Squid Storage Report

Muhamad H. Salman
muhamadhuseyn.salman@studio.unibo.it
Grabiele Basigli
grabriele.basigli@studio.unibo.it

May 22, 2025

Abstract

Squid Storage is a distributed storage designed to provide file management, replication, and synchronization across multiple nodes. The system employs a client-server architecture, where clients interact with a central server to perform operations such as file creation, updates, deletion, and locking. The server coordinates with data nodes to ensure redundancy and fault tolerance through replication mechanisms. It leverages a custom text-based communication protocol, to handle message parsing, file transfers, and request dispatching. The project offers a GUI to interact with the local filesystem.

1 Project Goals

The goal of the project is to provide a file storage that is reliable and strongly consistent, therefore avoiding write conflicts and update loss. The main goal is consistency, despite availability when necessary.

1.1 Usage Scenario

The primary use case is a collaborative, multi-client environment with a read-heavy access pattern. In this scenario, multiple clients must remain synchronized on a shared set of folders and files, with changes becoming immediately visible to all participants. The system performs best when concurrent writes to the same file are rare and clients are primarily reading or editing different files in parallel.

Users are expected to modify files exclusively through the provided GUI, which functions as a text editor and captures events that need to be propagated to other nodes. Editing files with external editors may lead to missed updates or delays in propagation. The images below show three use case scenarios.

Figure 1: Update file scenario

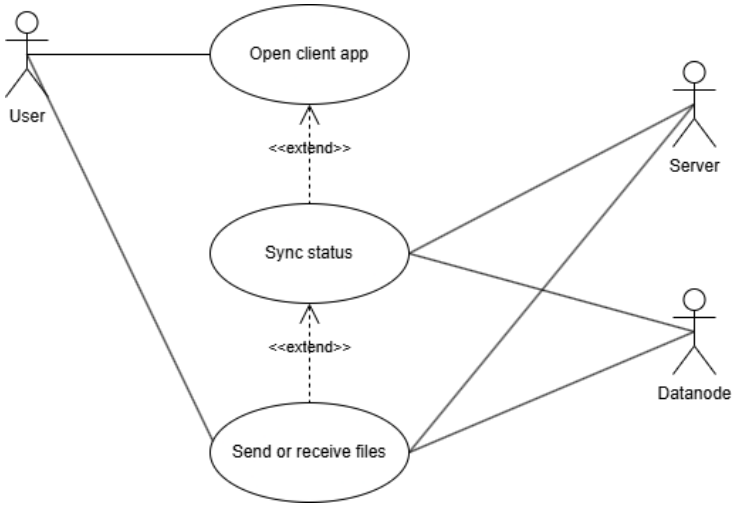

Figure 2: SyncStatus scenario

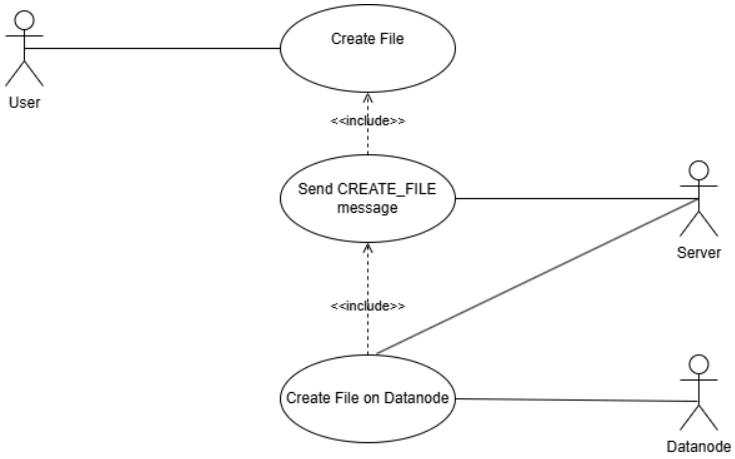

Figure 3: Create file scenario

1.2 Definition of Done

Artifacts delivered:

- Server executable
- Client executable
- Data node executable

The system is developed in C++ and it is built using CMake. The functionalities can be tested locally with a provided bash script that runs multiple nodes on different terminals and folders using the Tmux terminal emulator.

2 Background and link to the theory

Squid Storage draws upon several well-established architectural styles and interaction patterns from distributed systems theory. At its core, the system follows a client-server architecture, a centralized model in which a single server coordinates access to shared resources and maintains global state. This design was chosen for its simplicity and for its ability to enforce a global order of operations, which is critical in maintaining strong consistency across multiple nodes. The decision to centralize coordination reflects principles found in primary-replica replication patterns, where a central authority (the server) manages updates and synchronizes replicas (datanodes). This approach simplifies consistency but introduces a trade-off in availability and fault tolerance, particularly due to the server's role as a single point of failure.

From an interaction standpoint, Squid Storage utilizes both synchronous and asynchronous messaging. Communication between clients and the server is synchronous during request/response interactions, such as file operations and lock acquisition. However, the server asynchronously propagates updates to other nodes, a pattern aligned with event-driven architectures and publish-subscribe models, albeit implemented manually using raw sockets.

The system also incorporates heartbeat mechanisms, a well-known technique in failure detection, allowing the server to monitor datanode liveness. Similarly, the distributed locking mechanism is inspired by classical mutual exclusion algorithms, ensuring serialized access to shared files and preventing conflicting updates.

