. This autonomy would allow each worker to work independently of other workers.

Scalability was a key consideration, and since workers are hosted on the client's infrastructure, the goal was to make it easier to scale the system by adding more workers without affecting overall performance. Security was a primary concern, making it essential to ensure that worker-managed data would remain unusable in the event of a data breach of the worker. Finally, workers are not fully trusted by the network, so they should not have unrestricted access to information on the network.

Several key techniques were used to implement scalability: The design uses a distributed approach that allows storage providers to add self-hosted workers. This model ensures the workload is distributed across multiple nodes in the end-user's infrastructure, eliminating individual bottlenecks. Worker autonomy means that each worker can operate independently, reducing dependencies and bottlenecks. The centralized system also used dynamic resource allocation, adjusting the use of computing and storage resources based on current needs to maintain efficiency. In addition, modular design principles were applied to facilitate component integration and expansion without affecting existing functionality.

To implement security and semi-trust, several key techniques have been used to ensure privacy and integrity in a semi-trusted network. The main method was to split the encrypted file into pieces of random size. The random size of the chunks adds an additional layer of security by making it more difficult for an attacker to reconstruct the original file. Even if an attacker gains access to some chunks, they would not have a predictable pattern to follow, complicating the task of reassembling the data. This randomness, combined with the encryption and distribution of chunks across multiple workers, enhances the overall security and privacy of the system, ensuring that compromised data remains unusable without access to all parts and their correct sequence.

Strict authorization and access rights are implemented to ensure that workers can only perform predefined tasks like file chunk addition, retrieval, and deletion. In addition, redundant file block storage for multiple workers added a layer of failover that ensures data is available even if some workers fail. Together, these methodologies ensured system security and semi-trust, ensuring no worker could compromise data integrity or confidentiality.

Finally, to increase worker autonomy, it was designed so that each worker could work independently without relying on other workers. This was made possible by local task management, where each worker handled their own storage, file retrieval, and maintenance tasks. Centralized decision-making allowed workers to act without coordination with other nodes and complete tasks even when other network parts were unavailable. That independence allowed workers to deal with errors and maintain continuous operation.

Centralized Service

The design goals of the centralized system focused on ensuring coordination, security, and user accessibility across the network. Its goal was to create a robust framework for managing authentication, authorization, and encryption key distribution, ensuring that all network operations are secure and well-regulated. The centralized system was also designed to allow efficient communication and management of distributed workers, allowing them to work independently while maintaining the network's overall integrity. A major responsibility of the Centralized service is to act as the semi-trusted cryptographic key management server, handling the user’s encrypted cryptographic keys and applying authentication and authorization to them using the Identity Server. In addition, it aims to provide a user-friendly interface to manage storage and access rights.

WebSockets were the transport protocol selected with the goal of allowing real-time communication between workers and the centralized service. They provide a continuous, low-latency connection that enables data transmission and control instructions instantly. This decision was driven by the need for bidirectional communication, which is critical for coordinating distributed storage operations and maintaining overall system performance. In addition, WebSockets allow workers to connect without exposing a port, improving security by reducing the potential attack surface.

To ensure efficient management of encryption keys, the design used techniques such as client-side key generation and encryption of the private key in the client using a password to ensure that private keys never end up in the central system, thus improving security. Encrypted storage of keys in the central system protects them from unauthorized use. Strict access control ensured that only authorized users could obtain and use their keys. The system also used two separate passwords: one for authentication and one for key decryption, adding further security by separating the verification of the user's identity from using the data access keys. The centralized service played a key role in key management, acting as the Key Management Server and ensuring the keys were accessible only to authorized users. In addition, secure key distribution methods have been implemented to maintain the integrity and confidentiality of the keys throughout their lifetime, like having them encrypted a second time at rest in the database. Keys must also be rolled frequently, where new user keys are generated on the client side, and file keys are re-encrypted on the client side. Finally, the Centralized service requires authentication of any user with the Identity Server to provide access to his encrypted cryptographic keys, thus acting as the Semi-Trusted Key Management Server.

Files are encrypted using a process designed to ensure data security and confidentiality. Initially, the user's device generates a unique encryption key for each file in the client running in the browser, which is used to encrypt the file's contents locally. This encryption key is then encrypted with the user's private key. The system uses a file key sharing mechanism to share files, where the file's encryption key is encrypted with the recipient's public key, ensuring that only the intended recipient can decrypt and access the key and file. The encrypted file and encrypted file key are then stored securely. This method ensures that even if the storage is compromised, the data remains unreadable and secure, as a decryption key and appropriate permissions are required to access the original file. By managing the encryption process locally on the user's device and using encrypted file keys for sharing, the system maintains high data security during the storage and retrieval processes.

To increase reliability, the system used several key methods to ensure data integrity and availability even when workers are terminated or removed. Continuous monitoring of the status of each worker using the WebSockets API helped detect problems in real time, allowing immediate intervention in the event of disconnection. When new workers were connected, they were provided with the least represented data chunks in the network, optimizing chunk distribution and reducing the risk of a single worker becoming a performance bottleneck or point of failure. On the other hand, when workers disconnected, their information was immediately retrieved by the remaining workers and redistributed to maintain redundancy and prevent data loss. The central service played a crucial role, as it kept a detailed log of the location of file chunks between workers. By documenting which file pieces were saved to which worker, the system could quickly reorganize and maintain data availability.

Cryptography

The cryptographic processes designed in this system ensure security and data integrity throughout the user's interaction with the application. Upon registration, the user's browser generates asymmetric RSA-OAEP encryption keys and RSA-PSS digital signature keys. Subsequently, the user is prompted to create a second password distinct from the account password. This second password is hashed using SHA-512, and the hash is used to produce a symmetric AES-GCM key, utilized solely on the client side for encrypting and later decrypting the user's private encryption and digital signature keys. The public and encrypted private keys are then stored securely in the Key Management Server. During each login, the user must provide the decryption password, recreating the symmetric key to decrypt the private keys, thereby enabling access to the user's content.

A random symmetric key is generated to encrypt the file contents when uploading a file. The File Service then processes the encrypted file for chunking and distribution. After the file is chunked, for an added layer of security, the chunk is encrypted by a server-side AES-CBC 128bit encryption key, and the SHA-256 hash before and after the encryption is saved for validation upon retrieval. Simultaneously, the file's symmetric key is encrypted with the user's public key, and a digital signature of the file's original contents is created using the user's private key. The encrypted file keys and metadata are stored in the Key Management Service, ensuring that the unencrypted file remains only in the user's browser for enhanced security. For file sharing, the file's owner decrypts the file key in their browser and re-encrypts it with the recipient's public key. This re-encrypted key and corresponding ownership metadata are stored securely. Revoking access involves the file owner requesting the deletion of this key copy from the Key Management Service, effectively removing the recipient's authorization. When a user fetches a file, the necessary keys are retrieved and decrypted using the user's private key. The decrypted file keys are then used to decrypt the file, and its contents are validated using the owner's digital signature, ensuring the file's integrity and authenticity. When a user is de-authorized to access a file, his keys are deleted from the database, and even if he requests the file from the file service, the reassembly algorithm will not be run due to the lack of authorization.

Limitations

The above design has the following limitations:

- Initial Workers: For the system to initialize and be made available to the end-users, there need to be at least 3 available workers. Nonetheless, the system cannot accept files to be stored.
- Replication Factor: To balance storage efficiency and resiliency to worker failure, a replication factor of three for all files in the system has been selected.
- Worker failure: Due to the replication factor of three, the system can only recover from two concurrent worker failures.
- Storage availability: The usable storage is one-third of the available storage because every file is saved in three copies due to the replication factor.
- Data loss Point: Due to worker disconnections, the system will lose data when the available storage is less than the used storage.
- Single Point of Failure: Despite being distributed, the reliance on a central control point introduces a single point of failure.
- Client-Side Hosting: Hosting workers on client infrastructure introduces variability in performance, depending on the client's resources.

Related Work

Survey of Prior Publication

In this section of the thesis, a comparison will be made with the previous publication, "Semi-Decentralized File Sharing as a Service" (Argyropoulos, Patsakis and Alepis 2022), which aims to address the limitations of existing file-sharing platforms by combining the benefits of centralized and decentralized approaches. It proposes a peer-to-peer, semi-decentralized file-sharing system that leverages the InterPlanetary File System (IPFS) (Protocol Labs 2014) for decentralization and a microservices centralized architecture for control. This application's primary goal is to provide a user-friendly yet secure platform for sharing files, using IPFS as the primary backbone.

On the other hand, this thesis aims to integrate various storage types, including local, cloud, IPFS, and network-attached storage, into a unified storage pool using a self-hosted worker-based distributed infrastructure. One major difference is that while the publication acts as a centralized wrapper around IPFS and uses it exclusively for storage, the thesis aims to use any storage provider – traditional providers and IPFS – to homologate them using a distributed architecture comprised of workers hosted on client infrastructure. In terms of redundancy, the thesis ensures that files are stored redundantly across multiple workers, providing mechanisms to recover from worker node failures and guaranteeing that files are always available for retrieval. This contrasts with the earlier publication, which bases its redundancy on the permanent nature of IPFS.

Lastly, seeing both applications from a bird' s-eye view, we can distinguish that while the publication aims to accomplish that no usable copy of the file is stored on intermediary servers using IPFS as the main storage backbone, the thesis seeks to store files redundantly without being able to interpret their contents while homologating many storage providers into one storage pool.

Filecoin survey

Filecoin (Protocol Labs 2017) is a decentralized storage network that uses blockchain technology and IPFS (Protocol Labs 2014) to create a marketplace for buying and selling storage space, where participants are incentivized with Filecoin tokens to provide and verify storage services. It ensures data integrity and availability through mechanisms like Proof-of-Replication and Proof-of-Spacetime, allowing users to store and retrieve data securely and efficiently.

