
COLLIGO: A FRAMEWORK FOR DECENTRALIZED SOFTWARE DEVELOPMENT

ALIN-DANIEL PANAINTE, 201802942

MASTER'S THESIS

June 2020

Advisor: Niels Olof Bouvin

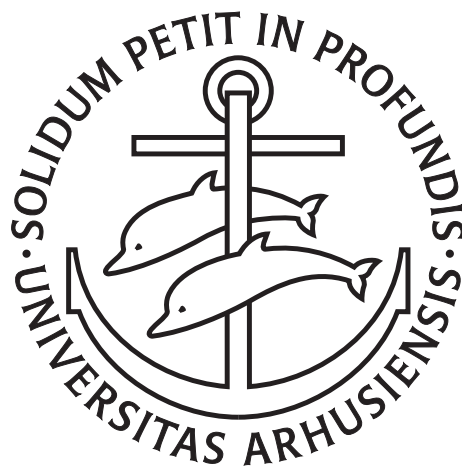


AARHUS
UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

COLLIGO: A FRAMEWORK FOR DECENTRALIZED SOFTWARE DEVELOPMENT

ALIN-DANIEL PANAINTE



Master's Thesis
Department of Computer Science
Science & Technology
Aarhus University

June 2020

CONTENTS

1	INTRODUCTION	1
1.1	Use cases	1
2	RELATED WORK	5
2.1	Similar projects	5
2.2	P2P Systems	5
2.2.1	IPFS	6
2.2.2	Dat Protocol	7
2.2.3	Secure Scuttlebutt	8
2.3	Authentication	11
2.4	Storage	11
2.4.1	OrbitDB	12
3	ANALYSIS	13
3.1	What do developers need from a peer-to-peer system?	13
4	DESIGN OF ARCHITECTURE AND EVALUATION	17
4.1	Initialization	18
4.2	General usage	21
5	IMPLEMENTATION	23
5.1	P2P-system = IPFS	24
5.2	Project creation	26
5.3	Dashboard	26
5.4	File synchronization	27
5.5	Extension modules implementations	28
5.5.1	Git	29
5.5.2	Chat	30
5.5.3	Git-bug - project management	30
5.5.4	Future extensions	31
6	EVALUATION	33
6.1	Feasibility	33
6.1.1	For users	33
6.1.2	For developers	40
6.2	Pre-conditions of a collaboration process	46
6.3	Challenges	46
6.4	Limitations	47
6.4.1	For users	47
6.4.2	For developers	47
6.5	Discussion	48
7	CONCLUSION	49
7.1	Tools and work assessment	50
7.2	User testing	51
7.3	Future work	51
7.4	Closing thoughts	51

LIST OF FIGURES

Figure 1	IPFS stack	7
Figure 2	SSB stack	9
Figure 3	Peer structure of Colligo	17
Figure 4	Colligo's work-flow	18
Figure 5	Set up steps for each peer	19
Figure 6	Extension module abstract structure	21
Figure 7	First run of the application for the user	23
Figure 8	Second and future run workflow	23
Figure 9	How the file transport works	24
Figure 10	View of a project's dashboard	27
Figure 11	Registration page	34
Figure 12	Project setup page with project creation interface	35
Figure 13	Project setup page with join project interface	36
Figure 14	IPFS performance test	38

LISTINGS

Listing 1	Configuration fields for Colligo.	25
Listing 2	Configuration fields for a project.	25
Listing 3	Database entry for main work collaboration.	27
Listing 4	Webserver part of the Git extension.	40
Listing 5	Communication with the framework section of the Git extension.	42
Listing 6	Adding folders recursively to IPFS.	46

INTRODUCTION

Arguably, the most popular software products have a centralized architecture. Popular applications such as Facebook¹, Google² and Dropbox³ have numerous clusters of servers that contains all the logic for the server. The client entities make requests to the servers and receive a response. This can be a favourable solution for maintenance, efficiency and control. It is easier to update and fix errors only for these servers as the entire logic is present there.

Having all the information in one place makes it fast to store and look up the information. Having absolute control over software's storage and logic it is easy to administrate and make sure that everyone follows the same rules.

This is also the case for software development products. Github is one of the biggest platforms for open-source application hosting. It offers version control, source code hosting, and various functionality to aid the software development process. The most vital component of Github, Git, is designed to have a decentralized architecture. Being used with Github the distribution property does not exist anymore. If Github goes down, developers will not be able to submit and synchronize their code with the main location of the repository. While Github and other mainstream services offer a large variety of tools to improve the productivity and collaboration between teams we always need to have alternatives. A different perspective can give a new view of the existing solution and perhaps offer a better answer that can also address self-reliability and not depending on a single entity for work management.

An alternative can be represented in a decentralized manner. With the same purpose of collaboration and file synchronization it is interesting to explore if it is possible to achieve the same experience as a centralized application.

1.1 USE CASES

To better understand the usage of a decentralized collaboration tool we can define a few examples of use cases. Virtual teams have an increased adoption not just in software solutions companies but in all other fields. The following examples will attempt to address different communication mediums. These mediums are outlined in the media

¹ <https://www.facebook.com>

² <https://www.google.com/>

³ <https://www.dropbox.com/>

richness theory [1], a framework to describe how different ways of communication can reproduce the information sent over it. For each medium, we offer a possible solution present in Colligo.

A team can use this framework to create, manage and protect it to others and every person their intellectual property work on a project. Having just a local connection in the office between the terminals of the users can communicate, synchronize, and merge their work. For remote work, users need to have a direct connection with the rest of the team without the necessity of a central server somewhere to provide the service. This guarantees that the data is processed and passed only between their computers. The project manager can choose to unify the approved teamwork and distribute it to others and every person can choose what work to merge from others. In the same sense as the work files, we can distribute messages to assure communication in the team. The file distribution makes sure the team won't lose their work if one terminal or server has a system failure as it is easy to just retrieve back the files from the others.

For this case we can outline some critical functionality: first, we have content distribution, as each node/person can distribute its work to others and retrieve work from others. Second, a version control system is needed to keep track of changes, the author of them and the chronological order that also allows users to revert to past progress. These two functions are enough for work content management. We can also add communication as a channel of discussion within the team. Being able to have a communication channel tied to the file distribution channel helps to easily refer specific work parts into the discussion without using other tools to complicate the workflow. The communication can replace emails and substitute dedicated video call software. All the teamwork-flow can be integrated into a decentralized work collaboration framework.

Homework projects for students in a class can present another possible use case. In a class, a project can engage each student to present their work in a given exercise. All solutions can be shared with the teacher. The students can also evaluate the work of other students from the distribution functionality and come up with a complete ideal solution. They can also work on different parts of a project and plug them together. The teacher can communicate on the work and for example, students can only pull work from the teacher's node while every student contributes with their work. This scenario is not limited to just code projects as it can be extended to other types of documents, such as the development of legal documents, molecule structures construction or a design for a building.

This thesis presents Colligo, a framework for development collaboration using a decentralized architecture. It's an exercise to see how we can integrate different peer-to-peer (P2P) tools to work together to have a complete solution for software development. How difficult is it

to integrate current tools in a standardized manner? How feasible is it to have a decentralized application for software development? Different tools are analysed and used with other tools to construct a framework that would serve as a decentralized solution while also making it easy to integrate other existing tools. An experimental analysis is done to see also how well these tools scale for different use cases. Colligo is an exploration of a decentralized solution in times where all the major software solutions are centralized. Data has become one of the most important forms of currency and as such, the most used applications have all the data flowing through their servers first. One big disadvantage is also a single point of failure. These well-known services would stop working if the central servers would fail.

We will analyse existing libraries of P2P systems that offer tools that can be used to create Colligo. The characteristics of a P2P system and current demands of today's developers for collaborative work define the foundation of the framework. For each one of those characteristics, we present relevant features present in the chosen P2P systems. Separate libraries that can be plugged into the P2P system will be used to comply with all the necessities to offer a fully-feature collaborative solution for developer teams.

Cloud solutions are popular now and offer great uptime. Data centers offer great redundancy for applications in case that a hardware component fails. As more and more companies are dependent on cloud solutions, even short and rare downtime can have a great impact on the business. They are far more reliable than on-site servers, but they also introduce security questions for situations such as: storing a company's private data on servers owned by a different company, issues regarding personal and sensitive information across borders. A centralized collaboration tool on downtime has a direct negative effect on team productivity.

A decentralized solution works completely differently. Information is stored in multiple locations — nodes. Participants share their work and communicate in an efficient matter without having one entity to be the central brain of the network. To make sure that the work documents are always available, the network can have cloud-assisted peers that store the information and their sole purpose would be data redundancy and high-availability. While the network would not rely entirely on them, having a hybrid system of both a cloud and a P2P system can meet production-ready standards that can match popular centralized solutions.

An application's availability during a crisis should be considered. On an occasion where there is a sudden surge in application usage, centralized applications deal with slow down periods where they need to adapt to the current demands and increase the server's power. In a decentralized application, the load is shared by all the nodes. With this distribution, a sudden increase in the usage will not affect

the network, as more nodes would mean more computation power and the duties split to more entities.

Currently, the scarcity of popular decentralized software can be explained by the lack of simplicity for setting up some of them and security. Having a central authority is easier to govern rules in an application and manage identity. In a distributed environment, identity is dealt with differently. Most of the people are familiar already with the existing centralized solution. For decentralized applications to cross the chasm of early adopters and be used by the general public can pose a challenge.

This project takes the software collaboration idea and puts it in a different perspective and analyses how it can work in it.

We will first present the existing tools and how they work. Each tool is presented in the current state of development and how it can be integrated into a framework. Next, we will introduce Colligo, the framework that will connect all these tools and its functionality. A standard way of decentralized communication and its principles are discussed and analysed how they can work in the framework. We will then analyse different experimental cases and the usability of the current implementation. Scalability, efficiency and maintainability are evaluated in comparison with existing centralized solutions.

Finally, we go a step into the future. Ways to improve and extend Colligo are discussed. Possible limitations are outlined, and we explain how to integrate more tools without having to modify the fundamental structure of the framework. We also investigate potential future work that can evolve from the main idea discussed in the thesis.

Having set the context of the thesis and considering the evolution of decentralized tools, the thesis experiments the following hypothesis questions:

- How feasible is a decentralized collaboration framework?
- What are the requirements for a collaboration process in a decentralized manner?
- What are the challenges of decentralized collaboration?