Squid Storage deliberately avoids using modern middleware, frameworks, or external consensus protocols (e.g., Raft) in order to explore the complexity of distributed coordination, failure handling, and state consistency. This hands-on approach enabled a deeper understanding of the design challenges associated with CAP theorem trade-offs, particularly the decision to favor consistency over availability during network partitions or client disconnections.

3 Requirements Analysis

3.1 Requirements list

Functional requirements:

- Clients should be able to read, write, create, and delete files via a desktop application. The application should allow the user to navigate files inside a folder, and thus, by clicking on a file, provide the ability to modify it with an internal file editor. Furthermore, the internal file editor should contain a "save" button displayed only when a connection with the server is present and the opened file isn't open by another client, and a "refresh" button shown when an updated file version exists.

- When a connection with the server is established, the client should be able to receive updates from others by signaling its status to the server. Afterwards, the server should deliver updates based on the client's status.

- Operations made on a file by a client should be propagated to others by sending a message to the server. Thereafter, the server updates other clients accordingly.

- The server should not keep files in its file system. Instead, it should store files in separate nodes called data nodes. A data node should not contain a copy of every file, but rather a subset of them.

- The creation of a file should be balanced evenly on data nodes based on a specific "replication factor".

- The server should propagate operations only to data nodes that holds a copy of the involved file.

- Each data node should passively perform operations requested by the server. These operations consist of creating, updating, and deleting files.

- The server should provide a distributed file locking system to avoid concurrent writes on the same file. When a client wants to modify a file, it should obtain its lock from the server. If the lock is locked, the client should not modify the file. Once the client closes the file, the lock should be released.

- If the server loses connection with a client that previously obtained a lock, it should release the lock. Alternatively, if the client is still connected but the lock expires, the server should release it.

Non-functional requirements:

- Client and data nodes synchronization should not interfere with user interaction.
- Clients and data nodes should try to reconnect when the connection with the server is lost.
- The server should recognize failures and crashes of other nodes and close the connection.
- The server address, port, and other configuration parameters should be configurable via command-line arguments.

3.2 Top-Down Analysis

A client-server architecture was chosen as the most suitable approach, given the need for a centralized orchestrator to enforce a global order of operations and prevent state divergence across the system. Additionally, the requirement to store redundant backup copies of files on remote nodes further supports the use of a central server, as it simplifies coordination compared to a decentralized design.

The system relies solely on socket-based communication, meaning that all point-to-point interactions are synchronous. However, when the server receives an update from a client, it immediately responds to release the client and then asynchronously propagates the update to the other nodes.

While this interaction model is not the most scalable -an entirely asynchronous messaging approach would have been more appropriate, we wanted to build a system from scratch without any framework, in order to experience the nitty-gritty details of distributed communication.

4 Design

4.1 Structure and Domain Entities

The main entities correspond to the three roles:

1. Client
2. Server
3. Data node

Some behavior is common to different roles, it is grouper in the Peer concept, which is then specialized by each role.
Further entities are related to file management:

- FileLock: represent the write access for a file, prevents other agents to write the same file while it's locked, but allow read access
- FileManager: manages file creation and deletion on the local filesystem
- FileTrasfer: encapsulates the logic of transfering a file over a network

The following entities are related to communication and protocols:

- Squid Protocol: represent the custom protocol designed to comunicate operation inside the system
- Squid Formatter: formats message following the format specified by the squid protocol
- Message: represent a message of the squid protocol, with specific fields and valeus

Figure 4 shows a UML diagram representing the relationships between these classes.

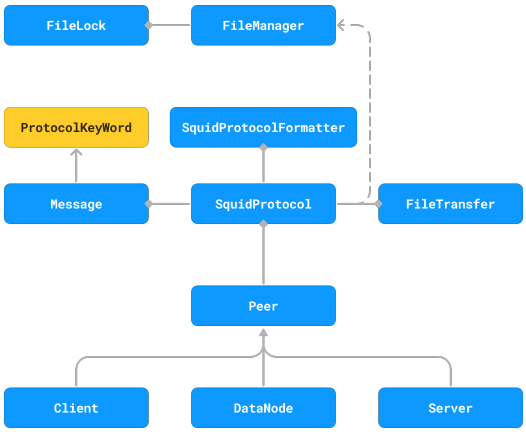

Figure 4: UML class diagram

4.2 Interaction

There is a distinction between connections from clients and those from data nodes. For each new connection, the server sends an identification message to differentiate. Then, the server initiates a handshake, which differs depending on whether the peer is a client or a data node.
When a data node connects, the server requests a list of all its files present and their versions, it stores this information to track what files are on which data nodes, This information is stored to keep track of which files are available on which data nodes. It is later used to dispatch client requests to the node holding the requested file. If the data node holds an outdated version of a file compared to the server, it is updated.
In contrast, when a client connects, the server requests that it open a secondary communication channel. The client responds by specifying the port to connect to. The second channel is needed to avoid message conflicts, since both parties have an active role. After establishing the second channel, the client requests to synchronize its state with the latest file updates. The server retrieves these updates from the relevant data nodes and sends them to the client.

Then, the communication proceeds following client operations and corresponding messages. Each message is acked by the counter part.