The Filecoin protocol inspires this thesis. It focuses on creating decentralized storage networks and aims to provide secure and reliable storage without relying on centralized entities. Additionally, it uses the principle of the network of independent storage providers to distribute and store data, which Filecoin implements to incentivize participants to contribute storage resources to the network.

Filecoin implements an incentive using its cryptocurrency to reward storage and retrieval miners. The thesis should incorporate a similar incentive mechanism in the future, possibly integrating a token system to encourage participation and reward users who provide storage resources. Additionally, Filecoin utilizes proofs such as Proof-of-Replication and Proof-of-Spacetime to ensure that data is stored, verifiably replicated, and maintained over time. The thesis should integrate similar proof mechanisms to enhance data integrity and reliability.

Filecoin builds on blockchain technology to provide a transparent, immutable ledger of all transactions, enhancing security and trust. Integrating blockchain into the thesis can be a double-edged sword since it can provide transparency and trustworthiness and severely impact performance and scalability. Also, Filecoin establishes verifiable markets for storage and retrieval, allowing transparent and decentralized transactions between clients and storage providers. This is another double-edged sword since the thesis can integrate verifiable contracts and payments to implement a public storage marketplace, but this can also negatively impact trust in closed organizations or consortiums. Finally, it is very important to note that Filecoin and the thesis, while having many similarities and common goals, focus on very different goals – Filecoin targets the global market, and the thesis targets many smaller installations.

Storj survey

Storj (Storj n.d.) is a decentralized cloud storage platform that utilizes a peer-to-peer network of nodes to store data securely and redundantly across the globe. By breaking files into smaller encrypted shards and distributing them to multiple nodes, Storj (Storj n.d.) ensures high availability, privacy, and security. Users can rent out their unused hard drive space and bandwidth, earning compensation in Storj (Storj n.d.) tokens, while data owners benefit from a scalable and resilient storage solution without relying on a single centralized provider.

Both the thesis and Storj (Storj n.d.) focus on distributed storage systems that aim to enhance data security, privacy, and availability through decentralization. The thesis proposes a system where storage nodes, or workers, contribute to a unified storage pool managed by a centralized control infrastructure, ensuring redundancy and high availability. Similarly, Storj (Storj n.d.) utilizes a decentralized approach, distributing encrypted data shards across multiple nodes worldwide to mitigate risks of data breaches and ensure consistent availability. Both systems emphasize security through encryption and redundancy, aiming to maintain data integrity and confidentiality even when some nodes fail or are compromised.

Inspired by Storj (Storj n.d.), the thesis could integrate more advanced peer-to-peer communication protocols to enhance node autonomy and reduce reliance on a centralized control point. Storj's (Storj n.d.) use of erasure codes instead of simple replication for redundancy could also be adopted to improve storage efficiency and reduce overhead costs. Additionally, the thesis could benefit from implementing a marketplace model like Storj (Storj n.d.), where storage providers are incentivized through a payment system for their contributions, potentially leading to greater scalability and resource optimization. Finally, adopting a more modular architecture as seen in Storj (Storj n.d.) could facilitate easier updates and maintenance, ensuring long-term sustainability and adaptability of the system.

Other literature survey

OctopusFS (Kakoulli and Herodotou 2017) a distributed file system manages multiple storage tiers, including memory, SSDs, HDDs, and remote storage, to optimize data management based on performance and capacity characteristics. It focuses on managing multiple storage tiers (memory, SSDs, HDDs, NAS) with pluggable policies for data management and emphasizes tiered storage. On the other hand, this thesis focuses on integrating different storage types into a unified storage pool, thus offering greater flexibility and usability. Additionally, OctopusFS utilizes a multi-master/slave architecture for scalability, with policies aimed at maximizing throughput and balancing load. However, the emphasis is on tiered storage management rather than a unified, scalable approach, which is the focus of the thesis. OctopusFS configures workers to manage data blocks on various storage media but does not focus on the autonomy and independence of worker nodes as strongly. The thesis incorporates the management of multiple storage tiers and optimizing data management based on performance and capacity characteristics from OctopusFS. It adapts the concept of tiered storage management to integrate various storage types into a unified pool, ensuring flexibility and usability.

The paper "A Blockchain-Based Hierarchical Semi-Decentralized Approach Using IPFS" (Athanere and Thakur 2022) presents a novel system for secure data sharing, combining the decentralized storage capabilities of IPFS with the security and immutability of blockchain technology. It outlines a two-level key management system to ensure data integrity and access control, aiming to eliminate single points of failure. It primarily relies on IPFS for decentralized storage, limiting the integration capabilities with other types of storage media, while the thesis supports multiple storage types. Additionally, the paper does not focus on creating a unified storage pool like the thesis but instead focuses on managing data in a more segmented manner using IPFS, thus limiting ease of use and possible use cases. Furthermore, blockchain-based systems like this paper suffer from higher communication overhead due to the need to broadcast transactions to the blockchain and synchronize blockchain nodes. On the other hand, by integrating a centralized control point, this thesis reduces the communication overhead typically associated with purely decentralized systems, improving performance and dependability. There are points that this thesis incorporates from this paper, like the principle of ensuring data integrity and access control through a hierarchical semi-decentralized approach. It adopts a two-level key management system to eliminate single points of failure and supports multiple storage types for enhanced flexibility.

Both this thesis and FileDES (Minghui, et al. 2024) aim to provide secure, scalable decentralized storage solutions, leveraging client-side encryption to ensure data privacy and eliminate single points of failure. This novel application distinguishes itself by integrating diverse storage types and employing a centralized control infrastructure for efficient node management and consistent performance, whereas FileDES (Minghui, et al. 2024) focuses on decentralized, encrypted storage without explicitly addressing such integration. However, FileDES's (Minghui, et al. 2024) use of advanced proof systems like Proof of Encrypted Storage and rollup-based batch verification for scalable proof generation and verification, along with its robust incentive mechanisms, presents areas for potential enhancement in this application. The thesis leverages client-side encryption to ensure data privacy and eliminate single points of failure from FileDES and it integrates diverse storage types and employs a centralized control infrastructure for efficient node management and consistent performance.

Concrete implementation of the System

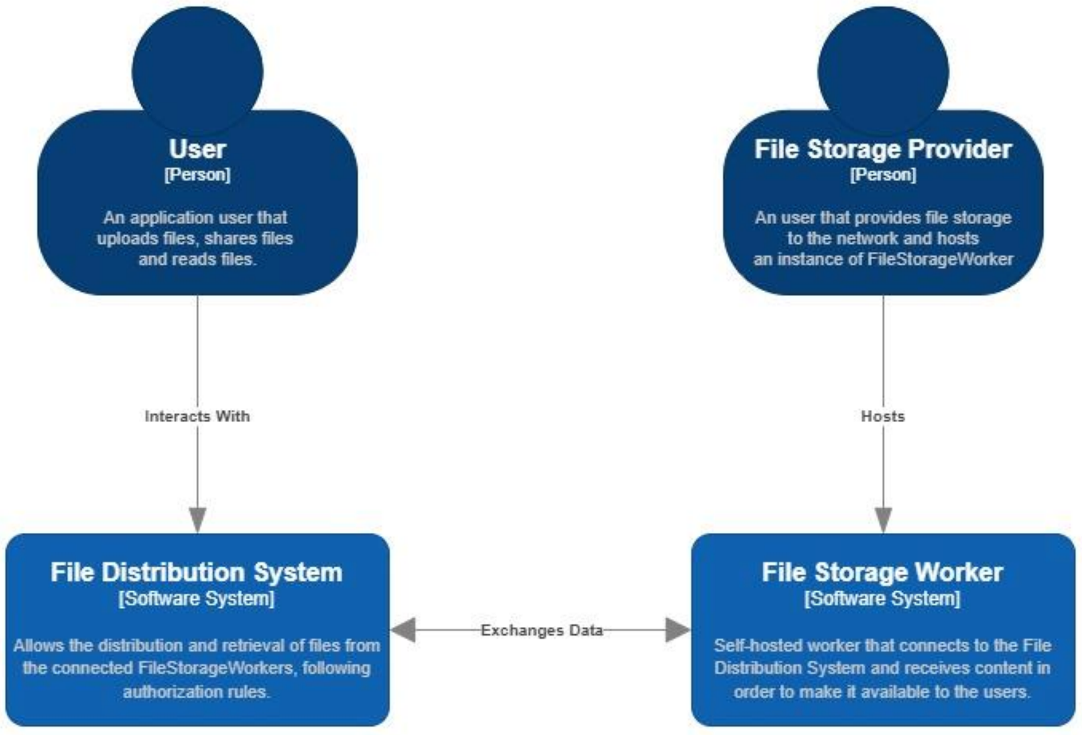
User
[Person]
An application user that uploads files, shares files and reads files.
File Storage Provider
[Person]
An user that provides file storage to the network and hosts an instance of FileStorageWorker
Interacts With
Hosts
File Distribution System
[Software System]
Allows the distribution and retrieval of files from the connected FileStorageWorkers, following authorization rules.
Exchanges Data
File Storage Worker
[Software System]
Self-hosted worker that connects to the File Distribution System and receives content in order to make it available to the users.

Figure 1: System Context Diagram

Figure 1 is focused on a bird's eye view of the software system illustrating the big picture of the system. The primary focus of this diagram is the users and their roles. The two types of users in the landscape are the user, who can upload files to the system, verify the existence of any file, fetch his files, share a file with another user, and transfer the ownership of a file to another user or group of users. The other user type is the File Storage Provider, which hosts an instance of the File Storage Worker and provides storage resources to the network through it.

The File Distribution system is the central node of this architecture and the main connection point for all application consumers. It serves the React-based frontend client to the users and exposes a WebSocket API—through SignalR—where the File Storage Workers connect to and receive commands. It is deployed on publicly accessible infrastructure in Docker Containers and supports horizontal and vertical scaling using a Redis SignalR backplane and a Kubernetes cluster.