RELATED WORK

A centralized architecture means that one or just a few entities have authority and control over the entire network. This is the most popular structure for web solutions. Limited availability, connectivity dependency and difficult maintenance are the main limitations for such a system. While a centralized system also has its advantages, in this thesis we will explore the characteristics and advantages of decentralized systems.

2.1 SIMILAR PROJECTS

Brig [2] is a distributed and secure file synchronization tool built on top of IPFS. It features CLI (command-line interface) commands like Git, a web UI and aims to be a peer to peer alternative to Dropbox. Brig creates an IPFS node as a separate process and it is communicating to the node using the CLI API. Like Colligo, it is a decentralized solution for file distribution and synchronization. It uses IPFS from its distributed system and Git for version control. Brig is different from Colligo by aiming for a solution focused on encryption and security of the data. Also, the main goal is to be like Git commands for familiarity. There is file navigation inside a Web UI similar to Dropbox for easy use. This tool concentrates on distributed file management and as such communication is not taken into consideration. Although, it has a system of identification and authentication to attribute the files for an entity.

2.2 P2P SYSTEMS

In [3] defines a distributed system as:

A collection of autonomous computing elements that appears to its users as a single coherent system.

From this definition there are two characteristics that we can name:

Autonomy. Each computing element should be able to work independently without approvals or permissions from either a central entity or other components.

Coherence. The network's nodes should act as a coherent system to offer a smooth experience similar to a centralized system where data and communication are reliable and the same for each node.

Of course, these two characteristics can be considered broad as we can go in-depth for each one to define *secondary* traits that are more specific and define better a distributed system.

For the rest of the chapter different P2P systems are presented in the prism of the two aforementioned characteristics while also highlighting more concise traits that can be used to compare and evaluate existing solutions for P2P systems.

When thinking about P2P network systems we have several implementations that we can use. Each system has a different goal that they want to carry out by using a P2P solution.

2.2.1 IPFS

The Interplanetary File System (IPFS) is a peer-to-peer distributed file system that aims to become an alternative to the classic Web by offering a distribution system of content-addressed block storage model, with content-addressed hyperlink [4]. The main design attribute of IPFS is that it gets successful proven ideas from earlier peer-to-peer systems, including DHTs, BitTorrent, Git, and SFS and integrates them in one place. Using all these technologies together makes them more accessible for use and also justifies IPFS to be a reliable solution to a distributed system.

The protocol of the system is a modular one consisting of various components that are available to use without depending on the other ones.

As the distribution is using content-addressed blocks one challenge in using this system is keeping track of all the files and making sure that other nodes have always the latest version of the files. For this, IPFS comes with two important sub-protocols: First, we have IPFS protocol for naming and mutable state. We can assign a unique hash name for a file so that every time we have a new link for an updated version of the file, we will always have the same name for it. Second, with the Files component, IPFS can easily represent a filesystem consisting of files and directories each with their links (for small files) and multiple links for each chunk of large files.

Another sub-protocol included is Bitswap. Data distribution takes place by exchanging blocks with peers using a BitTorrent inspired protocol. IPFS has elementary support for implementing a barter system where nodes can decide what nodes to favour.

An overview of all the sub-protocols included in IPFS can be seen in the [Figure 1](#).

One disadvantage of this P2P system is that it does not contain an explicit versioning control system. As such we cannot have multiple versions of a file as each modification will consist in a completely new file and link.

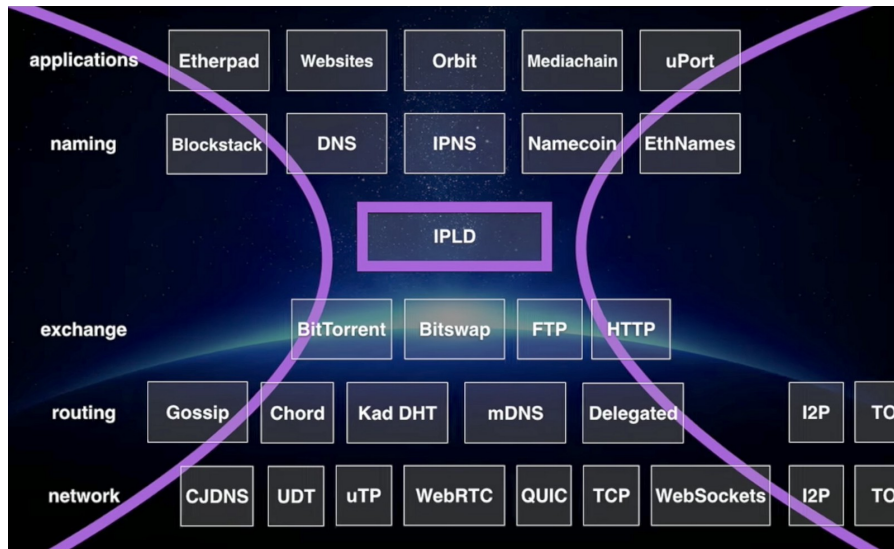


Figure 1: IPFS stack. It shows the technologies used to build the whole IPFS functionality as presented in its whitepaper. Not all of them are used by default but they represent all the tools available to use inside the stack. Picture taken from [5]

The main implementation of IPFS is in Go and JS but it also offers complete HTTP and CLI APIs to communicate and manage a node. This opens the door for implementations in any language to operate with the node. This, perhaps, is not a trait for distributed systems but it's a plus for authors, the flexibility to integrate IPFS into their applications.

Looking at projects similar to Colligo, IPFS is a popular choice for the file-sharing protocol [6] and communication between IoT devices [7].

2.2.2 Dat Protocol

Dat protocol is a distributed system aimed mainly at large dataset synchronization [8]. It emphasizes security and is implemented in an agnostic manner to the underlying transport protocol. The protocol is characterized by content and publisher integrity, network privacy and decentralized mirroring. The Dat network also works in low-connectivity contexts, this allows a large variety of possible users with different types of internet connections. Nodes that share the same files can discover each other and contribute efficiently to the swarm. Dat also offers an efficient traversal system for huge files hierarchies and a versioning system to keep track of the modifications.

As the protocol is focused on distribution and high availability of large files it lacks one important functionality for collaboration. A file can only be modified by the author. Making collaboration rather cumbersome to use.

For networking, Dat uses the SLEEP format to store and send data. It's specifically designed to allow sparse replication for downloading only parts of a repository in the case of optimizing the transfer.

Dat protocol implementation is represented by Dat SDK. Same as IPFS's implementation, it consists of several low-level submodules to work with data in distributed applications. They can all be used with a high-level API (application programming interface) provided by the SDK available for multiple platforms focused on Javascript language.

With Dat SDK we can easily manage files using Hyperdrive. Advertised as a "secure, real-time distributed file system", it represents a high level, easy to use API, that intends to offer files operation functionality very similar to node.js 's core module *fs*, with versioning and real-time replication across other computers.

2.2.3 *Secure Scuttlebutt*

Secure Scuttlebutt (SSB) is a novel peer-to-peer event-sharing protocol and architecture for social apps [9]. It was developed to be a database protocol for unforgeable append-only message feeds, but it can easily be generalized to be a P2P network fundamental to build offline-first decentralized.

Compared to IPFS's, SSB creates a people-centred Web versus the content-centred web that can be described as a data gossiping protocol. In other words, data is spread and synchronized through the friends that a user is following in a distributed network without central control.

Data is stored locally and synchronized between contacts (friends). Data is also encrypted; hence Scuttlebutt is also referred to as SSB or Secure Scuttlebutt. Each user/node has a public and private key and can connect to other nodes by exchanging their public keys.

Replication is a result of nodes exchanging log updates. With a point-to-point encrypted network, they run the gossip protocol. If a node receives a lower sequence number for a log than it sent, it sends the message that the other node is lacking. If new data becomes available later, it is automatically pushed to the connected nodes. The append-only logs can be an obvious comparison to a regular blockchain. The distinguishing feature here is that each node has its own cryptographically signed log.

SSB is split into three independent protocol layers. First, the most important protocol is the message format. Each node needs to agree on what constitutes identities, valid messages, and how to compute hashes to address messages and blobs. Second, the mechanism by which the nodes exchange data. There are different alternative ways of communication. Nodes can communicate indirectly with other nodes that have a different mechanism if intermediary nodes can communi-

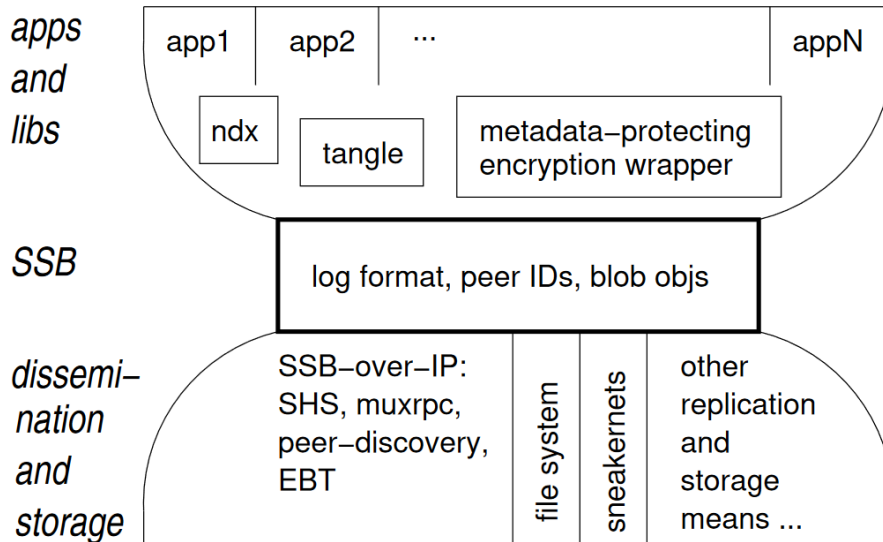


Figure 2: The SSB's stack is split in three main parts: a backend part where data is stored and transported, the central part that serves a 'dashboard' for the user and the frontend layer used to build tools on top of it. Figure taken from [9].

cate in both ways. Lastly, we have the publishing and interpretation of application data in such messages.