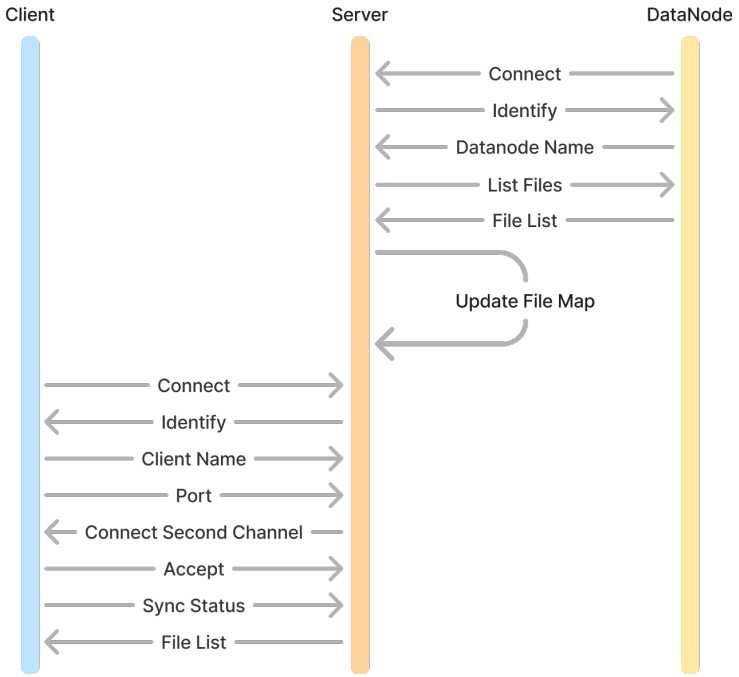

Figure 5: Initial identification and handshake

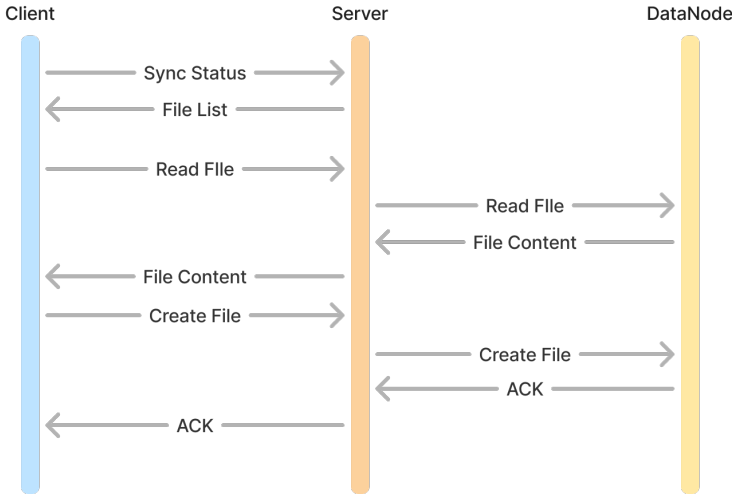

Figure 6: Normal operation sequence

Clients, before modifying a file, needs to acquire the corresponding lock on the server, and then release it when the file is closed by the user.

The file lock has a built-in timeout, therefore the client need to re-acquire the lock when it expires. This way, if a client crashes while holding a lock, the lock will eventually be released.

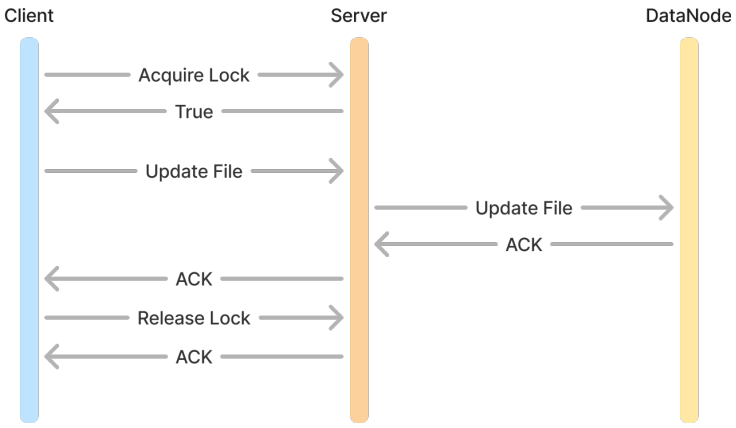

Figure 7: File locking sequence

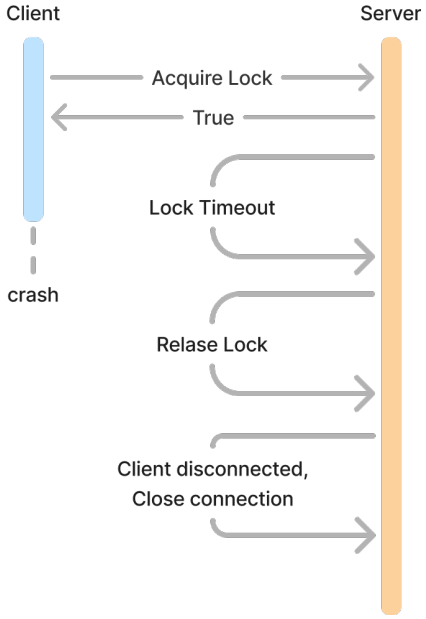

Figure 8: File lock timeout

The server sends periodically messages to data node to check the health of the connection. If a data node is offline, the server removes it from the list of nodes on which dispatch client requests and updates.