The File Storage Worker is the distributed node of this architecture and is deployed in infrastructure owned and operated by the File Storage Providers. It consumes many different storage media provided by the client and receives commands from the File Distribution System over the SignalR connection. It also contains a local database that holds the records of the individual chunks of files stored along with their location. For it to connect to the network, an API key needs to be requested from the File Distribution System and provided upon the First connection with the SignalR Hub. Furthermore, it requires the connection configuration to the available storage mediums along with its size limit.

Technical Components of the System

File Storage Worker Analysis

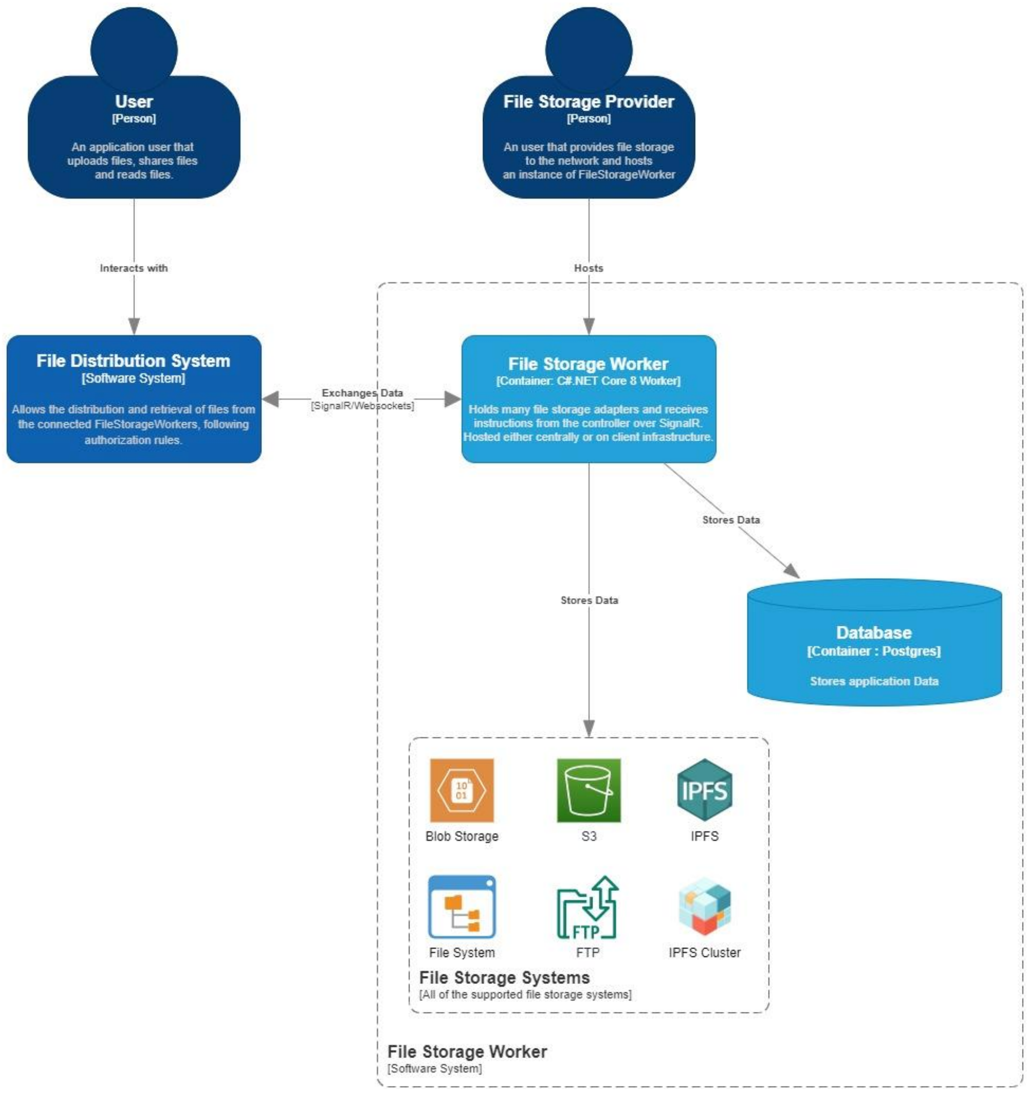
User
[Person]
An application user that uploads files, shares files and reads files.
File Storage Provider
[Person]
An user that provides file storage to the network and hosts an instance of FileStorageWorker
Interacts with
Hosts
File Distribution System
[Software System]
Allows the distribution and retrieval of files from the connected FileStorageWorkers, following authorization rules.
Exchanges Data
[SignalR/Websockets]
File Storage Worker
[Container: C#.NET Core 8 Worker]
Holds many file storage adapters and receives instructions from the controller over SignalR. Hosted either centrally or on client infrastructure.
Stores Data
Stores Data
Database
[Container : Postgres]
Stores application Data
Blob Storage
S3
IPFS
File System
FTP
IPFS Cluster
File Storage Systems
[All of the supported file storage systems]
File Storage Worker
[Software System]

Figure 2: Container Diagram - File Storage Worker

Figure 2 provides a more detailed view of the File Storage worker and its interactions with the software ecosystem. It is a component written in C# .NET Core 8 (Microsoft 2024) and utilizes a SignalR client to connect to the File Distribution System. Every worker is either owned by the host of the File Distribution System or a File Storage Provider, who is anyone with available storage resources to add to the distributed network. For the worker to connect and use the available storage resources, it accepts the connection options in the form of connection strings in its startup configuration. The worker can interact with any storage using the Simple Storage Service (S3) API, like Amazon S3, Ceph, etc. It can also interact with Azure blob Storage and any file System, either over File Transfer Protocol (FTP) / Secure Shell (SSH) File Transfer Protocol (SFTP) or directly using a mount. Lastly, it can control private or public InterPlanetary File System (IPFS) nodes and nodes in an IPFS cluster. Because the storage providers are implemented using the Adapter and Factory Design Patterns, it is quite easy to add more in the future if support for more storage types is needed. In the startup configuration, it is necessary to add a limit to the amount of storage the worker can provide to the network.

When the worker first connects to the File Storage and Key Management Service SignalR hub, the unique worker identifier is provided to authenticate and identify the worker. After that, the worker broadcasts all the files that it contains from previous executions, to synchronize with the service and receive a deletion event for any files needing to be deleted. After the synchronization process, if the worker has more than half of the total storage capacity available, a request is made to the Service, and the least represented chunks of data get stored in the worker.

The worker, after the initialization process, provides a set of available methods for the Service to invoke. Primarily, it provides the method for file chunk storage, which receives the file chunk's unique identifier and the file chunk in the form of an array of bytes. If there are multiple storage providers available, it runs a decision algorithm to select the appropriate provider. After that, it saves the file chunk and registers it in the correlation database along with the file chunk's unique identifier and the location where it is stored. Finally, it returns to the Service an indication that the operation was successful.

Another method provided by the worker is the retrieval of a file chunk using its unique identifier. Firstly, the database is queried to fetch the location of the file chunk, and the appropriate storage provider is requested from the File Repository Factory implementation. Then the file is fetched and either returned as a response or returned using a Hub method invocation, depending on the provided parameters. Similarly, there is the Deletion event, where a file chunk identifier is requested for deletion and the metadata entry along with the file chunk is deleted. In case the file is saved in IPFS, the metadata entry is deleted, and the CID is un-pinned, causing the file chunk to become garbage, as a result allowing for a “destroy” operation in an IPFS node, something not natively supported by the design of IPFS (Politou, et al. 2020).

Lastly, to optimize the performance of the Service on File upload, a method is created in the worker that accepts a file size and returns whether there is enough room to accept that file. This is used by the Service to decide which workers should send the file without making complex queries to the Service Database.

The above design of the worker has a lot of advantages – the most significant being that it does not have any knowledge of the contents stored within it. This happens because it hosts parts of encrypted files and does not know which chunks compose the original encrypted file, or their order. Any file within the system after it is split into chunks, can be stored partially in many mediums, e.g. Some chunks of the file are stored in IPFS and some other chunks in S3 storage. This further ensures that even in the event of the worker’s storage being compromised by an attacker or the storage medium not being fully trusted (Karapapas, Polyzos and Patsakis 2024), the data is unusable, and the original files cannot be retrieved.