SSB's stack can be overviewed in Figure 2. Currently, SSB has a Javascript implementation exposing high-level functions that are accessible for developers to create apps on top of it. While it is one of the newest P2P systems compared to the other examples, it has a great emphasis on security.

Developers can choose from a large variety of tools and technologies they can use to build software products. A lot of the past phases of work are now streamlined, opening the door for more complex architectures and applications. Libraries and tools are created mostly to ease the work of a developer by offering an out-of-the-box work environment, automating processes such as compiling, testing and deploying, leaving the developer to focus only on what's important — developing.

To categorize all these existing tools, we can outline some main categories :

- **Communication.** Not just developers but every team that works on a project needs an efficient, easy-to-use communication tool. From a collaboration perspective, ideally, we want a tool that can offer persistent storage of communication logs, real-time and asynchronous communication. A simple communication system that interoperates with other existing popular already used applications. A decentralized and very extensive tool for communication is Matrix.¹ . It offers a set of open APIs for de-

¹ Official website: <https://matrix.org/>.

centralized communication which consists not just of text messages communication but also VoIP and IoT communication. Apart from this, Native SDKs for multiple platforms, are included which makes it easier to develop custom tools that use it. Matrix is open to exchanging data and messages with other platforms.

However, to run the Matrix protocol, a central server needs to be implemented where the users can create an account and use the protocol. Concerning this thesis's aim, to use this protocol does not feel decentralized. For a truly decentralized communication system, the user should not need a server to create an account and communicate with others. Anyone can indeed create a self-host a server for Matrix protocol, but if that server shuts down, the users will not be able to communicate.

- Integrated development environment. No matter what programming language is used, the developer needs to be as efficient as possible. An IDE is a full work environment that can fulfil all the developer's needs in the process of developing an application.
- Version Control. Collaborating on multiple files is important to be able to effectively merge, compare and complement your work with others. These days, version control has become a vital component not just for teams but also for individual work as a tool to effectively track progress and revert to old versions of the work. Here, Git is the most used tool as it offers a superior and complete experience for its users. Interestingly enough, Git has been developed as a decentralized solution, but it is mostly used with centralized software. In this thesis, we will present Colligo, a framework that maintains the decentralized idea of it while making it as effective as it would be in a centralized solution.
- Project management. A team needs to have an overview of the progress, current issues and milestones. For this, an issue tracking tool or project organizer tool is important. They might not be used a lot by developers but they play an important role in the development process of a software product.

A possible solution for a decentralized issue tracker is git-bug. It is presented as a distributed, offline-first bug tracker embedded in Git [10]. Additionally, it offers different UIs to easily view the information, lightweight and can easily be integrated with other applications.

2.3 AUTHENTICATION

When a decentralized application is in the design stages one subject that will arise is security. In a centralized application, different security protocols are well-known and defined as the industry's standard. The most popular choice is the OAuth protocol[11]. This is a protocol designed with a client-server architecture in mind. A central server registers as a trusted entity where the user can serve its third-party application credentials. Here, an account created on a different website is used to log into another application. Applications that use OAuth protocol are not storing any passwords instead, they store a token that helps for the authentication.

With a decentralized application, there is no central server where the user can register himself. A proposed option is to authenticate a user with a blockchain-based mechanism that uses Public-Private key pairs. DAuth [12] is an authentication system that uses Ethereum to store and verify the user's identity. It allows for an authentication system that is independent of any central actors. Authentication can happen even if the user is the only one in the application potentially hosting its DAuth service. In its architecture, the system uses a DAuth server that holds the private keys and has an endpoint to receive the user credentials. Blockchain is a very well used tool involved in decentralized systems [13] and IoT [14] security.

As the authentication subject is used in this thesis but it's not the focus point, a more naive solution is possible where an append-only log data structure can be used to store users' credentials. It can be shared with all the peers and when a new peer joins the network it can register on it and next time it connects, the peer's credentials are compared with the ones stored in the log.

As this thesis focuses on the collaboration stand, the work presented in the next chapters focuses on the characteristics related to collaborative work.

2.4 STORAGE

Every modern application needs to store some persistent information that is not wiped when the application exits. An established enterprise solution for distributed storage is Cassandra [15] a database solution developed by Facebook that has the design goal to work with big data workloads replicated across multiple nodes to ensure no single point of failure. This is a tool intended for a massive amount of data and dedicated servers hosting multiple nodes that replicate the data. On a large scale, this is a great solution for P2P storage. In the context of the thesis, Cassandra represents an over-complicated solution that serves the purpose, but it does not fit with the rest of the application.

2.4.1 *OrbitDB*

A lightweight alternative of distributed storage is a distributed storage built on top of the IPFS system. OrbitDB ² offers different types of minimal storage solutions that can be managed and stored without any complex setup. It presents different data structures such as log (append-only), feed (a mutable log where entries can be added and removed), key-value (similar to non-relational databases), docs (a document database) and counter. The application does not require a server and it's an excellent choice for decentralized small-scale applications that also work offline. To create or connect to an Orbit database, it needs an IPFS node running for transport.

² Official website: <https://orbitdb.org/>.

ANALYSIS

3.1 WHAT DO DEVELOPERS NEED FROM A PEER-TO-PEER SYSTEM?

A system that can be easily managed through a flexible API is preferred in this case. File management and synchronization with other collaborators is crucial for development.

IPFS has a CLI and an HTTP API that makes use of the full functionality. This allows us to either control it through the command line or with any application that can manage HTTP requests. While the HTTP capability can also be created with an application that wraps the CLI commands, having them natively implemented in the library guarantees a correct and well-tested flow of commands ran on the node. The IPNS modules help to keep track of each file and the MFS makes files management in a p2p context a seamless transition. Files can be treated as in a normal name-based filesystem while all the work of updating the links and hashes is taken care of.

Dat Protocol has only a CLI API and a Javascript wrapper implemented to control it. The hyperdrive offers a secure, real-time distributed file system to control files. Dat also has explicit versioning control support which can overlap Git's functionality. It's worth noting that the hyperdrive Javascript implementation is not fully implemented yet.

Scuttlebot is a log store used as a database. It does not have an explicit filesystem and as such it does not support any out-of-the-box functionality for file management. Treating the content information as a database makes it easy to replicate the information and distribute it. A very important characteristic of all tools from any category is integration. No matter how suitable a tool is at its job, it is also important how much can it be integrated with other tools is a crucial feature.

To encapsulate multiple solutions in one we need a standard way of communicating with all comments. Either we can choose a popular programming language in which the modules and the p2p system is implemented or an API that can be easily accessed and used by the modules.

Comparing the presented peer-to-peer systems we need to select the most comprehensive and flexible choice that can be easily managed from different implementations and with a standardized way of communication.

Now that we presented and clarified the related resources that can be used and similar project choices, specific steps can be discussed.

The proposed framework for this thesis can have countless extensions and possible functionality. To outline clear progress and evolution of its development we will present some functionality milestones. Each one of it has a brief description of its characteristics and the importance of it compared to other milestones.

Before thinking about different collaboration tools and communication channels the fundamental functionality is file distribution and synchronization. Having this done, we can focus on the tools that will use this, such as communication (addressed or unaddressed documents) and additional productivity tools.

This first milestone makes sure we have a basic working decentralized application where peers can exchange information over a network. This service exposes a simple interface for other components to use as at their core, all the other components should rely on decentralized communication. Here we will utilize IPFS infrastructure to make use of peers to connect, communicate and exchange data. These operations will be accessed by other components, but it can also be used just as a file distribution application without additional features. All node and framework information are accessible through a Web interface consisting of a dashboard for easy interaction with the application.

The second functionality to be implemented is version control. As IPFS does not have an explicit version control system and this feature is essential in project collaboration, we think this is a notable service to be present. For this, Git will be integrated into the framework as a submodule. The combination features Git's remote interaction with a decentralized source while not affecting the usual commands of Git applied locally on the work documents. The user should be able to see brief information about the repository on the Web interface of the framework and execute basic commands from there.

The next step is focused on communication. A submodule that uses a decentralized communication system tied close to the work content and the existing p2p system can make the framework to be seen as a complete solution for collaborative work while still being able to extend with extra features. For this, Matrix protocol will be used to implement a decentralized messaging system with end-to-end encryption.

When a message is sent in Matrix, it is replicated over all the peers that participate in the conversation or nodes that are connected to the network. This is similar to how commits are replicated in Git repositories.

One feature that can be considered as an extra service, in this case, is issue tracking. With Matrix, we have the communication functionality but as it's just a decentralized conversation store, it does not provide organizational support for issue tracking similar to what GitHub provides. For this, we use Git-bug [10]. Without polluting the project

files, we can create offline-first bugs and visualize them either in the dedicated Web UI or just integrate them in the framework's web interface.

From here the main priority is to simplify the process of creating new extensions and couple them in the framework. These extensions can make use of the distributed network and access information provided by either the core of the framework or other already implemented components.

For the use cases presented in [Chapter 1](#) these milestones should fulfil all the functionality and features needed to make the framework a noteworthy tool.

Milestones summary:

- Basic decentralized file distribution application with web interface.
- Extension component for version control using Git.
- Extension component for messaging within the framework.
- A web interface to track issues, comments and commits in a project.

DESIGN OF ARCHITECTURE AND EVALUATION

To answer this thesis's questions and showcase the usability and simplicity of Colligo, in this chapter we will present its general architecture and work-flow. A fast and simple set up is imperative for users' satisfaction so that they can focus on their actual program without getting distracted by the collaboration part.

Due to the decentralized nature of the framework, the architecture of it is represented by a network of peers. For this, a detailed view of the peer structure can be viewed in the [Figure 3](#).