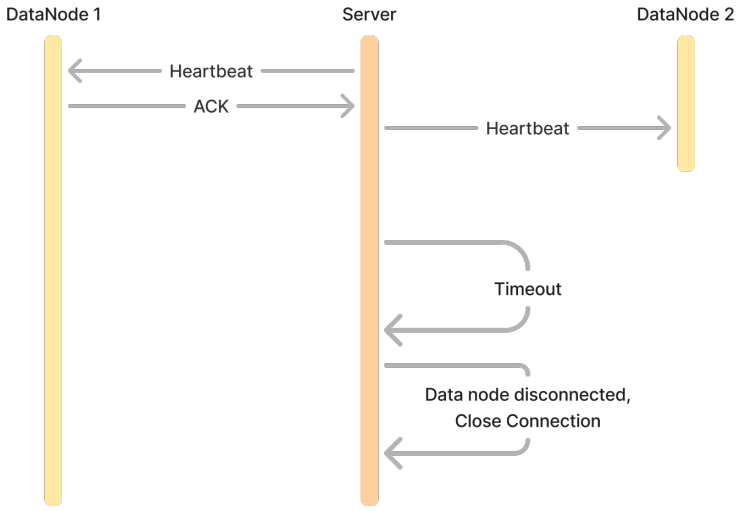

Figure 9: Heartbeat towards data nodes

4.3 Behaviour

Each entity in the system can assume an active and/or passive role in communication. An active role refers to the ability to initiate a message exchange, while a passive role indicates that the entity cannot take the initiative and simply responds to incoming messages.

- Server: it plays an active role when propagating updates, as it initiates communication with clients. Simultaneously, it assumes a passive role when handling client requests, such as serving files or receiving updates.
- Client: similar to the server, the client can act in both roles. It is active when sending updates to the server and passive when receiving updates from other clients via the server.
- Data node: since it is just a storage node without any logic, the data node is purely passive; it waits for instructions from the server and responds accordingly without initiating any communication.

4.4 Architecture

The deployment of Squid Storage involves three main types of nodes: Server, DataNodes, and Clients. Each node runs its respective artifact. With them are also shipped all the utility classes needed to interact with files and communicating following the protocol, namely classes like FileLock, FileManager, FileTransfer, SquidProtocol and SquidProtocolFormatter.

FileLock and FileManger are strictly bonded classes. While the FileLock class manages the lock on a file, the FileManager class manages file creation, update, delete, and the proper use of the FileLock. Specifically, the fileLockMap, used to keep track of multiple file lock states, is used only by the server, while the client uses only one FileLock instance. Another crucial role of the FileManager class is keeping track of every file version with a fileVersionMap stored in a file.

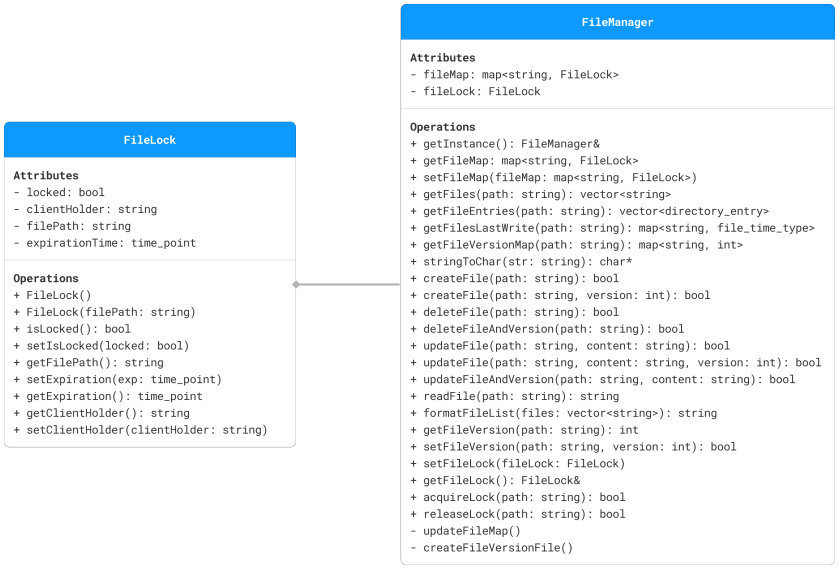

Figure 10: FileLock and Filemanager classes

The SquidProtocol class is the communication core of the project. It allows nodes to communicate by sending, receiving, and parsing messages. An internal dispatcher enables the execution of specific actions upon the reception of certain messages.

SquidProtocol

Attributes

```
# socket_fd: int
# buffer: char[BUFFER_SIZE] = {0}
# alive: bool
# processName: string
# nodeType: string
# fileTransfer: FileTransfer
# formatter: SquidProtocolFormatter
```

Operations

```
+ SquidProtocol()
+ ~SquidProtocol()
+ SquidProtocol(socket_fd: int, nodeType: string, processName: string)
+ isAlive(): bool
+ setIsAlive(isAlive: bool)
+ getSocket(): int
+ setSocket(socket_fd: int)
+ getProcessName(): string
+ getNodeType(): string
+ toString(): string
+ receiveAndParseMessage(): Message
+ receiveMessageWithLength(): string
+ handleErrors(bytesRead: ssize_t): bool
+ requestDispatcher(request: Message)
+ responseDispatcher(response: Message)
+ identify(): Message
+ connectServer(): Message
+ closeConn(): Message
+ listFiles(): Message
+ createFile(filePath: string): Message
+ createFile(filePath: string, version: int): Message
+ readFile(filePath: string): Message
+ updateFile(filePath: string): Message
+ updateFile(filePath: string, version: int): Message
+ deleteFile(filePath: string): Message
+ acquireLock(filePath: string): Message
+ releaseLock(filePath: string): Message
+ heartbeat(): Message
+ syncStatus(): Message
+ sendMessage(message: string)
+ transferFile(filePath: string, response: Message)
+ sendMessageWithLength(message: string)
+ response(lock: bool)
+ response(port: int)
+ response(ack: string)
+ response(nodeType: string, processName: string)
+ response(fileVersionMap: map<string, int>)
+ response(filesLastWrite: map<string, fs::file_time_type>)  // deprecated
+ response(fileTimeMap: map<string, long long>)
```

Figure 11: SquidProtocol class

The SquidProtocol class makes use of the Message class, the SquidProtocolFormatter class and the FileTransfer class. The first one defines what a message looks like (keyword and arguments). The second one converts a string into a Message instance and formats a string according to the protocol, allowing it to be sent through sockets. Keywords are defined as enum type, allowing for easy protocol changes. The third and last one allows endpoints to send and receive files.