File Distribution System Analysis

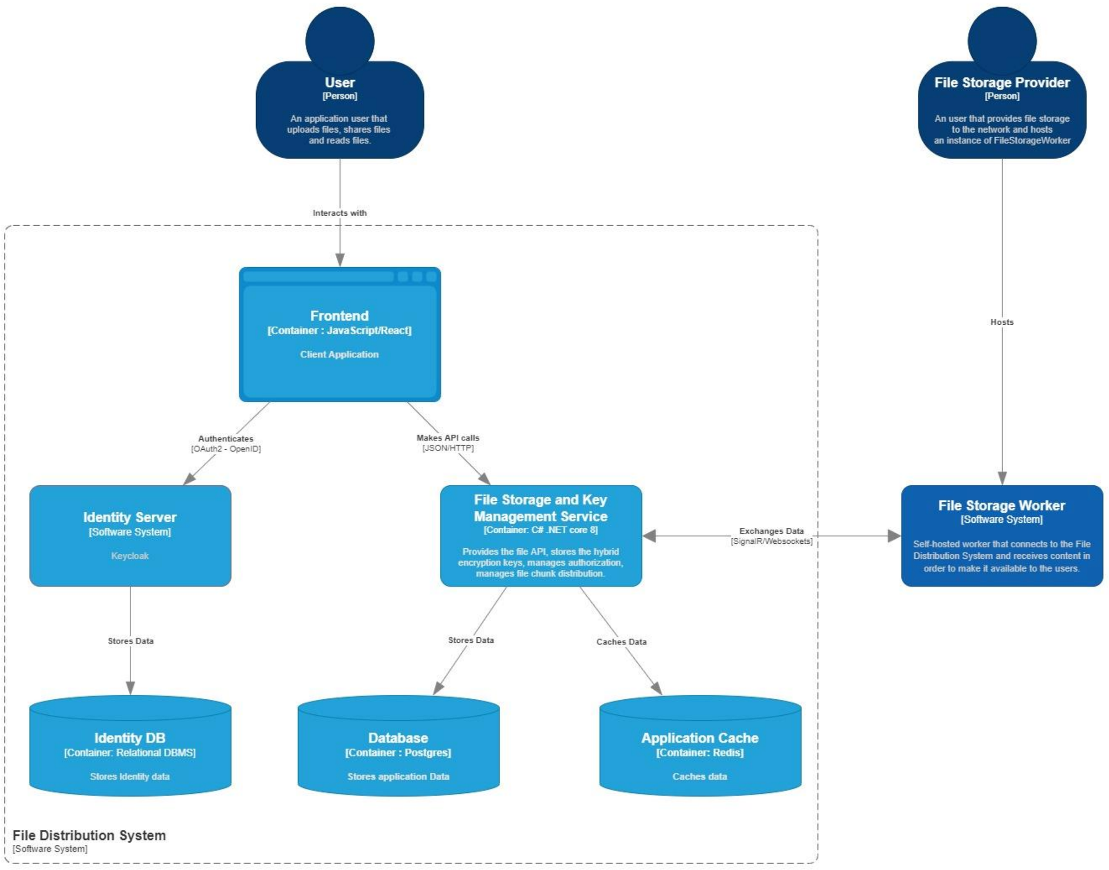
User
[Person]
An application user that uploads files, shares files and reads files.
File Storage Provider
[Person]
An user that provides file storage to the network and hosts an instance of FileStorageWorker
Interacts with
Hosts
Frontend
[Container : JavaScript/React]
Client Application
Authenticates
[OAuth2 - OpenID]
Makes API calls
[JSON/HTTP]
Identity Server
[Software System]
Keycloak
File Storage and Key Management Service
[Container: C# .NET core 8]
Provides the file API, stores the hybrid encryption keys, manages authorization, manages file chunk distribution.
Exchanges Data
[SignalR/Websockets]
File Storage Worker
[Software System]
Self-hosted worker that connects to the File Distribution System and receives content in order to make it available to the users.
Stores Data
Stores Data
Caches Data
Identity DB
[Container: Relational DBMS]
Stores Identity data
Database
[Container : Postgres]
Stores application Data
Application Cache
[Container: Redis]
Caches data
File Distribution System
[Software System]

Figure 3: Container Diagram - File Distribution System

To further understand how the File Distribution System fits in and interacts with the overall environment, we will zoom in on the previous diagram and end up with a container diagram of the File Distribution System in Figure 3. We define a container as a separately runnable and deployable unit that executes code, contains an external service, or stores data. The technology used to achieve that result is Docker, which is the most widely used open-source containerization platform. It packages the developed services or external applications into standardized executable components by combining the source code with any needed Operating System libraries and dependencies into a self-contained container. This zoomed-in view of the diagram illustrates the high-level architecture of the software system and how data and responsibilities are distributed across it. It also highlights the major technological choices and how the containers communicate with each other.

The Containers of the File Distribution System can be grouped into three distinct layers. The first layer is the client-side application, written in JavaScript using React. The second layer is the server-side application layer, which contains the Identity Server, based on Keycloak, and the File storage and Key management Service, written in C# using .NET core 8 (Microsoft 2024). The last layer is the Data storage layer, which contains the Databases used by the above services, in a Postgres Relational Database Management System (RDBMS) Container, and the Application cache used by the File storage and Key Management Service in a Redis-Stack Container.

Frontend Client

The Frontend client is a JavaScript application based on React (Meta Open Source n.d.). It allows users to interact with the available Services in a user-friendly way. Integrating with the Keycloak Identity Server uses the @axa-fr/react-oidc npm package, the @mui npm package for the UI components based on Google Material Design, and the browser’s crypto API for the cryptography functions.

When the user registers with the application, an account is created in the Keycloak Identity Server (Keycloak 2024). This account holds the login credentials and the user's role information. Immediately after the first registration, the browser will generate a set of asymmetric RSA-OAEP encryption keys and asymmetric RSA-PSS digital signature keys. After that, the user is prompted for a second password, different than the account password, and from its SHA-512 hash, a symmetric AES-GCM key will be generated, which will be used only client-side to encrypt and later decrypt the user's private encryption and digital signature keys. After that step, the public and encrypted private keys are sent to the Key Management Server and saved with the user's identity. After every login, the user is asked for the decryption password, the symmetric key is recreated, and the private encryption and signature keys are decrypted, thus allowing the user to access his content.

When a file gets uploaded to the frontend client, a random symmetric key is generated and used to encrypt the file contents. After that, the encrypted file is sent to the File Service, where its chunking and distribution occur. In parallel, the file's symmetric key is encrypted using the user's key, and the digital signature of the original file's contents is created using the user's key. The encrypted file's keys and metadata are then saved in the file service database. It is crucial for the security of the implementation that the file exists unencrypted only in the user's browser.

When a file's owner shares it with another user, its key is fetched and decrypted in the user's browser. After that, using the recipient's public key, a copy of the file key is encrypted and saved along with ownership metadata in the File Service's Key Management Service. When the file access is revoked, the file owner requests the deletion of this key copy from the Key Management Service, thus removing the authorization to that user.

When a user fetches a file, the appropriate keys are fetched from the server alongside it. The user's private key then decrypts the keys. Using the decrypted file keys, the file is decrypted, and its contents are validated using the owner's digital signature.

Other notable functionalities that the frontend client provides to the end-user are a user-friendly user interface to manage his files and a dashboard to discover other available users. A dashboard is also available to administrators that shows the available storage workers, their storage capacity, and their status.

Identity Server

The identity server selected for this application is Keycloak (Keycloak 2024), an Identity and Access Management (IAM) software. Its primary goal is to verify the identity of users or systems requesting access and to evaluate a set of rules determining what features and assets those users or systems can access.

Identity management and access management, though related, are distinct entities. Identity management focuses on verifying users' identities, while access management maps these verified identities to specific permissions within an environment. Typically, access management relies on an Identity Provider to verify the user. It then builds on this verification to define what the user or system can do within a given environment. Role-Based Access Control (RBAC) is a common method used to implement an access control layer.

A support layer, usually consisting of a set of protocols, is added to facilitate this access management in conjunction with other applications. OAuth2 is a standard protocol often used to manage access control parameters on the authorization software side and provide guidelines for applications consuming it. Keycloak integrates the functionalities of both an Identity Provider and an Authorization Server, offering a complete, open-source IAM solution. It supports out-of-the-box integrations with social login sites, other identity providers, and federated external user databases, such as LDAP servers.

In this novel application, a dedicated Realm has been established within Keycloak. Within this Realm, appropriate clients have been configured for each specific service, ensuring that all necessary access and identity management requirements are met. Local roles have been defined within each client for finer control over permissions. These roles specify the exact capabilities and access rights for users within the context of each service. Additionally, global compound user roles have been created to aggregate multiple local roles, providing a streamlined way to manage user permissions across the entire application.

The storage of this configuration data and user data is handled by a PostgreSQL database. This database is deployed as a Docker (Docker 2024) container, which provides an isolated and consistent environment for the database, enhancing both security and scalability. This setup ensures that data integrity and availability are maintained, even as the application scales.

Every user must initially register with the identity server before accessing the application. This registration process involves creating a unique user account within Keycloak, which then maps to the roles and permissions defined within the Realm. Once registered, users gain full access to the application's features according to their assigned roles. This streamlined registration and access management process ensures that only authenticated and authorized users can interact with the application, thereby enhancing security and user management efficiency.

File Storage and Key Management Service

The File Management Service is developed in C# utilizing .NET Core 8 (Microsoft 2024). This service is responsible for the encryption keys, authorization policies, and the control of worker processes related to user files. Specifically, it securely stores users' encrypted keys and manages the distribution and authorization of files shared by users among the workers. The authorization policies include multiple sharing options: sharing a file with another user, sharing with a group, sharing anonymously via a public URL, or keeping the file personal.