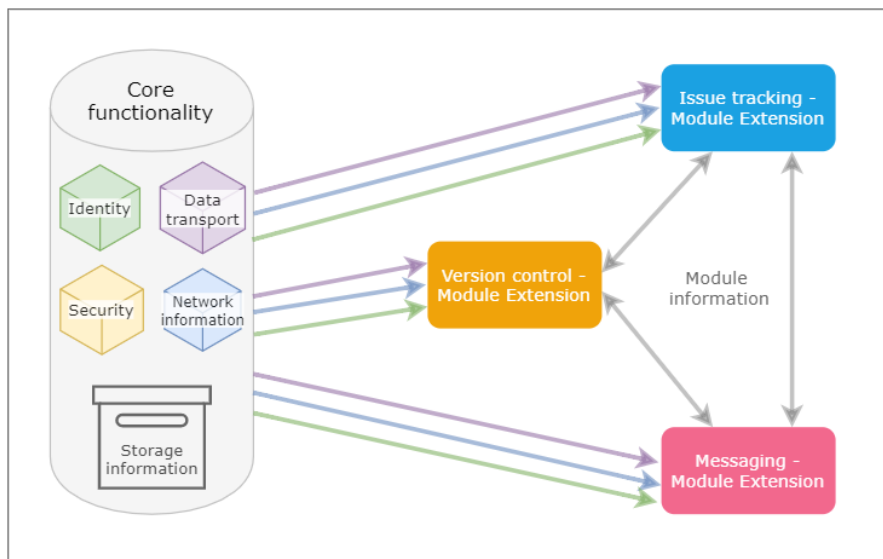


Figure 3: Peer structure of Colligo with a highlight on the core and extension modules.

The core service is represented by the IPFS functionality. Colligo's core concepts involve the identity and authentication of the node, data transportation and network awareness. These services help the connected modules to act in symbiosis with other modules. Each module has access to the peer's core functionality. Apart from that, each node might have useful information for other modules that can be shared.

Each peer stores information about itself and other connected peers for an efficient reach of data in the network. This structure represents the overall framework's state for this thesis.

Colligo can easily support flat or hierarchical organizational structures. Each peer can have assigned a role for specific file or extension component access. Colligo's components can have two types: Core components — currently the P2P component and the Auth compo-

nent. These components are required for the framework to run. They are present in every instance and offer the fundamental functionality in using a decentralized collaboration tool. Extension components e.g. version control component, messaging component are not required but they are needed and help to make the usability and simplicity better for the framework. Each network can be configured in the beginning to use specific components.

These components are available only within the framework and have access to the node's information and make use of the P2P's data transport service. They can exchange information and communicate with each other. To maintain the decentralized concepts, they should adhere to the distributed architecture and make use of the peer information to run.

Colligo can later be extended with core or extension modules. The more modules the more the framework can offer a full-fledged service as a tool for collaboration.

The aim for the proposed work-flow is to be automatized in a way that the presented steps, do not require many user actions and most of them to be carried out under the hood.

4.1 INITIALIZATION

A bird's-eye view of the system is presented in [Figure 4](#).

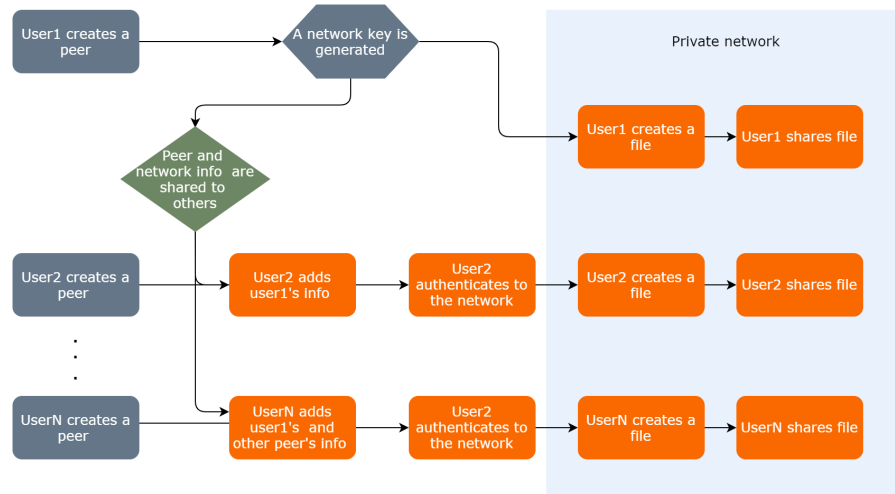


Figure 4: Colligo's work-flow representing the steps that a user can take without considering specific choices for libraries.

The first user to initialize the group project will also be the one to create the network. The presented structure will not address technology-specific information. It will instead present general steps and descriptions that apply for any library used in this role. A brief view of the steps is shown in [Figure 5](#).

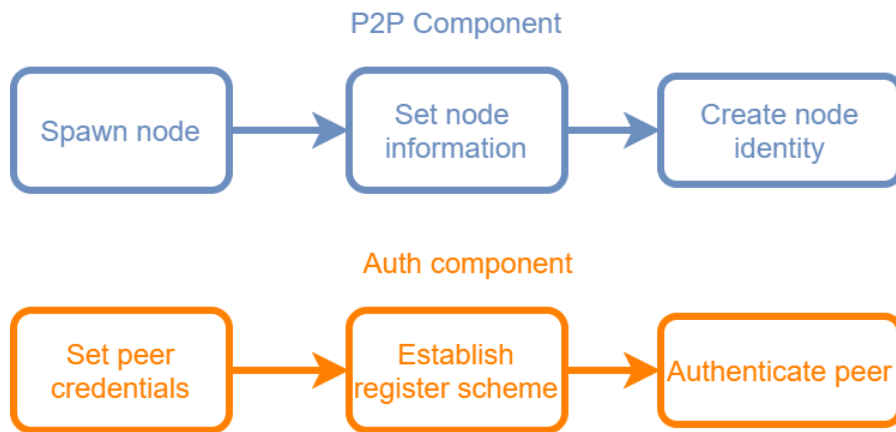


Figure 5: The image shows the main steps of P2P and Auth components. The goal is having fewer steps possible for the users to do so that he can focus directly on the project work instead of setting .

The first component is the *P2P* component. It's used to organize and decide how the team will communicate and how files are distributed. Steps of the P2P component:

- Spawn a node. The P2P system designates an internal ID for each project to differentiate itself from other nodes. This is different from the displayed Identity as the PeerID is used inside the system operations and the displayed Identity is a basic identifier that can be used by the users to list or search the current participants. At this step, it's also determined the location of the files used and created by the framework. This is also the time where the initiator of the project chooses what extension modules to use. From the available components, we can choose basic file distribution within the network or the complete solution to aid the work for each person. Each node has a personal work-space where either just personal information is encrypted or all project-related files are protected to be accessible only within the framework.
- Set node information. In an ideal case, this should be the only time a user's action is required for setting up the network. The ways to identify the user within the network are defined. The information at this step can coincide or overlap with the information set for network authentication.
- Create node identity. The designated work-space is created/assigned and peer's configuration files are created. Personal information is stored in the system inside the specified work-space and its made accessible to both core and extension modules. An optional part is to define a group where the peer will be part of. It can have different rights from other nodes and form a hierarchy in the network. Node identity information is stored,

ready to be shared with the existing and also new peers in the network.

The workspace/directory is connected and represented by a network. A node can connect to different networks. As for each network, the node is required to have a network key, the node can manage and choose on which network to connect. In Colligo, the network can be seen as a repository/project. For each collaboration project, we can have a different network. The P2P component's role is to create a basic file distribution system and offer a simple way of data transfer for the extension modules. Upon node's creation, the authentication information is required. Here the Auth component is called to set out the rules of how other people can register and authenticate to the network. The overall steps in this component are:

- Set peer credentials. Personal information set earlier can be used or for more security, different accreditation can be set.
- Establish register scheme. Peer registration is taking place and other existing peers in the network are made aware of the new peer. The component checks the uniqueness of the peer in the network and validates the information introduced by the user. As an optional small step, in the event of roles and groups existence, this is the moment where also the position of the peer is validated in the network. This can be done via either an extra key or a certain password to be used.
- Authenticate peer. Now that the peer has been registered it gains access to the network. Authentication data is saved in the configuration files to automate the authorization next time the peer connects to the network.

Instead of a classic database here authentication information is encrypted and stored locally for each peer.

For each user, we have *friendly* information (such as Name, E-mail, etc) and network identity. For example, for IPFS each user has a PeerID for each project. When the peer is finally created and connected to the private network it can now create and share data. The user's identity is stored in the network so that on the next connection in the framework the authentication happens both locally and on the network. With the identity created the user has access to request data from either specific peer in the network or a specific group/role.

This has been the description of the initial steps for the usage of Colligo. Next, a brief description of the existing extension components will be given and specific work-flows will be described.

4.2 GENERAL USAGE

The user can now start working on the project and enjoy the advantages of Colligo. For a better experience, extension modules can enhance the work-flow of the team.

Extension components follow a certain template for their structure. All of them have the same access to the peer's information and can be flexible on what data they process and what information they share. We can see in [Figure 6](#) how an extension component can operate.

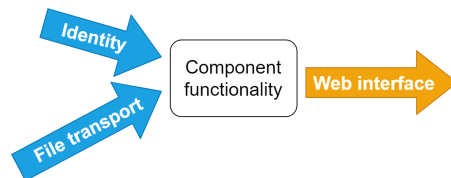


Figure 6: The image shows an extension module's abstract structure. Each extension module can be thought an external independent application that talks to the framework for identity and transport and processes independently its data.

To keep track of file changes nowadays a version control system is indispensable. The version control extension component is used to offer Git commands support for the existing files. Identity and file distribution are assured by the P2P component so that the versioning component can focus only on offering insights about file differences, comparison, etc. Files can be retrieved from other users for comparison or to update with the current progress of the work. The functionality of the component is simple and it does not care about how the data is retrieved but the location and the owner of it.

Another extension module present in Colligo is a messaging module, a component that can assure an extensive decentralized communication protocol for the framework. Same as for all extension modules, the employed identity is the one created by the P2P component. As each extension module can share its functionality or data this opens the door for interesting use cases. One example is that using the communication module we can tag or mention version control commits into discussions. We can also use information from other future extension modules in different extension modules while they can still act independently from each other. These cases make use of all the information present in the framework and unify it for a complete team experience.

For the last extension module, we will talk about issue tracking. While version control functionality is important for a project, one popular function present in all existing collaboration teams is a project management tool. Creating milestones and issues is important for a project's steady development. As a showcase, Colligo has a component that focuses on this aspect. It would use information from the

version control modules to address work changes and it allows planning and organizing the teamwork.

IMPLEMENTATION

Colligo's architecture emphasizes modularity and a high potential for extension. Currently, there are plenty of open-source applications built with a decentralized architecture in mind. Testing the feasibility of Colligo is translated as the simplicity that these tools can be integrated and communicate with other applications.

Colligo stores locally identity information and meta-information about the projects that the user is involved in. For each project, we have a separate folder where we store the content of the project.