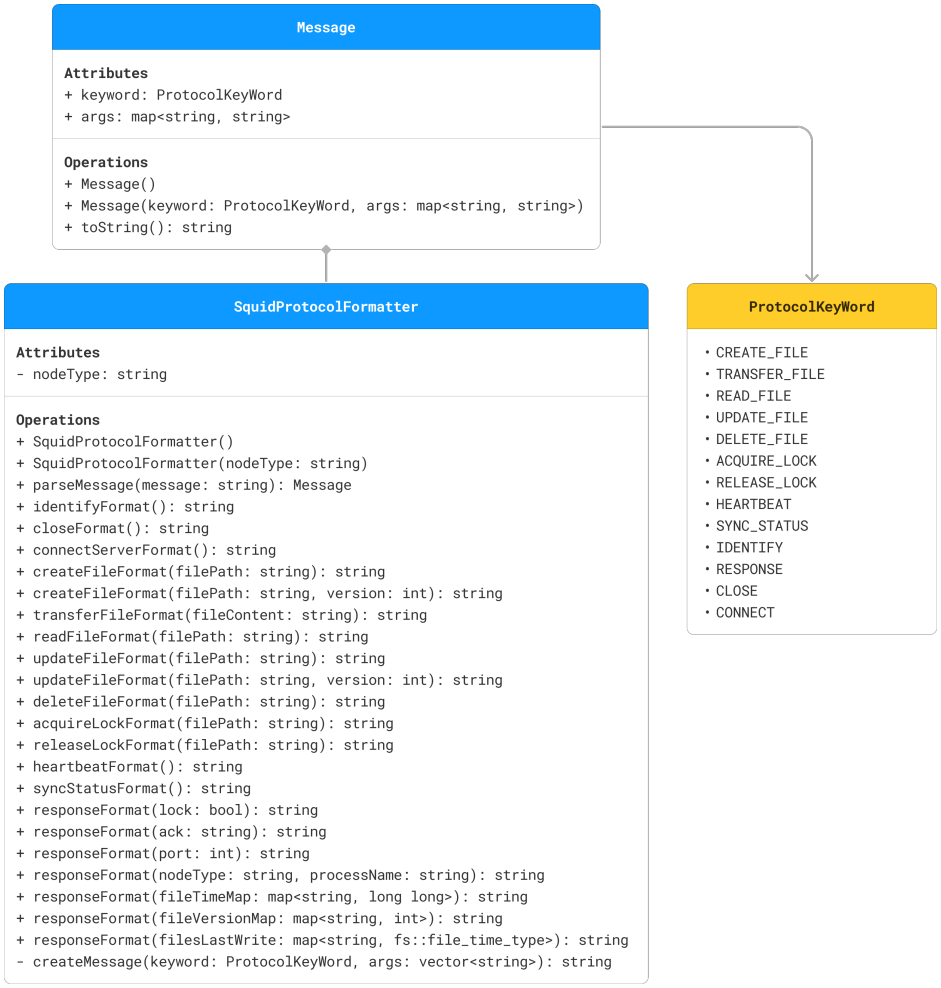

Figure 12: Formatter and Message classes

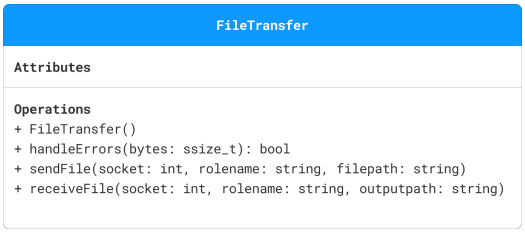

Figure 13: FileTransfer class

The Peer class represents a general node, and it uses the SquidProtocol class. A node must have some common characteristics, such as "run" and "reconnect" methods. Client, DataNode, and Server classes inherit from the Peer class. Every subclass adds its unique characteristics.