Internally, the File Storage and Key Management Service implements many Design Patterns and algorithms to ensure Data Availability and Data Integrity. The workers are connected to the Service using SignalR, a library that facilitates real-time web functionality, allowing server-side code to push content to clients instantly as it becomes available and supports bi-directional communication between server and client over WebSockets. This allows the Service to communicate with the workers and exchange information – file chunks and commands. Also, it allows for monitoring the connected workers, which in turn allows the system to spread the files of any worker that disconnects to the others that are available. Additionally, internally the Service uses the Command-Query Responsibility Segregation (CQRS) Design Pattern, which separates read and write operations for a data store, allowing for optimized and scalable handling of commands (writes) and queries (reads). This allows for the efficient handling of heavy tasks - like a file initially uploaded to the system – and the optimized and fast composition of a file from the chunks redundantly stored in the distributed workers.

```
Algorithm Worker Connection
Algorithm WorkerConnection(worker_id, available_capacity):

if worker_id is registered worker then
    Save worker.status = active
    Save worker.available_capacity
    stored_chunks[] = Send a request to the worker to list all stored file chunks
    outdated_chunks[] = find chunks in stored_chunks[] that do not exist in the current system

    for chunk in outdated_chunks[] do
        Send Delete request to the worker for the chunk
    end for

    least_represented_chunks[] = Query database for least represented chunks across all workers
    Filter least_represented_chunks[] to fit in worker.available_capacity
    for chunk in least_represented_chunks[] do
        Send chunk to worker
        Update new chunk location in database
    end for
else
    Discard connection request
end if
```

Figure 4: Worker Connection Algorithm

To thoroughly understand the mechanisms implemented within the Service, it is necessary first to explain the foundational principles. The first Algorithm is the Worker connection algorithm, outlined in Figure 4. For the worker to connect to the Service, it must first be registered by an administrator and assigned a unique identifier, which will be, in turn, validated during the establishment of the SignalR connection. Upon the client's first connection, the connection Identifier and worker's available data capacity are saved in the database along with the connection status. After that, to synchronize the worker and the Service, the worker sends the identifiers of the chunks of files it contains from previous connections. Any chunks of files present in the worker of earlier connections and deleted from the Service in the timespan that the worker has been offline are found, and garbage is collected. Any chunks still relevant to the service are registered as existing. After that, it is populated with the chunks of data it should host, selected using an algorithm that queries the database, and finds the chunks of files least represented in the network, which sums up to half of the available capacity of the newly connected worker. The highest priority chunks are the ones that are represented three or fewer times in the network and are always added to newly connected workers. When a worker disconnects or the connection drops, its connection status is updated in the database, and the chunks of files hosted by the worker are distributed among the other available workers. The goal of this mechanism is always to ensure that all the file chunks are distributed to at least three different workers, allowing up to two concurrent failures of workers before any data is lost or temporarily unavailable. To ensure that the Service is always available, it is run in multiple instances on different machines, and a Redis cluster backbone is used to synchronize and Scale the SignalR hub. The database is deployed in a cluster, regularly snapshotted, and backed up on multiple machines.

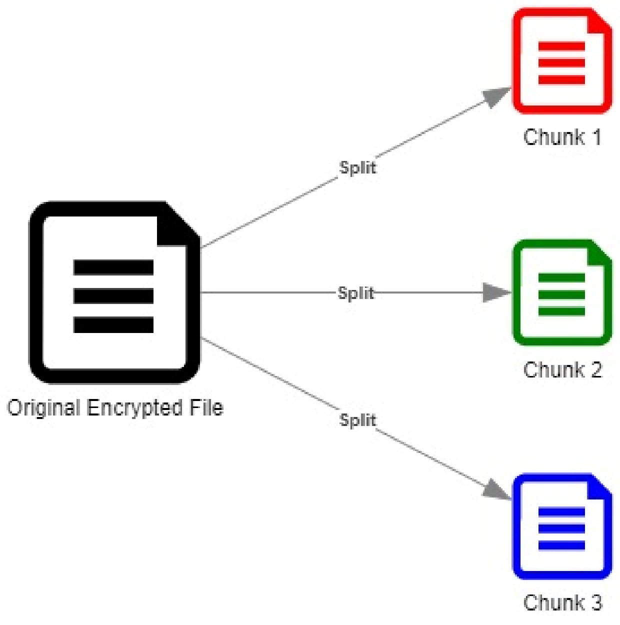

Figure 5: File Splitting into chunks

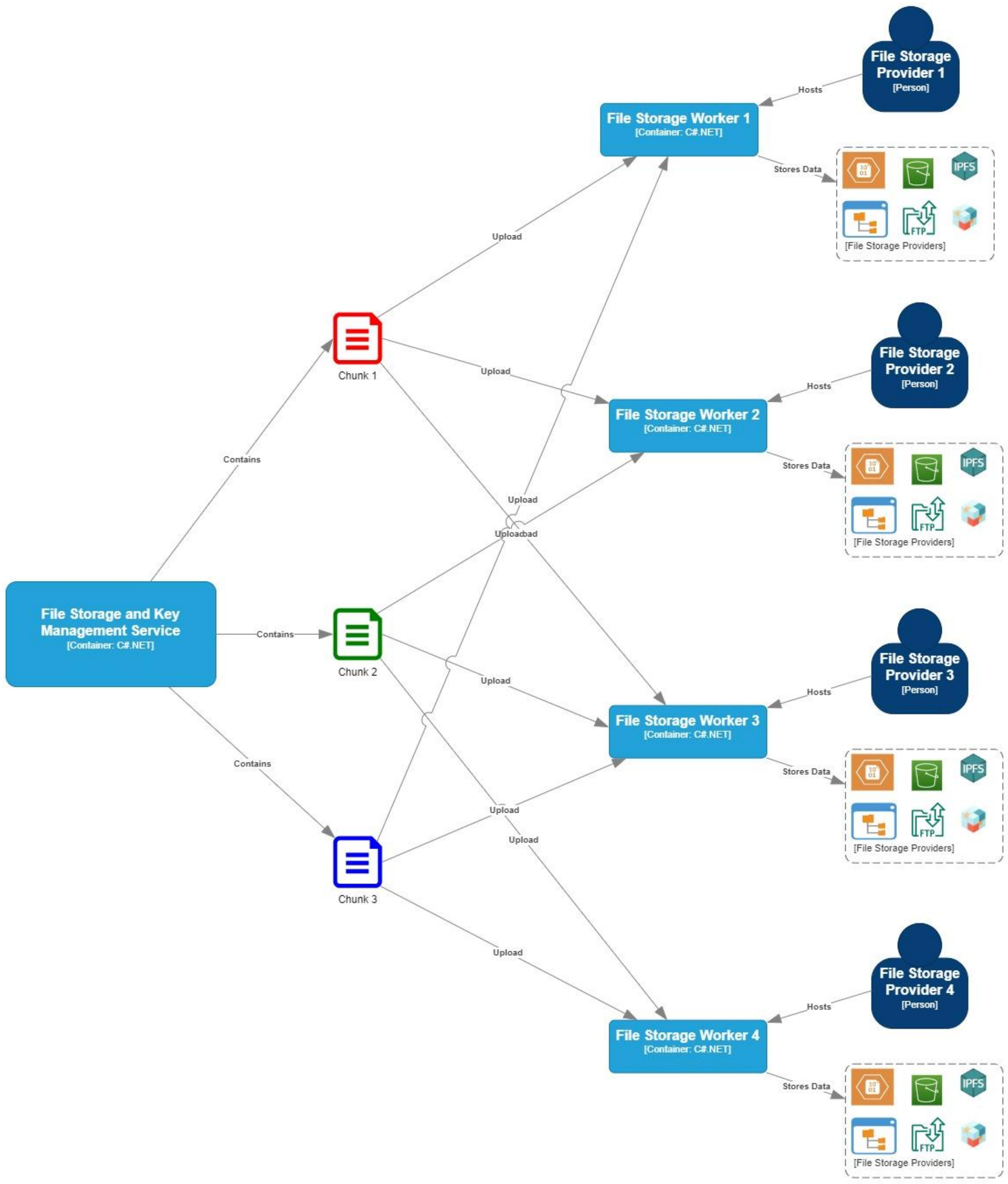

Figure 6: File Upload Process.

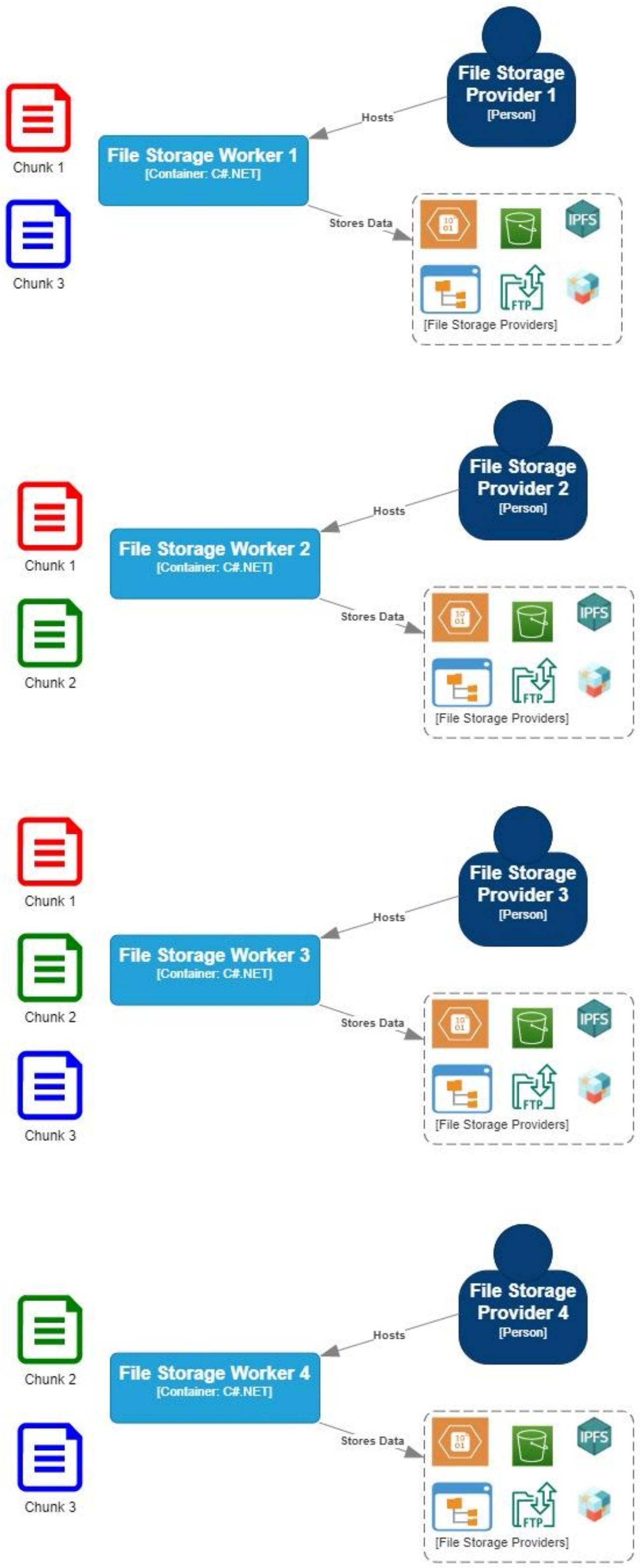

Figure 7: State of workers after file upload.

```
Algorithm  File Upload Process
Algorithm FileUpload(file, user_id):

file_metadata = extract metadata from file (hash, file_size, content_type, user_id, Generate Unique File Identifier)
chunks[] = split the file into chunks of random size
chunk_metadata[] = extract metadata from each chunk (hash, size, parent file, order, Generate Unique Chunk Identifier)
Save file_metadata and chunk_metadata[] to the database

for each chunk in chunks[] do
    encrypted_chunk = Encrypt the Chunk using the server key
    selected_workers[] = Select the Workers from the Database that can accept the file
    if selected_workers[].length less than 3 then
        Abort the upload
    end if
    for each worker in selected_workers[] do
        Send the encrypted_chunk to the worker
        Update the Database with the worker.id the chunk is saved and Hash(encrypted_chunk)
    end for
end for
```