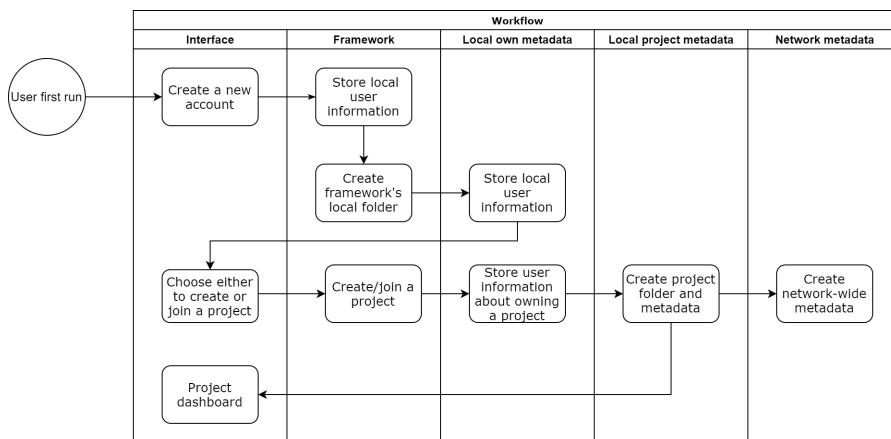


Figure 7: Highlight of the steps that Colligo goes through on the first run.

On the first application run, explained in [Figure 7](#), credentials are only known locally. As the user creates a project, those credentials will be encrypted and stored by every collaborator in the project. This helps to validate future authentication requests. On the initial run, the framework's basic configuration and local path are set.

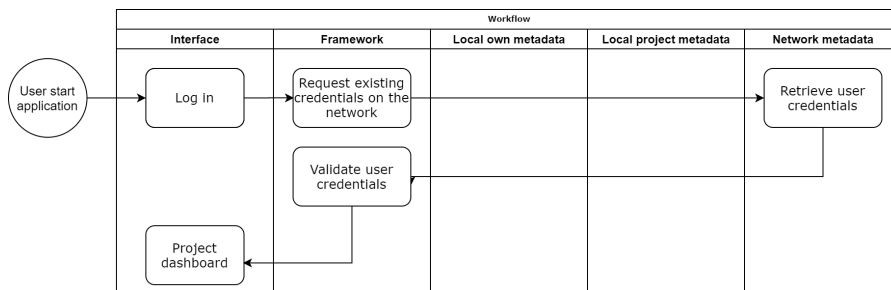


Figure 8: Highlight of the steps that Colligo goes through after the application has already been initialized and a project is existent.

On future application start, shown in [Figure 8](#), the user has only to enter credentials and he will be automatically redirected to the dashboard of the last project opened.

5.1 P2P-SYSTEM = IPFS

For the P2P system, IPFS technology has been chosen. The reason why this initial state of Colligo uses IPFS is the flexibility of it. As it is a very general system it can be used with any kind of data. It is implemented in multiple programming languages, has the option to either run it as an independent process and communicate over CLI commands or HTTP requests or run it as a library inside an application. Another reason is that other tools that are used in Colligo are created on top of IPFS which results in a nice synergy of the functionality.

Colligo's instance runs an IPFS node dedicated to each existing project.

When the user either logs in or create an account, he is presented with two options. Either create a new project or join a project. A project has a few main components :

- a dedicated local folder
- an IPFS node
- two initial OrbitDB databases. An append-only log database to keep account of existing users and a key-value database to store the project files.

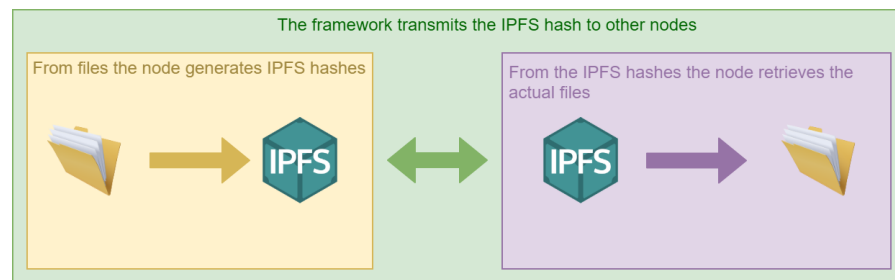


Figure 9: Overview of how Colligo actually offers the file transport between nodes.

The node associated with a Colligo instance is aware of the project files' local path but when it needs to send files to the other nodes, it will send IPFS hash values [Figure 9](#). This will put less overhead on automatic file synchronization and file information retrieval. A user can retrieve the file on its own after he retrieves the hash from the database.

For each user we store the following information :

```

1 {
2
3   previousProject: '',
4
5   projects: [],
6
7   user: {
8
9     name:
10
11     email:
12
13     password:
14
15     localPath:
16
17   },
18
19   p2psystem: 'ipfs',
20
21   ipfsIds: []
22 }

```

Listing 1: Configuration fields for Colligo.

`previousProject` represents the id of the project that was used last time. This is used to facilitate the usage start of Colligo. `projects` array contains projects as data objects created by the user or fetched from the network when joined them. To keep track of the needed configuration files Colligo sets a `localPath` to keep all the files in the same place. Each project can be represented by a new or existing folder that can be located outside of this folder.

On each created project we have the following information:

```

1 {
2
3   name: 'project name' ,
4
5   author: 'name of the author' ,
6
7   p2psystem: 'ipfs',
8     modules: [],
9
10  usersDB: 'OrbitDB database address' ,
11
12  repoDB: 'OrbitDB database address'
13 }

```

Listing 2: Configuration fields for a project.

Here, apart from the basic meta-information, we keep track of the hashes of the project's databases — the one for the users and one for the work files. These OrbitDB databases are stored inside Colligo's folder and their hashes are used to retrieve and publish information.

An alternative implementation of file distribution that has been explored is using only the IPFS functionality. Libp2p, the IPFS's networking protocol offers trigger events to watch when a peer joins the project. This is helpful for peers communication, to share file changes, new files added and peer information.

OrbitDB exposes similar trigger events that are more focused towards database operations events which are also more applicable for this case. There are two reasons that OrbitDB was chosen: OrbitDB is a well-tested library that offers stability and without extra overhead, it offers valuable support for distributed data storage. Secondly, the thesis explores the current ability that a Github alternative can be built. We evaluate how existing tools can communicate and form a complete user experience that can at some point match mainstream centralized applications.

Authentication is represented by basic e-mail/password credentials. In the user's database, we store each user's email and hashed password. The user is allowed to execute database operations only after his submitted credentials match with an existing item in the database. Personal credentials are required to also be preserved locally. The authentication process for the user happens before joining or creating a project. As such, Colligo's interface's access is granted based on local credentials and network operations are granted by confirmation of the user's database.

5.2 PROJECT CREATION

Since IPFS is the only available option so far, the steps involved in creating a project will assume that it's a project with IPFS configuration.

A peer in IPFS has by default public access and is exposed for communication with other public peers. To overcome initial connectivity for a new node in the network and the challenges of decentralized bootstrapping into peer-to-peer systems [16], a newly created IPFS node has some bootstrap nodes set. For a secure collaboration in Colligo's use case, it uses IPFS as a private network. There are two elements to make this work. First, we need to remove all initial bootstrap nodes to avoid connecting to unwanted peers and secondly to we need to generate a swarm key. This key is used to tell network peers to connect and communicate only with peers that share the same key. Colligo creates this key automatically and stores it in the project folder.

5.3 DASHBOARD

The main and most complex view of the application is the project dashboard. Here we have an overview of the core functionality of the framework and also the extension modules.

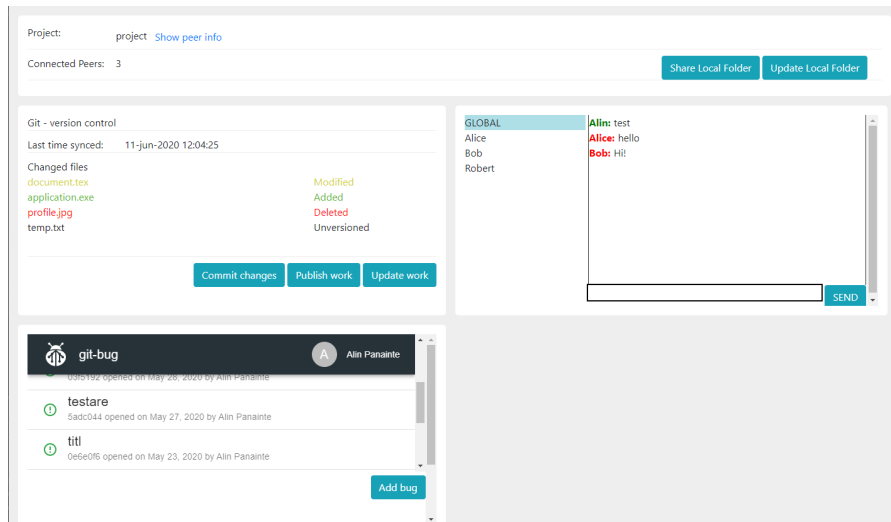


Figure 10: View of a project's dashboard. The web interfaces offers a place for all the needed information regarding the current project.

Figure 10 shows the default view for a project that has all the extension enabled. On the top side, meta-information about the project is presented. There we can see the number of active peers at the current time. Clicking on the `Show peer info` will reveal the necessary information for new users to join the project. The page is updated periodically with information regarding the network, synchronization state and information specific for each extension module.

Some extension components have summary information presented in the dashboard and a link to expand that component view into a separate webpage.

5.4 FILE SYNCHRONIZATION

The key-value database of the projects keeps track of the work files. Each peer can submit a key-value submission with an IPFS hash of the content of work. The database used for repository synchronization contains entries shown in Listing 3 that contain the IPFS hash, author (the account name), the current IPFS ID for the author and the current time.

```

1 {
2   hash:
3
4   author: {
5
6     name: 'name of the author',
7
8     ipfsId: 'ipfs hash'
9     },
10
11   time:
12   }
```

 Listing 3: Database entry for main work collaboration.