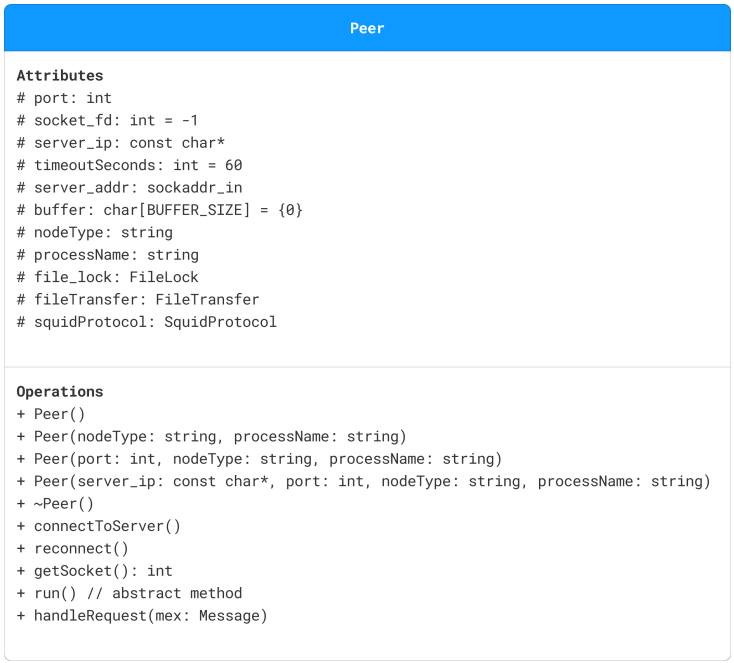

Figure 14: Peer class

Server

Attributes
- port: int
- opt: int
- buffer: char[BUFFER_SIZE]
- replicationFactor: int
- server_fd: int
- new_socket: int
- address: sockaddr_in
- peer_addr: sockaddr_in
- addrlen: socklen_t
- filename: string
- fileTransfer: FileTransfer
- mapMutex: mutex
- fileLockMap: map<string, FileLock>
- fileTimeMap: map<string, long long>
- dataNodeEndpointMap: map<string, SquidProtocol>
- clientEndpointMap: map<string, pair<SquidProtocol, SquidProtocol>>
- primarySocketMap: map<int, SquidProtocol>
- dataNodeReplicationMap: map<string, map<string, SquidProtocol>>
- endpointIterator: map<string, SquidProtocol>::iterator

Operations
+ Server()
+ Server(port: int)
+ Server(port: int, replicationFactor: int)
+ Server(port: int, replicationFactor: int, timeoutSeconds: int)
+ ~Server()
+ run()
+ buildFileLockMap()
+ releaseLock(path: string): bool
+ acquireLock(path: string): bool
+ handleConnection(clientProtocol: SquidProtocol)
+ handleAccept(new_socket: int, peer_addr: sockaddr_in)
+ sendHeartbeats()
+ checkFileLockExpiration()
+ eraseFromReplicationMap(datanodeName: string)
+ checkCloseConnetions(master_set: fd_set&, max_sd: int)
+ rebalanceFileReplication(filePath: string, fileHoldersMap: map<string, SquidProtocol>)
+ getFileFromDataNode(filePath: string, clientProtocol: SquidProtocol)
+ propagateCreateFile(filePath: string, clientProtocol: SquidProtocol)
+ propagateCreateFile(filePath: string, version: int, clientProtocol: SquidProtocol)
+ propagateUpdateFile(filePath: string, clientProtocol: SquidProtocol)
+ propagateUpdateFile(filePath: string, version: int, clientProtocol: SquidProtocol)
+ propagateDeleteFile(filePath: string, clientProtocol: SquidProtocol)
- getFileVersionMap(): map<string, int>
- printMap(map: map<string, long long>, name: string)
- printMap(map: map<string, SquidProtocol>, name: string)
- printMap(map: map<string, FileLock>, name: string)
- printMap(map: map<string, map<string, SquidProtocol>>, name: string)
- printMap(map: map<string, pair<SquidProtocol, SquidProtocol>>, name: string)

Figure 15: Server class

Figure 16: Client class

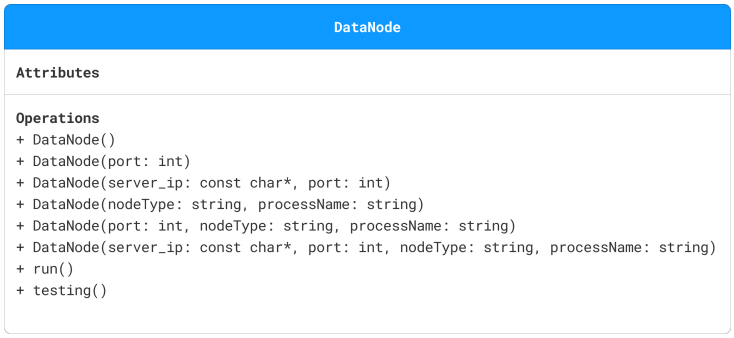

Figure 17: DataNode class

The server API reflects the set of possible message defined by the Squid-Protocol. The same messages are also used by clients and data nodes.

Table 1: Protocol Messages

Keyword	**Arguments**	**Response**
CreateFile	filePath, fileVersion	ACK
TransferFile	filePath	ACK
ReadFile	filePath	ACK
UpdateFile	filePath, fileVersion	ACK
DeleteFile	filePath	ACK
AcquireLock	filePath	isLocked
ReleaseLock	filePath,	ACK
Heartbeat	-	ACK
SyncStatus	-	fileVersions
Identify	-	nodeType, processName
Close	-	ACK

The server is deployed as a single instance to act as the central coordinator. Multiple datanodes are deployed to store file replicas, they connect to the server and stay connected until the system is running. Multiple clients can freely connect and disconnect from the server and perform file operations.

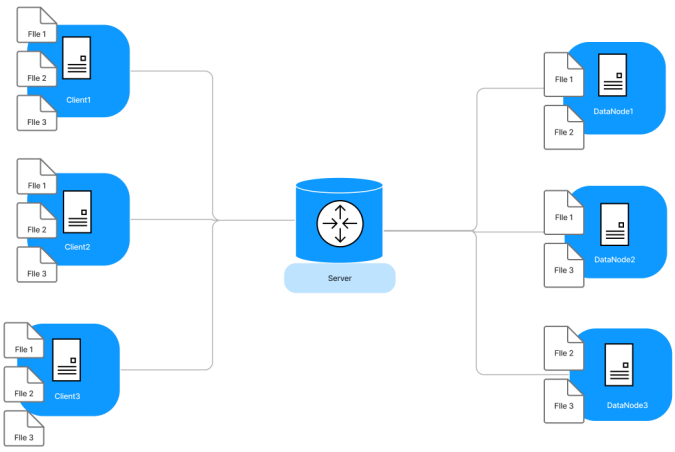

Figure 18: Deployment example

Data flows through the server, which routes it to all active clients and to the data nodes holding the replica of the modified file.