Figure 8: File Upload Algorithm

When a client-side encrypted file is initially uploaded to the Service, an algorithm illustrated in Figure 8, is run in a background job using Hangfire, a C# .NET core library that enables efficient background processing, that chunks the file and distributes it along the connected workers. Firstly, the size of the file and its metadata are saved to the database, and the file is assigned a unique identifier. After that, the file contents are split into chunks - as illustrated in Figure 5 - with the following rules: Files smaller than 1 byte cannot be uploaded. All files must be split into at least two chunks. The chunk size is from one byte up to 32 Megabytes. Each chunk's size is random. After the file is split into chunks, the metadata of each chunk – order, size, parent file identifier - is saved to the Database. After all the above, a message is broadcasted to the connected workers, with the size of the complete file, and any worker with available space for the whole file responds. If there are less than 3 workers with enough space to receive the entire file, the file cannot be uploaded. If there are exactly three workers with enough space to hold the entire file's chunks, they all receive all the shuffled file chunks, and all the chunks are prioritized for distribution once any new workers connect. If more than three workers respond, then the file chunks are shuffled and distributed partially to each worker following these rules: All file chunks cannot be present in one worker. All chunks must be saved to at least three workers. The chunks must be shuffled before distribution. An example of this process for three chunks and four workers is illustrated in Figure 6. Importantly, after the upload process is completed, the original encrypted file is never present whole in the Service. This mechanism ensures that any file uploaded to the system is distributed to multiple workers, providing redundancy in the event of a worker failure. Because of the minimum replication factor of three copies of all chunks of files, the storage is efficiently managed – one-third of the available storage is usable - but the system can only recover from two concurrent worker disconnects or failures.

When a file is queried from the Service, the algorithm illustrated in Figure 9 is run. Firstly, the chunks are retrieved from the database, the chunk identifiers are then broadcasted to all connected clients, and the response for each chunk is awaited. Each file chunk request is resolved once the first available worker responds with the correct content, which is validated by its SHA-512 hash, both encrypted by the server and decrypted, and all further duplicate responses are ignored. An example of this process is illustrated in Figure 10. Once all the file requests have been resolved, the file chunks are aligned according to their order, and the final response is generated and returned to the client, as illustrated in Figure 11. If all the available clients respond and at least one of the chunks of the file is not found, then the file is invalid and not returned. Because the fastest worker response for each file is used, the file retrieval time is relative to the quickest response for each chunk. Finally, the file is reassembled and validated by its SHA-512 hash. When a file is requested multiple times, and if there is enough storage in the network, a job is triggered to replicate the file’s chunks to more workers than the replication factor of three, further accelerating the query times. Notably, the editing of a file is implemented by first uploading the new version of the file to the system and then removing the old version. Due to this, it is straightforward to maintain a version log of the file's history by not deleting the old versions.

Last but not least, the Service is responsible for acting as the Key Management Service and storing the File Access Control List (FACL) for each file. When a user generates the asymmetric keys in the browser and encrypts the private asymmetric key with the password-based symmetric key, the resulting ciphertext and public key are stored in a Database Table and the user’s unique identifier. The Private key ciphertext can only be accessed by its owner, and anyone can access the public key. When the user shares a file with another user, the file’s symmetric key is encrypted with the recipient’s public asymmetric key, and the resulting ciphertext, along with both user’s unique identifiers, is stored in the FACL. When access to a user is revoked, the corresponding entry in the FACL is removed. Before a user can fetch a file, the FACL entry is checked, and only if it exists in the file is the decryption key returned to the user. This ensures that the files are accessible only by the authorized users and only for the span of time that the authorization is active. If a user that is not authorized tries to access the file, the server will not run the algorithm to assemble the encrypted file from the workers. Furthermore, the user will be unable to access the decryption key because it will be deleted from the database, resulting in a secure way to de-authorize a user.

Algorithm File Retrieval Process

```
Algorithm FileRetrieval(file_id):

metadata = Query File metadata from the database
chunk_metadata[] = Query chunk metadata from the database
valid_chunks = []

for chunk_metadata in chunk_metadata[] do
    Send a message to all workers with the chunk_metadata.chunk_id
    repeat
        chunk = wait for chunk response by worker
        if Hash(chunk) not equals chunk_metadata.encrypted_chunk_hash then
            Discard chunk (tampered)
        end if
        decrypted_chunk = decrypt the chunk using the server key
        if Hash(decrypted_chunk) equals chunk_metadata.hash then
            valid_chunks.append(decrypted_chunk)
        else
            Discard chunk (tampered)
        end if
    until Retrieval of valid chunk
end for

reconstructed_file = Using the order stored in the metadata, reassemble the original file by reordering the valid_chunks

if Hash(reconstructed_file) == metadata.hash then
    return reconstructed_file
else
    return Error
end if
```

Figure 9: File Retrieval Algorithm

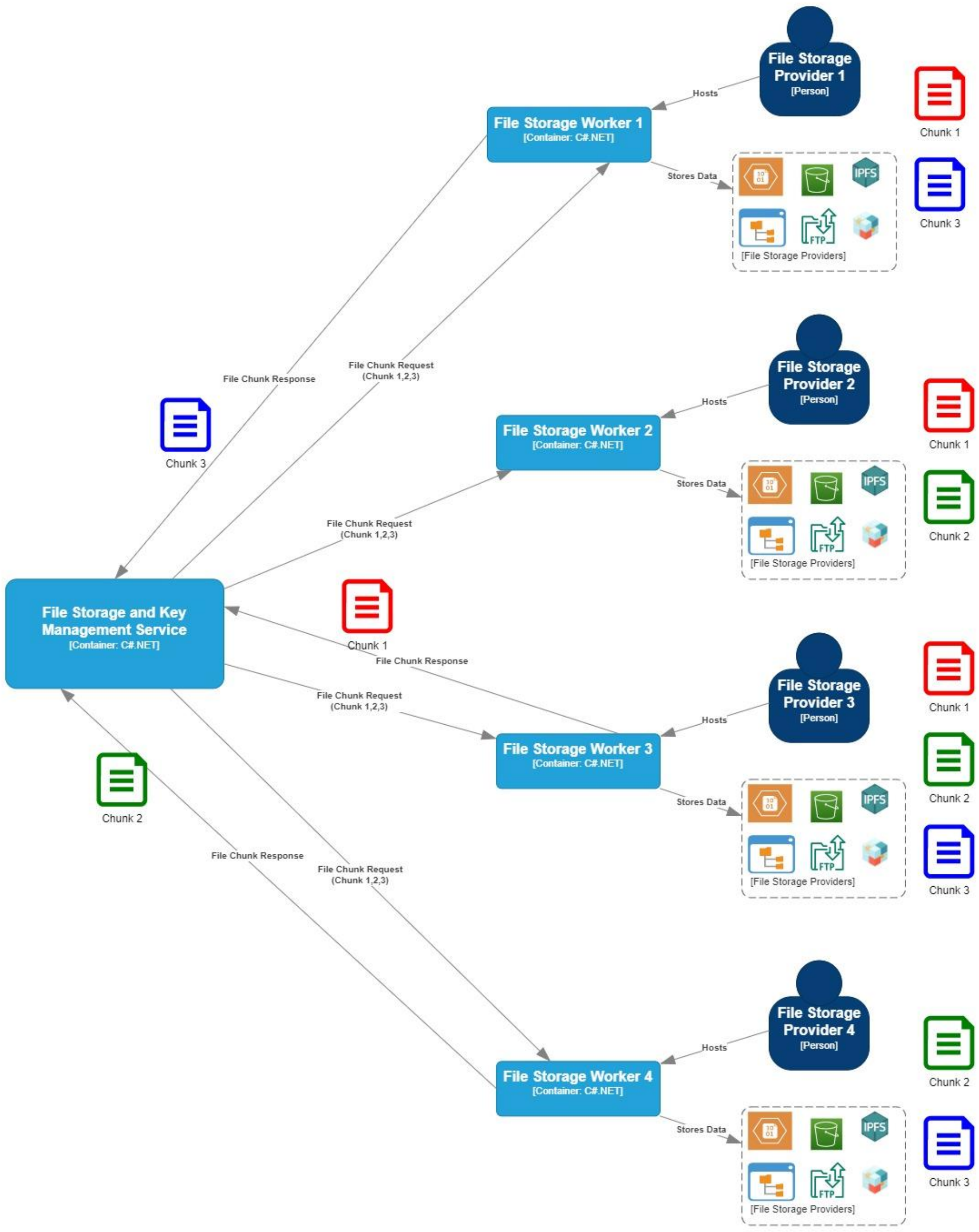
File Storage Provider 1 [Person]
Hosts
File Storage Worker 1 [Container: C#.NET]
Stores Data
[File Storage Providers]
Chunk 1
Chunk 3
File Chunk Response
File Chunk Request (Chunk 1,2,3)
Chunk 3
File Storage Provider 2 [Person]
File Storage Worker 2 [Container: C#.NET]
Chunk 1
Chunk 2
File Storage and Key Management Service [Container: C#.NET]
Chunk 1
File Chunk Response
File Storage Provider 3 [Person]
File Storage Worker 3 [Container: C#.NET]
Chunk 1
Chunk 2
Chunk 3
Chunk 2
File Storage Provider 4 [Person]
File Storage Worker 4 [Container: C#.NET]
Chunk 2
Chunk 3
IPFS
FTP

Figure 10: File Retrieval Process.