To assure high availability of the documents, a possible future work here is to have cloud peers. On the trigger event that the database has been updated with a new work file submission the cloud peers can automatically retrieve and store the files so that if the original author of the work is offline, the work is still available for fetching. If the user chooses for a continuous synchronization, the database update trigger is used and every time someone makes changes, the published changes are fetched locally. They are stored in a different folder inside the project for a future merge with the local work.

For a clear view of the differences between the local work and the fetched updates, the versioning extension module is used to compare files and show line differences for each file.

5.5 EXTENSION MODULES IMPLEMENTATIONS

Colligo allows extensions to store their persistent data across a project in the framework and also share certain information between them.

For each extension inside the project, folder Colligo generates a configuration file that can be used by the extension to store specific information. This allows either to save information in offline cases or continuity when the user starts and stops the application. To store and retrieve the data the framework exposes HTTP routes that can be called with the necessary data.

Each extension has in the dashboard its own allocated space where an iframe points to the web application that is running on each extension.

To facilitate communication with module extensions, Colligo exposes an HTTP API of POST requests with the essential commands for the extensions. Apart from that, a dedicated Orbit database is used to store the information that can be present in the project for all collaborators. This data can be accessed only by the extension that created it and can be updated or synchronized by every collaborator in the project. Later when each extension is discussed, specific information about how they make use of these facilities is presented.

HTTP API for extensions:

- `/identity` - The first call to the framework is made by the extension to get information about node's identity information and existing extension's configuration. The extension sends a name parameter for a simple identification for the framework to know what extension is contacting.
- `/update-config` - Each module extension can store some permanent information related to a project. This information is stored

in the project folder and can be accessed and modified only by the extension. With a name parameter the extension can call this route to update its persistent information.

- `/publish-shared-data` - Each module extension can share and access a pool of shared information by either other extensions or the framework. Regarding the shared data there are two routes available to update and retrieve the data. For this route the extension must provide the extension's name and the data that wants to be shared.
- `/retrieve-shared-data` - A route accessible to every extension to get all the pool of shared data information.
- `/publish-data` - This is the route used by the extension to publish over the network its information. This is different from the shared data as this is data needed for the extension to run. It is only available to the specific extension and is distributed via IPFS and OrbitDB. This method accepts a name parameter and a path of a folder parameter to point to IPFS to create a hash of.
- `/update-data` - In order for a peer to synchronize its extension data it can call this data.

5.5.1 *Git*

Git in its form is a CLI application. For an easier communication between Git and the framework the simple `git`¹ library is used. It provides a Javascript interface of Git for NodeJS and browsers. Using this library is easy to create an extension module in Javascript that communicates with the framework's API and executes Git commands on the local files.

For a truly decentralized Git one matter that needs discussion is the `git remote` command. In Github's case, it contains a link to the repository where users can push and pull the code. In this context, we want to directly push and pull from other peers in the node. For this, we are using two separate folders for the repository. First, we have the standard repository, the one shared by everyone and a bare version of the repository that holds only the files required by Git. A difference between these two types of folders is that the bare repository does not allow commit commands but only push and pull commands on it. As such the default repository will push and pull code from the bare repository, while the bare repository is synchronized across the network. This is also what Github is hosting on their servers — the bare repositories (among other extra functionality).

With simple `git` the versioning extension module offers support for a diff view of local and fetched documents and contains existing

¹ Official website: <https://github.com/steveukx/git-js/>.

Git commands available for use. It has built-in functions for the most used commands but also allows for the execution of raw commands of which input can be parsed inside the application.

The extension calls the identity route at start and gets —if any— the local configuration. This extension shares the bare repository folder through the network. For every push, the local repository executes `git commit` and `git push` commands. Then, a new IPFS hash of the bare repository is created, stored and available for share with other peers. Every time the users pulls the code, the latest hash of the folder is retrieved and then the local repository executes a classic `git pull` operation. The hashes are stored in an append-only OrbitDB database along with the author and the time of creation.

This extension uses the shared data functionality to share information about all the existing commits in the used repository. It also makes uses of the framework's shared data — user names, so that when a commit is created, the user can mention a user or an issue that is created by the Git-bug extension.

5.5.2 *Chat*

Having a live chat between collaborators can be useful when the user wants to quickly discuss a matter without creating a bug using the `git-bug` extension. To make the instant message feature this extension uses an Orbit database. All messages are stored in an append-only feed where an entry contains `sender`, `recipient`, `message`. It allows also for a broadcast where the recipient is `"*"` and everyone receives the message.

The extension listens on the database's events when someone sends a message and displays the message in the web interface.

In terms of identity, it uses the same as the other extension and also for the contact list it makes use of the framework's shared data of project users.

In the interface, the user can select a contact, see the past conversation and send new messages.

5.5.3 *Git-bug - project management*

With `git-bug` we can have another useful functionality existent in centralized alternatives – issue tracking. The library is an executable that, present in a repository folder, exposes a WebUI and a CLI to provide a platform for contributors to address existing issues and plan future work. The extension for this framework works with this executable through the CLI and GraphQL while making use of the WebUI to display the contents.

The extension has a WebUI that is used for display in Colligo's dashboard. Unfortunately, the web interface is still in development

and operations such as creating a new bug are not present in the interface. For these, the extensions have extra buttons to facilitate those features. The extension communicates with the executable through the executable and the GraphQL API.

At the initialization, it calls `/identity` route to the framework for identity information that is needed for `git-bug` to work. Bug information is stored in a lightweight manner in the Git directory. That means to synchronize with other peers we need to synchronize the Git directory similarly to the [Section 5.5.1](#).

5.5.4 *Future extensions*

As we have seen in the aforementioned extensions, it is very easy to create a new framework extension. With the availability of the framework's extension routes in [Section 5.5](#), the only requirements for the extension are to run a minimal web server. With existing libraries, this is a simple task in all major programming language. There are no mandatory routes to be used. All possible requests are just optional features that can be used by the extension.

EVALUATION

In the introduction, Colligo is presented as a decentralized alternative to repository hosting services like Github and also file hosting and distribution services like Dropbox. As such, a comparison between core features present in these services and their corresponding features in the framework represents a point of interest.

Colligo is a decentralized application. This requires a different type of connectivity between collaborators in the project.

In the feasibility section, the decentralized aspect will also be discussed.

Since the presented application is a framework, ways of expanding need to be discussed. As the initial implementation represents the basic functionality, it's important to know how it can be extended and what the limitations are.

6.1 FEASIBILITY

Feasibility covers several, aspects of an application, especially for a framework. Starting from user experience, responsiveness until inter-application communication.

This section is separated into two parts. First, there will be discussions about Colligo's features, performance, and how it compares to existing alternatives. The second part is about expanding the application's functionality. Details about implementing extension modules and their requirements are the focus.

6.1.1 *For users*

6.1.1.1 *Initialization*

On Dropbox, account creation can take place either on their website or directly inside the application. The user has to download the application, wait for installation and then sign in or create an account. After small configuration to set up directory path and ways of synchronizing, the application is ready to use.

In Colligo, the first time the user launches the application, it displays the register page shown in [Figure 11](#). Here, the user creates their account with the desired credentials and types the path where the framework can store all of its data regarding user and framework configurations. Before displaying this page, the framework check for

any existing users. If there is no previous initialization data, the *Existing account* button is not displayed.

Next page is project setup [Figure 13](#). Here the user can either create a new project or join an existing one.

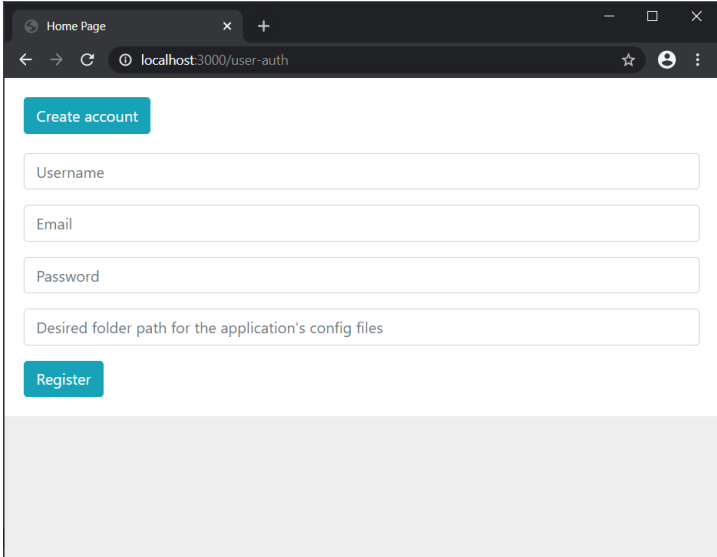
For project creation path, the user specifies the project's name and the path to store project information. Lastly, the user can choose what modules to include for the project and choose the preferred p2p system.

Currently, Colligo implements only IPFS for this option. From extension modules, the users can choose either Git repository features, issue tracking interface and a messaging module. Usage of extension modules is not mandatory. A project without them works with basic features for collaboration.

To join an existing project the user can select the *Join a project* button. To start working on an existing project, the user needs the `swarm.key`'s contents to join the IPFS's swarm and bootstrap nodes' information to retrieve information about the project.

Joining an existing project requires that at least one node that is already in the project is online. This is necessary so that the framework can retrieve the needed information regarding the OrbitDB's databases for both the project's work and the extension modules.

In the background, the application creates the necessary folders and configuration files. For each project, a repository folder is created inside the folder created at the specified project path. This is where the user can store its work documents.



The screenshot shows a web browser window with the address bar displaying 'localhost:3000/user-auth'. The page has a dark header with 'Home Page' and navigation icons. The main content area is white and contains a registration form. At the top of the form is a blue button labeled 'Create account'. Below it are four input fields: 'Username', 'Email', 'Password', and 'Desired folder path for the application's config files'. At the bottom of the form is a blue button labeled 'Register'. The bottom of the browser window shows a grey taskbar.

Figure 11: The first step in the application initialization process.