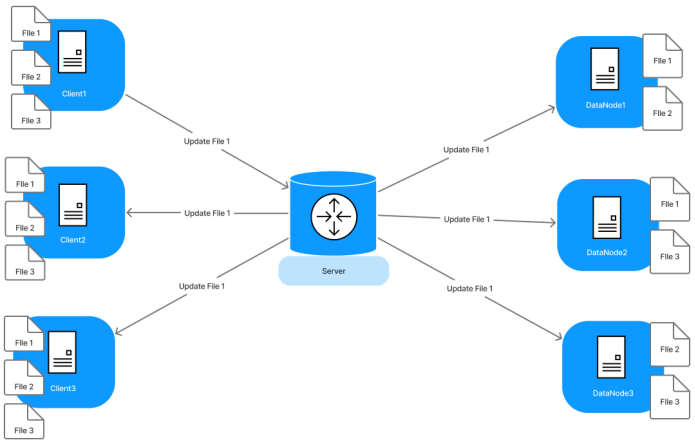

Figure 19: Data flow example

4.5 Corner Cases

Graceful Close Communication channels are closed gracefully using the `close` message of the Squid Protocol. This message is sent by the client when the GUI is closed. Server and data nodes are meant to be deamons that do not terminates. This way, the remote file back-up is always available to clients. If the connection is closed abruptly, nodes are designed to be robust and do not crash.

Timeouts and Re-connection Socket operations are configured with timeouts on all nodes. If a timeout occurs, the node assumes the connection has failed and abort the current operation. Instead, when a socket has been abruptly by the other endpoint, the node detects it when it tries to send the next message. In that case the connection is closed and re-opened. Client and Data nodes tries in loop to reconnect to the server. Moreover, the GUI goes in read-only mode to prevent file modification. After re-connection, the client and the data retrieves the latest updates they have missed.

Faults Detection Faults are detected using mechanisms such as heartbeats and timeouts. The server periodically sends heartbeats to all connected data nodes. If a data node fails to respond with an acknowledgment, the server marks the data node as unavailable and removes it from its data structures.

Fault Management The server uses the select system call to monitor multiple sockets. If a socket becomes invalid, the server detects this and removes the corresponding file descriptor. Moreover, the server tracks file locks held by clients. If a client disconnects, the server releases any locks held by that client to ensure other clients can acquire them.

Recovery Strategy If a data node becomes unavailable, the server reassign the affected files to other healthy data nodes, in order to meet a specified redundancy threshold. Finally, if the connection is lost while transferring a file, the file is delete and the operation aborted, to avoid keeping files in an inconsistent state.

5 Salient Implementation Details

The systems relies only on sockets. For ease of development the server is single-threaded and uses the `select` system call to await on all client sockets at the same time. Requests are therefore processed serially.
Socket communication is encapsulated inside a SquidProtocol object which rules the communication. The server holds a SquidProtocol protocol object for each client and data node connected.
Towards client, the server holds two socket at the same time. These is necessary because both parties have an active behaviour, so to avoid conflicts and message overlapping the communication is divided on two logically different channels. The primary socket is used bidirectionally and only the client can start communication on it. On the other hand, the secondary socket is used bidirectionally and only the server can take initiate a message exchange. So doing, the server can use the secondary sockets to propagate updates received form a client, and other clients can still send their updates meanwhile.
To do so, the server has different data structures to hold and manage sockets towards multiple nodes.

1. `primarySocketMap`: used by the select system call, maps the file descriptor to the corrispondent SquidProtocol object.

2. `clientEndpointMap`: maps a client unique identifier to its two SqudiProtocol objects (representing the two sockets).

3. `dataNodeEndpointMap`: maps a data node unique identifier to the corrispondent SquidProtocol.

4. `dataNodeReplicationMap`: maps each file to the list of data nodes that holds a copy of that file.

5. `endpointInterator`: for each file, store the iterator to the list of data nodes that hold a copy, in order to load balance reads request on these nodes using round robin.

6. `fileLockMap`: tracks the lock status for each file.

7. `fileVersionMap`: tracks the version of each file, which is an unsigned interger that is incremented at each modification.

When new connection are accepted or old connection are closed, the correct maps are update. The same stand for file creation or deletion and their maps. When a data node disconnects, the file is replicated on a new data node.

Finally, below is shown the format that each message of the Squid Protocol follows:

```
Keyword<ArgName1:Arg1, ArgName2:Arg2, ..., ArgNamen:Argn>
```

The SquidProtocolFormatter has the responsibility to serialize Message objects into this test format and deserialize text messages into Message objects.

Regarding the management of files and their version, originally we thought of using the file's last write timestamp. However, maintaining a distributed system that relies on time is not a good idea. So we embraced an "happened before" mechanism in order to model the system as a sequence of executed actions. Whenever a client successfully modifies a file, its version is incremented by one. Other clients looking for file updates look at their respective versions; old files will have a version number lower than the new one. The FileManager class manages this file versioning using a persistent map inside a file called ".fileVersion.txt".