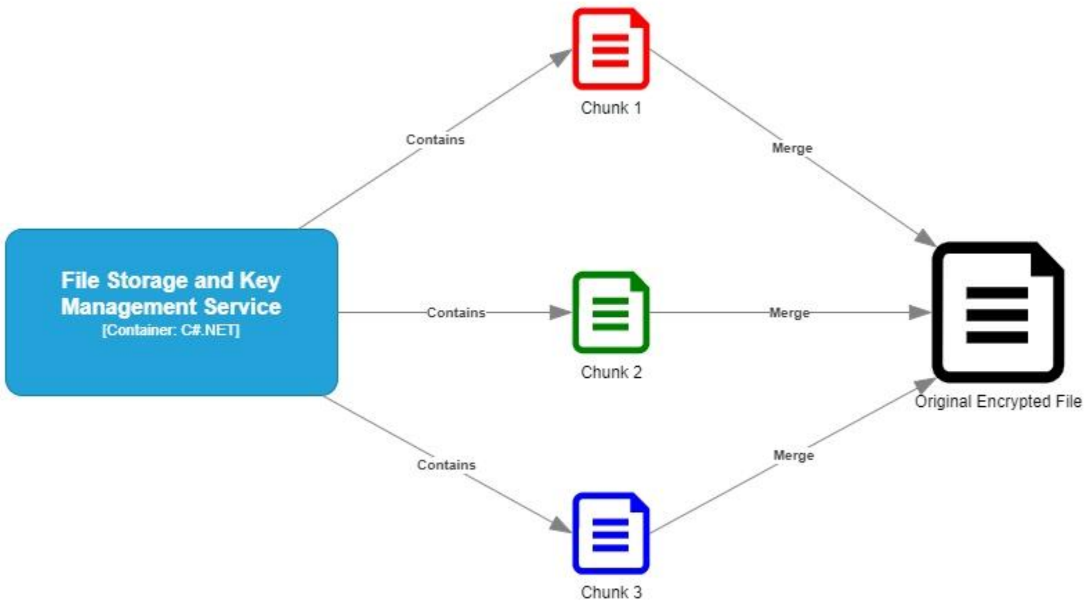

Figure 11: File Reassembly from chunks.

Another very important mechanism that runs in the file storage service is the ability to detect when there is a graceful or ungraceful exit of a worker and the system's overall health. This uses some mechanisms implemented in the service, the first of which is the SignalR polling mechanism, a health check built in the SignalR server that determines if a connection is still alive or should be garbage collected. If it is garbage collected, an event fires with the connection identifier, which is used by the service to determine from the database the worker identifier, and the files that are stored in the worker are distributed to other available workers. The second mechanism implemented to achieve this is a periodical content ping, where the server broadcasts the request for the hash of some chunks to the clients. All clients' responses are awaited and validated against the database records, and any chunk copies that have differences are removed and added to another worker.

Performance evaluation

This section of the thesis presents a performance evaluation of the thesis. Performance is critical in this application because it is designed to integrate various storage types and manage many user-hosted workers. Therefore, it needs to scale efficiently to handle increasing amounts of data and users. The performance also directly impacts the system's ability to provide redundancy and high availability. Quick recovery from node failures and efficient distribution of file chunks across nodes make it possible on a large scale.

Evaluation of System Performance using the Message Quantity metric

File Upload Process

To calculate the number of messages sent during the file upload process, we follow the steps outlined in the system overview:

- Broadcast to Workers: The service broadcasts a request to all connected workers for available storage space. Assuming n workers, Messages: **n**
- Worker Responses: Each worker responds with their available space. Messages: **n**
- Chunk Distribution: The service splits the file into chunks and sends each chunk to at least three workers. Assuming m chunks, Messages: **3*m**
- Worker Acknowledgment: Each worker acknowledges the receipt of each chunk. Messages: **3*m**

Total Messages for file upload: n + n + 3*m + 3*m = **2*n + 6*m**

File Retrieval Process

To calculate the number of messages sent during the file retrieval process, we follow the steps outlined in the system overview:

- Broadcast to Workers: The service broadcasts a request message to all connected workers with unique file chunk identifiers. Assuming n workers, Messages: **n**
- Worker Responses: Each worker responds with the requested file chunks. We only use the first response for each chunk and discard the rest. Assuming m chunks: Messages: **m**

Total Messages for file retrieval: **n + m**

Worker Connection Process

To calculate the number of messages sent during the worker connection process, we follow the steps outlined in the system overview:

- Worker Registration: The worker sends a connection request to the Service with its unique identifier and available storage capacity. Messages: **1**
- Synchronization Request: The Service requests the list of file chunks currently stored on the worker to synchronize its records. Messages: **1**
- Worker Response: The worker responds with the list of file chunks it currently holds. Messages: **1**
- Deletion of outdated chunks: The Service sends to the worker the list of file chunk unique identifiers to delete. Messages: **1**

- Worker Population: The service Sends the worker the least represented file chunks in the network to save. Assuming m chunks and n workers. Messages: n-1 + m (retrieval) + m (Save) = **2*m + n – 1**

  Total Messages for worker connection Process: 1 + 1 + 1 + 1 + 2*m + n – 1 = **2*m + n + 3**

Worker Disconnection Process

To calculate the number of messages sent during the worker disconnection process, we follow the steps outlined in the system overview:

- Broadcast to Remaining Workers: The Service broadcasts a request to remaining connected workers. Assuming n workers, Messages: **n**
- Remaining Workers Respond: Each remaining worker responds with available storage capacity. Messages: **n**
- Chunk Redistribution: The Service distributes the file chunks found in the database to other workers. Assuming the disconnected worker had m chunks and each chunk needs to be replicated to one other worker: n + m (retrieval) + m (save) + m (acknowledgment) = **n + 3m**

  Total messages for worker disconnection process: n + n + n + 3*m = **3*n + 3*m**

Evaluation of System Performance using Time-based metrics

To evaluate the system performance, we will record the time needed to complete some frequent operations, such as uploading and retrieving a file, using 3 and 4 worker instances, all uploading to AWS S3. The File Storage Service and the workers are deployed in a local network Virtual machine (4 cores, 4GB RAM) inside docker containers in an Ubuntu server 24.04 host running in a local network VMware ESXi hypervisor. The timing is calculated using the C# .NET System Diagnostics and starts upon arrival of the request to the controller method and ends on the response generation.

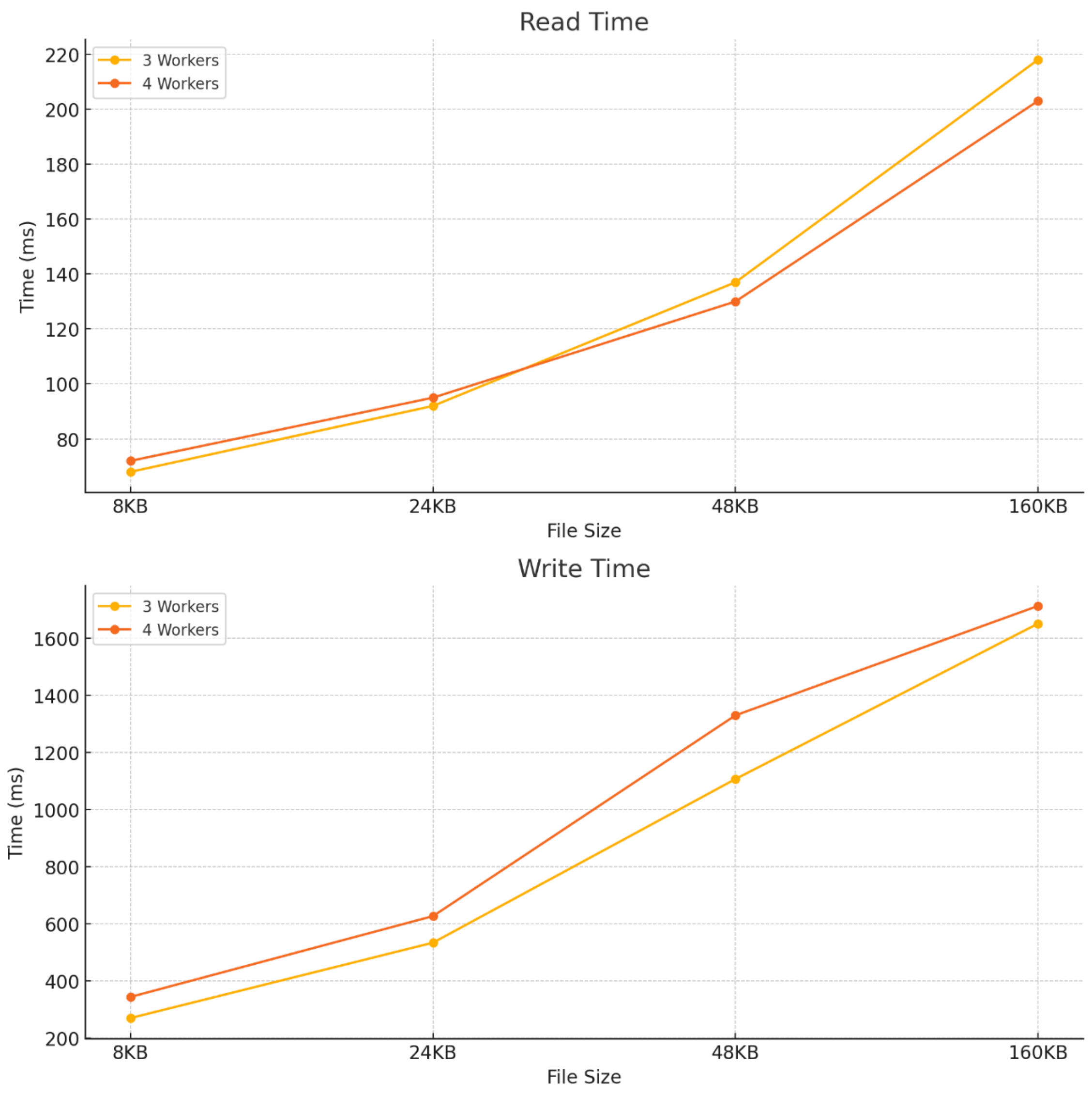

Figure 12: Performance Evaluation Graph

Analyzing the graph of Figure 12 We can see that the read-and-write performance scales linearly as the file size grows. Additionally, adding more workers decreases the read times due to the first response mechanism, and increases write times due to the additional communication overhead. These numbers indicate that the read performance scales efficiently with the addition of more workers, and the write performance scales linearly with file size. Adding more workers to the write operation causes some delays, most probably caused by IO bottlenecks and network delays.