Compared to the centralized alternatives the initialization is not much different. In Colligo for account registration, only the user's details and project path are needed. We can see that from a usability

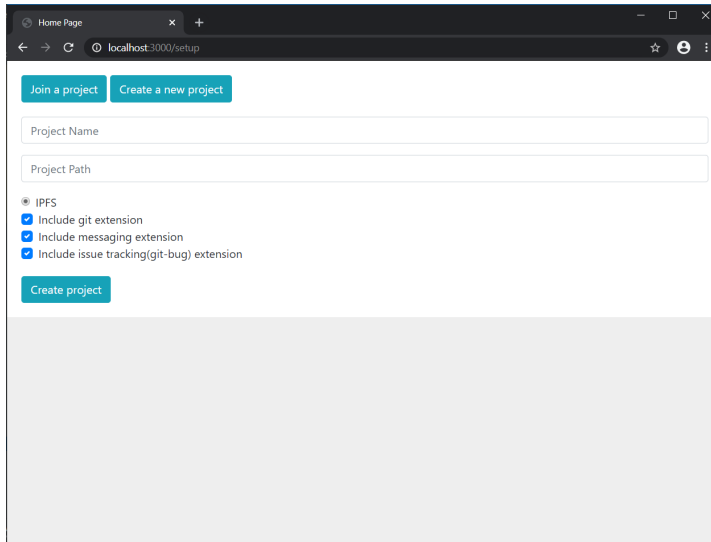


Figure 12: The second step in the application initialization process. The user can either create a project or join an existing one. Under the buttons it's displayed the create project interface.

point of view we can have a decentralized alternative that can be tailored to be user-friendly for better adoption.

6.1.1.2 File synchronization

On file management, the most common operations are synchronization and distribution.

On Dropbox, to upload a file/folder the user can drop the documents in the Dropbox folder and they are automatically uploaded in the cloud servers. To share files, the user creates a shareable link corresponding to the file and shares it with other people. Here the advantage is that with that shareable link it can be available for anyone to access the file without creating an account or install Dropbox. This aligns with the purpose of quick sharing documents without concerning collaboration functionality.

On Colligo, similarly, the user can publish documents with other people. After the project setup, the users are redirected to the dashboard shown in [Figure 10](#). On the top side of the page, the user can see useful information about the IPFS node instance currently running and can access the basic features. They can see how many users are currently active on the IPFS's swarm and clicking on the *Show peer info* button it displays the required information for other users to join the project: the swarm.key file's contents and local addresses for bootstrap nodes.

There are two options for project collaboration. The users can either *Share local folder* or *Update local folder*. The first option will read the contents of the repository folder's contents and create an IPFS hash that will be stored in an entry in the OrbitDB repository [Listing 3](#).

Figure 13: The figure shows the Join project page. The user can join an existing project by filling project information

This can be retrieved by other nodes to fetch the current work of the user.

The *Update local folder* button will retrieve the latest submitted progress on the project work. From the database, it retrieves the latest entry that contains an IPFS hash and recreates its contents into the repository-sync folder. The user can navigate to the folder and manually review and replace their local work with the updated one.

On the dashboard, there are also the extension modules that will be discussed in the following subsections.

The dashboard aims to offer basic features and be accessible without the overhead. With just one click of a button, the user can either share or synchronize its work.

One feature seen in centralized alternatives that can be useful is a shareable link option for either a specific folder or file. Currently, only users that use the framework can exchange files. This is not implemented in this first implementation but can be extended with a module extension.

6.1.1.3 Versioning

On version control, we can compare features from centralized solutions for repository hosting with the framework's versioning module.

Centralized solutions like Gitlab, Github, Bitbucket, etc offer the option to either create a project (repository) on their servers and then clone it locally for contributions or import an existing project from a

bare repository. On these solutions with minimal configuration of the project meta-data, the user can start working on the project and enjoy the available features. From there, they have a web interface for small editing and committing changes. Otherwise, local users can push and pull commands with classic Git commands on the remote servers.

Colligo has a simple and streamlined work-flow. The users select in the beginning a folder path. If the folder is already an existing Git repository it will show the features for this repository. For an empty folder, a new Git repository is automatically created to utilize the module's features. The user can issue commit, push and pull commands from inside the module's interface.

Colligo offers issue tracking in the form of an extension module. Each module has its settings such as folder path, which has to be set initially. After this, the simplicity of managing and using these features are similar to the centralized version.

From the interface point of view, the user experience for all the basic features is the same as mainstream centralized solutions.

6.1.1.4 *Issue tracking*

The git-bug extension module can be used to create issues around a Git repository. Users can mention other users and existing issues. The current implementation is showcasing the web interface already shipped with the git-bug library.

The current implementation of the interface is not completed and as such, some features such as creating a new issue are not present. To support this, the extension modules offer an auxiliary web interface that complements the issue creation feature and communicates with the library's web interface. Both the main web interface and the auxiliary one are present in one section in the dashboard.

The user can navigate the main interface to visualize the existing issues and comment on them.

6.1.1.5 *Performance comparison*

Colligo has been developed and tested on a machine with a 2.20 GHz processor, 16GB RAM and SSD disk. The initialization and processing times should not be affected by a machine's specifications as the application is not computationally expensive and communication relies on the internet connection.

Currently, after the user either created or joined a project it takes 3 seconds to initialize the IPFS node and less than a second to initialize OrbitDB instance.

Testing IPFS performance can be an extensive task. For Colligo, it was possible to measure the time it takes to process files into IPFS hashes and the time to resolve an IPFS hash into files.

The processing time is more focused on the IPFS process. This process involves one `ipfs add` command for a folder. This command translates the IPFS process of breaking the data into chunks are then creating a hash for each file that contains a link and content of the file.

For the fetching process, there are two parts involved. First, there is the command of `ipfs get` command that fetches all the data chunks that are returned as a buffer and the second part is writing, from that Buffer, the files on the disk.

The results can be seen in Figure 14. The tests showcase three scenarios that involve randomly generated binary files. The resulted times are a result of an average of 20 iterations for each step. First, we measure the time to process and fetch a single file with variations on the size between 100 MB and 1000 MB. Next, there are two scenarios analysed. First the processing and fetching time evolution for a variable number of 10 MB files and another case for a group of 10 files of different sizes.

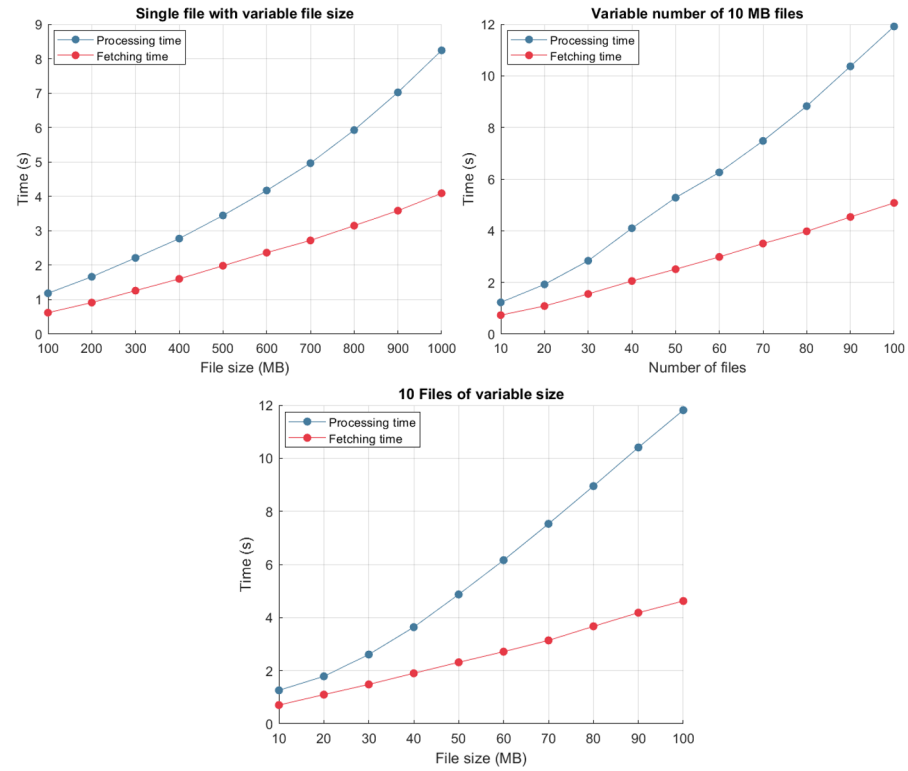


Figure 14: Evaluation of the processing time and the fetching time for different scenarios.

We can notice that for all cases, the processing time gets much higher than the processing time. Even if the fetching process involves two steps and might be a subject for further optimization, the processing time is still higher.

A possible test scenario that could have also been tested is how IPFS performs when an existing folder has been processed for a hash and new files would be later added. How would the processing time for the existing folder with new content be affected? For this, it has been noticed that there was no performance gain at all. IPFS does not cache indexes for existing files. File chunks are stored but not used to improve further processing times that include previously indexed files.

Variables such as seeders, leechers, latency and file size greatly influence the performance. The IPFS team is currently developing for the Go implementation, a Testground where a different algorithm can be tested to see how it affects IPFS's performance. Up to date and detailed information can be observed on their Github page¹. Briefly, the analysis is focused on the time to fetch files in different scenarios. It has been observed that latency does not affect the time to fetch the content. With files not exceeding the bandwidth size, in the scenario of 1 leecher and 1-4 seeders, IPFS achieved a fetch time of 2 seconds.

A more detailed approach is analysed in [17] where the highlighted operations are resolving and downloading operations. The paper compares IPFS with HTTP on I/O operations. It concludes that I/O performance is greatly influenced by the access patterns of online applications. Resolving and downloading processes can represent an impediment for the remote node's read operations. In this thesis's context, we can present the HTTP protocol as the representation for centralized solutions for file synchronization and collaboration.

6.1.1.6 Scalability

Colligo can work offline and for just one person. A user can 'share work' to create an IPFS hash of the current work and later revert to the state of that past IPFS hash. From one person it easily scales to a whole team, without losing data synchronization. Currently, all IPFS hashes are stored in an Orbit database. An extension can backup or navigate through past IPFS hashes and retrieve their contents. With a bigger team, the content retrieval speed is not affected. As a user retrieves the content of an IPFS hash, the IPFS nodes announce the availability to later distribute itself that content. This is done under the hood by IPFS. By design, the instance will only store contents that have been explicitly requested by the user.