6 Validation

Choosing the CAP theorem and fault-tolerance as evaluation criteria, because connection loss is detected and handled properly (re-connection in client and data nodes, endpoints map update in the server), we can state that the system is tolerant to network partition in the most important cases. Consistency is prioritized over availability because when a client is disconnected, it is not able to write files, so the goal of being consistent in all main failure scenarios is met.
The system may not be tolerant to all possible failure scenarios, there might be some operation that, if interrupted, brings the system into an inconsistent

state. Moreover, the server has only visibility of files on data nodes when those data nodes are online. If all the data nodes holding a specific file are down, the server doesn't have any knowledge of that file and its version, so it might propagate an older version stored by a client.
Some scripts are available as manual tests to simulate different deployment scenarios. The necessary libraries are often pre-installed in Unix-system, if not, check the Deployment 7 section to install the necessary dependencies. Also the `tmux` terminal emulator is needed.
The testing scenarios are the following:

- `testMultiClient.sh`: executes multiple clients, one server and one data node.
- `testMultiDataNode.sh`: executes multiple data nodes, one server and one client.
- `testMulti.sh`: executes multiple data nodes, one server and multiple clients.

To run these tests, go in the `test` folder of the project, and execute the corrispondent test:

```
./testMultiDataNode.sh
```

Listing 1: Run a test

If the artifacts are missing, there are equivalent scripts to compile and build the project again. Inside the `build` folder, execute:

```
./buildTestMultiDataNode.sh
```

Listing 2: Build and run

7 Deployment Instructions

The artifacts produced can run on Linux and MacOs. Each of them can be deployed on different nodes, or on the same node for testing purposes. The only library needed are the graphical ones on the client node:

On Linux:

```
sudo apt update
sudo apt install libsdl2-dev
sudo apt install libgl1-mesa-dev
```

Listing 3: Install run-time libraries

On Mac:

```
brew update
brew install sdl2 sdl2_image sdl2_mixer sdl2_ttf
brew install glew glfw
```

Listing 4: Install run-time libraries

Then each artifact can be run on different nodes:

```
./ClientNode
./DataNode
./ServerNode
```

Listing 5: Start each component

8 Usage Examples

The deployment of Squid Storage involves three main types of nodes: Server, DataNodes, and Clients. The server should be deployed on a single node, instead the Client and DataNodes can be deployed on multiple nodes and all of them connects to the same server. These represent the case of multiple clients working on the same set of files and folders. The server keeps all their file synchronized and storing remote back-ups on datanodes.

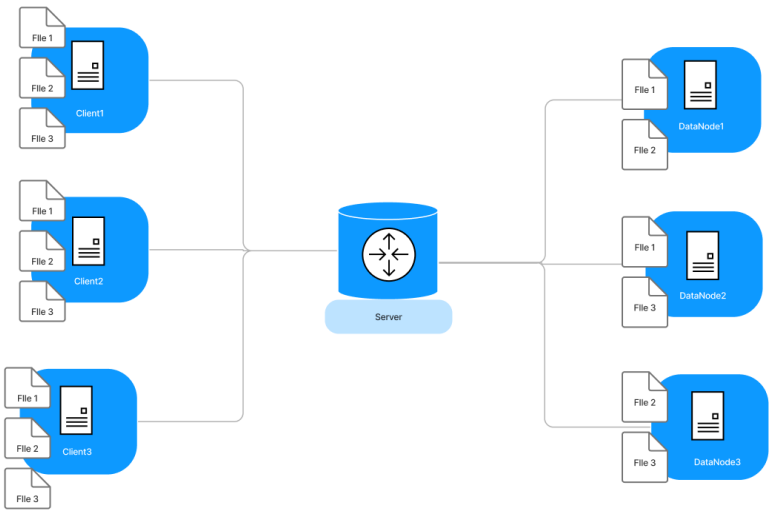

Figure 20: Deployment example

9 Conclusions

The Squid Storage project successfully implemented a distributed storage system with robust replication, fault tolerance, and consistency mechanisms. The system is well-suited for scenarios where data consistency is critical, such as collaborative file editing or distributed backups. While trade-offs were made to simplify the design and prioritize consistency, the system provides a solid foundation for further enhancements.

9.1 Future Works

The systems right now is missing server replication, improvements in the Squid Protocol, server metadata durability, automated testing and continuos integration. The server is a single point of failure, but we didn't have time to add replication. Furthermore, the Squid Protocol is missing support for `Request to Send` and `Clear to Send` messages. Introducing these message types would enable the use of a single socket per client, replacing the current approach in which the server maintains two separate sockets for each client.

9.2 What did we learned

Throughout the development of Squid Storage, we gained valuable hands-on experience with the fundamental challenges of building distributed systems from the ground up. By avoiding the use of existing frameworks and libraries, we were able to directly confront and understand low-level concerns. We learned how difficult it is to maintain consistency in a distributed environment, when nodes may join, leave, or fail at any time. Implementing mechanisms like distributed locks, heartbeat monitoring, and replication rebalancing taught us how to design resilient systems that can tolerate failures. Designing the Squid Protocol helped us appreciate the importance of clear, extensible communication standards and the trade-offs involved in protocol design. We also encountered the complexity of coordinating active/passive communication roles, which pushed us to think carefully about connection management and concurrency. Moreover, we experienced firsthand the scalability limitations of synchronous, blocking I/O and the benefits that could be gained from adopting asynchronous communication models.