Performance Evaluation Conclusions

The system's performance evaluation outlined some strengths and some areas for improvement. The system can efficiently scale the read performance by adding more workers, mainly due to the selection of the first valid response from the workers. The main bottleneck of the read operations is the network delays between both File Service and Workers and workers and storage infrastructure. The read performance can be improved by adding a cache to the workers and introducing a storage classification, where the most frequently accessed chunks get stored in the most performant workers and cached locally to the File Service.

On the other hand, the write performance, while it scales linearly with file size, experiences a decrease in performance with the addition of more workers. This is because the file was uploaded to multiple workers, generating a traffic spike. This can be heavily improved by adding a write cache layer, where the chunks are locally cached to the File Service and incrementally sent to the workers when network resources are available. Another technique that can significantly improve overall system performance is the addition of a gossip protocol, where the workers can communicate directly with each other to more efficiently spread data on file upload and worker exit.

Evaluation of Threat Mitigation Strategies

The proposed solution addresses the previously mentioned threat assumptions through a combination of security measures and architectural design choices. To mitigate the risks associated with eavesdropping, the system is designed with the principle that all data transmitted between nodes and the central control infrastructure is encrypted and only partial chunks, meaning that a whole file is never transmitted to workers, ensuring that even if data is intercepted, it remains unintelligible to attackers. To counter denial-of-service (DoS) attacks, the system incorporates traffic filtering and rate-limiting mechanisms through which it communicates with the workers. These measures help maintain service availability and protect against downtime and data loss.

For the threats related to individual storage nodes or workers, the system assumes these cannot be fully trusted and may be compromised or abused. To mitigate these risks, each client-side encrypted file is split into further encrypted chunks, with each chunk distributed across multiple nodes. This approach ensures that even if a node is compromised, the attacker cannot access complete data, preserving data integrity and confidentiality. Additionally, the system implements authorization and access control policies, preventing storage nodes from performing unauthorized operations or accessing data beyond their permitted scope. The system uses client-side encryption to address the threat of authorized entities attempting unauthorized access to other files, ensuring that only entities with the correct decryption keys can access the data. Additionally, before a file is returned to a user, the user's authorizations are checked, meaning that if there are no keys in the Key Management Service, the file will not be returned to the user.

The system also addresses the challenges posed by periodic worker failures and disconnections by the system's redundancy and failover mechanisms. Data is stored redundantly across multiple workers, allowing the system to recover and maintain data availability even when some fail. Furthermore, the system continuously monitors the status of each worker and redistributes data as needed to ensure redundancy. In the case of disconnections caused by network issues or maintenance activities, the system's design ensures that data accessibility is maintained by reallocating tasks to connected workers.

Use Cases

This thesis presents a versatile distributed storage solution that can be applied across various use cases, particularly benefiting large organizations and corporations. These entities often require robust backup and disaster recovery systems to safeguard critical data. The proposed distributed architecture ensures data redundancy and high availability, making it an ideal choice for maintaining business continuity in the face of hardware failures or cyberattacks. Additionally, the system's ability to integrate and unify different storage mediums allows organizations to optimize their existing storage infrastructure, reducing costs and improving efficiency. For instance, corporations can leverage local, cloud, and network-attached storage mix to create a cohesive storage pool that meets their diverse needs.

This storage solution can greatly benefit educational institutions such as universities and schools. These institutions often face short-term bulk storage requirements, such as during exam periods or research data collection phases. The flexibility and scalability of the proposed system make it well-suited for handling such temporary storage demands efficiently. Furthermore, the system's encryption and access control mechanisms ensure the privacy and integrity of sensitive academic and research data. The capability to perform due diligence, coupled with features like IPFS destruction, enables secure and thorough data management, ensuring that obsolete or sensitive data can be irreversibly deleted when necessary. This makes the system a valuable asset for educational institutions aiming to enhance their data storage and management capabilities.

Conclusion

Limitations

Despite its advantages, the proposed system has several limitations. One significant limitation is the reliance on a central control point, which introduces a potential single point of failure, compromising the system's overall resilience. Additionally, the requirement for a minimum of three workers to ensure adequate data redundancy means that the system will not function with fewer nodes, limiting its applicability in smaller-scale deployments. The replication factor of three also results in higher storage overhead, as only one-third of the total storage capacity is usable for actual data storage. Furthermore, hosting workers on client infrastructure introduces variability in performance depending on the client's resources and available network infrastructure, potentially affecting the system's reliability and efficiency.

The cryptographic implementation, while robust, also presents limitations, particularly regarding key management. If a file encryption key becomes compromised, the affected file will be re-encrypted with a new key. This resource-intensive process requires secure re-distribution of new keys and re-encryption of data across distributed nodes. Ensuring that all instances of the compromised key are replaced without data loss or unauthorized access adds another layer of complexity. Finally, while the system currently integrates various storage mediums, its scalability and performance could be challenged by the need to handle increasingly diverse storage types and large volumes of data and users.

Future work

This thesis presents a novel application we developed to homologate user storage contributions into a unified pool, effectively integrate diverse storage types, and ensure high redundancy and availability. The system achieves scalable and efficient distributed node management by employing a centralized control infrastructure, enhancing overall performance. This approach optimizes resource utilization and ensures consistent and reliable access to stored data across the network.

Future work will focus on incorporating advanced proof systems, such as Proof of Encrypted Storage (Minghui, et al. 2024), Proof of Replication (Protocol Labs 2017), and Proof of Spacetime (Protocol Labs 2017) and incentive mechanisms (Protocol Labs 2017), possibly using blockchain technology to enhance data integrity and user participation. Furthermore, the system will undergo additional security enhancements to protect user data and ensure compliance with regulatory standards, ultimately strengthening the system's overall robustness and appeal. Cryptography implementation can be significantly improved by introducing more advanced cryptography concepts, like zero-knowledge proofs, proxy re-encryption, etc. Leveraging the system’s modular design, more storage connectors can be easily added in the future to accommodate more types of file storage. Lastly, many technologies available should be investigated, including the possibility of replacing SignalR transport, like IPFS Pub-Sub, OrbitDB, and WebRTC. Lastly, the findings of the performance evaluation can be implemented with the goal of significantly accelerating the read-and-write performance of the system and improving the efficiency of the system.

Bibliography

Argyropoulos, Vasileios, Constantinos Patsakis, and Efthimios Alepis. 2022. "Semi-Decentralized File Sharing as a Service." *2022 13th International Conference on Information, Intelligence, Systems & Applications (IISA).* Corfu, Greece: IEEE. 1-8.

Athanere, Smita, and Ramesh Thakur. 2022. "Blockchain based hierarchical semi-decentralized approach using IPFS." *Journal of King Saud University Computer and Information Sciences* 1523-1534.

Docker. 2024. *Docker.* Accessed 07 15, 2023. https://www.docker.com/.

Kakoulli, Elena, and Herodotos Herodotou. 2017. "OctopusFS: A Distributed File System with Tiered Storage." 05: 65-78.

Karapapas, Christos, George C. Polyzos, and Constantinos Patsakis. 2024. "What's Inside a Node? Malicious IPFS Nodes Under the Magnifying Glass." In *ICT Systems Security and Privacy Protection*, 149--162. Springer Nature Switzerland.

Keycloak. 2024. *keycloak.* Accessed 06 15, 2024. https://www.keycloak.org/.

Kubernetes. 2024. *k8s.* Accessed 06 15, 2024. https://kubernetes.io/.

Meta Open Source. n.d. *React.* Accessed 2024. https://react.dev/.

Microsoft. 2024. *C# .NET core 8.* Accessed 06 15, 2024. https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-8/overview.

Minghui, Xu, Zhang Jiahao, Guo Zhang, Cheng Xiuzhen, Yu Dongxiao, Hu Qin, Li Yijun, and Wu Yipu. 2024. "FileDES: {A} Secure Scalable and Succinct Decentralized Encrypted Storage Network." *CoRR* abs/2403.14985.

Politou, Eugenia, Efthimios Alepis, Constantinos Patsakis, Fran Casino, and Mamoun Alazab. 2020. "Delegated content erasure in IPFS." *Future Generation Computer Systems* 956-964.

Protocol Labs. 2017. "Filecoin: A Decentralized Storage Network." 07 19.

—. 2014. *IPFS.* Accessed 06 15, 2024. https://ipfs.tech/.

Storj. n.d. *Storj: A Decentralized Cloud Storage Network.* Accessed 2024. https://www.storj.io/storjv3.pdf.