6.1.1.7 Security

IPFS nodes have communication secured with the swarm key. They can only communicate and be discovered with nodes that share the same key. The key is stored in the application folder encrypted with

¹ IPFS Go Implementation Repository: <https://github.com/ipfs/go-ipfs/issues/6782#issuecomment-579973116>.

the public key generated from the IPFS node. One possible extra layer of security would be for the transferred files. Currently, anyone that has possession of the hash can download the contents that it represents. To overcome this, an asymmetric key encryption scheme can be used on the files. The work-flow would be slightly modified. The sender would receive the public key of the receiver. It would encrypt the content of the documents to be sent. An IPFS hash would then be generated and shared with the receiver. In this way, no matter who gets into the possession of the IPFS hash needs to have the corresponding private key.

6.1.2 For developers

The minimal requirement for a module extension is to be able to act as a web server. The extension should be able to communicate through HTTP requests with the framework routes presented in [Section 5.5](#).

The module can have a web interface that can be showed in Coligo's interface. Each module has access to the available shared data and can process the information.

It is recommended if possible to use the identity and transport services for optimization and to avoid redundancy. Otherwise, the module is free to implement any type of operation that uses the available information.

A basic web-server that listens and launches web requests can easily be achieved in under 500 LoC and with much fewer lines of code in certain programming languages or used libraries.

If we include all the available routes from the framework they can be covered in less than 100 LoC with minimum processing.

As an example, the Git extension module, excluding the boilerplate Node.js express code, has ~200 LoC for the communication with the framework and another ~200 LoC for internal processing. In [Listing 4](#) routes for the Git extension are defined. Apart from Git related routes, (set-repo and commit, etc.) the routes are common on the other extension modules.

```

1  var express = require('express');
2  var router = express.Router();
3  const internal = require('../app/internal');
4  const framework = require('../app/framework');
5
6  /* GET home page. */
7  router.get('/', async function(req, res, next) {
8
9      if(global.connected === false) {
10
11  return res.redirect('/loading');
12      }
13
14  if(global.moduleConfig.repoPath === undefined ) {
15

```



```

16 return res.redirect('/set-repo');
17 }
18
19 let diffChanges = internal.GetFilesStatus;
20
21 return res.render('home', { fileChanges:diffChanges});
22 });
23
24 router.post('/commit', async function (req, res, next) {
25
26 console.log('received commit message ',req.body.message);
27 let response = await internal.CommitRepository(req.body.message);
28
29 return res.json(response);
30 });
31
32 router.post('/push', async function (req, res, next) {
33
34 return res.json(await internal.PushRepository());
35
36 });
37
38 router.get('/set-repo', async function (req, res, next) {
39
40 return res.render('set-repo');
41 });
42
43 router.post('/set-repo', async function (req, res, next) {
44
45 global.moduleConfig.repoPath = req.body.repo;
46
47 if (global.moduleConfig.identity.is_author === false) {
48
49 //retrieve data
50
51 await framework.SynchronizeData('git-bare-repo', global.moduleConfig.
    repoPath);
52
53 await internal.CreateRepository(global.moduleConfig.repoPath);
54 } else {
55
56 await internal.CreateRepository(global.moduleConfig.repoPath);
57 }
58
59 internal.SaveConfig();
60
61 internal.CreateBareRepo(global.moduleConfig.bareRepoPath);
62
63 internal.InitializeGitConfig();
64
65
66 return res.redirect('/');
67 });
68
69 router.post('/getfilestatus', async function (req, res, next) {
70
71 return res.json(await internal.GetFilesStatus());

```

```

72
73 });
74
75 global.connected
76 router.post('/pull', async function (req, res, next) {
77
78   return res.json(await internal.PullRepository);
79 });
80
81 router.post('/status', async function (req, res, next) {
82
83   return res.json({status:global.connected});
84 });
85 router.get('/loading', async function (req, res, next) {
86
87   return res.render('loading');
88 });
89
90 router.post('/get-shared-data', async function (req, res, next) {
91
92   return res.json(await framework.RetrieveSharedData());
93 });
94
95
96 module.exports = router;

```

Listing 4: Webserver part of the Git extension.

The part where the extension communicates with the framework is defined in the `framework.js` of [Listing 5](#). Using HTTP requests, the extension gets identity information from the framework and manages the data. Shared data is managed on the `\extension\publish-shared-data` and `\extension\retrieve-shared-data`. With the first one, the Git extensions module sends data about the existing commits that can be used by other extensions and with the latter route we get all the existing shared data among the extension modules.

```

1  const axios = require('axios');
2  const internal = require('./internal');
3  const path = require('path');
4  exports.GetIdentity = () => {
5
6    axios.interceptors.response.use(
7
8      function (response) {
9
10       global.connected = true;
11
12       return response;
13     },
14
15     function (err) {
16
17       if (err.code === "ECONNREFUSED") {
18         //we cannot reach the framework.
19

```

```

20         console.log('Cannot reach framework.')
21     }
22
23
24     return Promise.reject(err);
25
26 }
27
28 );
29
30 axios.post('http://localhost:3000/extension/identity', {
31     name: 'git',
32
33 })
34
35 .then(async (res) => {
36
37     if (res.data.status === true) {
38         let new_identity = {
39             name: res.data.identity.name,
40             email: res.data.identity.email,
41             is_author: res.data.identity.is_author,
42             projectPath: res.data.identity.projectPath
43         };
44
45         global.moduleConfig.identity = {...global.moduleConfig.
46             identity, ...new_identity}; //update new identity
47
48         global.moduleConfig.bareRepoPath = path.join(global.moduleConfig.identity
49             .projectPath, 'git-extension', 'bare-repo');
50         internal.SaveConfig();
51         console.log('Retrieved identity for Git successfully!');
52
53         internal.GetCommits();
54         console.log('done initializing Git module');
55
56     } else {
57         console.log('Failed to get valid identity information.');

```

```

75 | })
76 |
77 | };
78 |
79 |
80 | exports.PublishSharedData = (sharedData) => {
81 |
82 |   axios.post('http://localhost:3000/extension/publish-shared-data', {
83 |
84 |     name: 'git',
85 |
86 |     data: sharedData
87 |   }
88 |   )
89 |
90 |   .then((res) => {
91 |
92 |     if (res.data.status) {
93 |       console.log('Shared data published successfully!');
94 |     }
95 |   })
96 |
97 |   .catch((error) => {
98 |
99 |     console.error(error)
100 |   })
101 |
102 |   exports.RetrieveSharedData = () => axios.post('http://localhost:3000/
103 |     extension/retrieve-shared-data', {
104 |
105 |     name: 'git',
106 |
107 |     data: sharedData
108 |   }
109 |   )
110 |
111 |   .then((res) => {
112 |
113 |     if (res.data.status) {
114 |       global.sharedData = res.data.content;
115 |       return {status:true,content:global.sharedData}
116 |     }
117 |   })
118 |
119 |   .catch((error) => {
120 |
121 |     console.error('Error retrieving shared data:',error.toString());
122 |   })
123 |
124 |
125 |
126 |
127 |
128 |
129 |
130 |

```

```

131
132 exports.PublishData = (sourcePath, folderName) => {
133
134   // publishing data for this extension modules means we will publish hash
      for the bare repo
135
136   axios.post('http://localhost:3000/extension/publish-data', {
137
138     name: 'git',
139
140     path: sourcePath,
141
142     folder: folderName,
143
144   }
145     )
146
147   .then((res) => {
148
149     if (res.data.status) {
150       console.log('Bare repo published successfully!');
151
152     }
153
154   })
155
156   .catch((error) => {
157
158     console.error(error)
159
160   })
161 }
162
163 exports.SynchronizeData = (folderName, targetPath) => {
164
165   // synchronizing data for this extension modules means we will update the
      local repo
166
167   axios.post('http://localhost:3000/extension/update-data', {
168
169     name: 'git',
170
171     path: targetPath,
172
173     folder: folderName
174
175   }
176     )
177     .then((res) => {
178
179       if (res.data.status) {
180         console.log('Bare repo updated successfully!');
181
182       }
183
184     })
185

```

```

186 .catch((error) => {
187
188 console.error(error)
189
190 })
191 }

```

Listing 5: Communication with the framework section of the Git extension.

6.2 PRE-CONDITIONS OF A COLLABORATION PROCESS

In the case of IPFS, the TCP (Transmission Control Protocol) ports need to be open on system-defined ports for communication. This is required for IPFS nodes to find each other.

6.3 CHALLENGES

Using IPFS, focused on generality, it is not straightforward to persist in a folder structure when creating a hash. Currently, when `ipfs add` command is called on a folder, it returns objects with a hash for each file that preserves internal folder structure in the folder and lastly an object for the whole folder. This response object represents a hash for IPFS nodes to retrieve the data but it does not contain the folder structure.

To overcome this limitation Colligo uses Globsource, an IPFS data structure to preserve the folder structure. The `AddFolderToIpfs` function of `??`, accepts a folder path. For each file, a hash is created and stored to be sent. The resulting object contains the name of the folder and information about each file (path and hash).

```

1  async function AddFolderToIpfs (folderPath) {
2
3  console.time("adding ipfs files");
4
5  let files = [];
6
7  try {
8
9  for await (const file of ipfs.add(globSource(folderPath, {
10
11  recursive: true
12
13  }))) {
14
15  files.push(file);
16
17  }
18  } catch (e) {
19
20  console.log('Unable to add folder to IPFS:', e.toString());
21  }

```

```

22
23 return {
24
25   folder: folderPath,
26
27   files: files,
28   }
29 };

```

Listing 6: Adding folders recursively to IPFS.

Having extensions to run in different processes can result in some desynchronizations between the framework and the modules extensions. This has been implemented in this fashion to isolate the extension modules from the framework and run separately.

Another challenge is testing scalability. As presented before, the IPFS team is developing a Testground where different scenarios for the networks and peers can be tested.

6.4 LIMITATIONS

6.4.1 *For users*

Currently, in the Javascript implementation of IPFS, the connectivity is not fully implemented. To overcome this, it is mandatory to use Bootstrap nodes. Another issue on IPFS is to connect to remote nodes when running on a machine over WIFI networks. In the future, these limitations will be solved and it will allow for a smoother integration that would make the decentralized architecture look similar to centralized applications from a user experience point of view.

Another limitation here is the missing Oauth functionality. The users need to create a new user account for Colligo. On existing centralized solutions, it is common to use existing e-mail accounts to use without having to remember extra credentials. For a purely distributed system, this can be a challenge as there is no central authority to verify and authorize the use of existing Oauth tokens to be used. A future solution here can be to assign the node that creates a project as the authority of the network. This node could register for an API key that can be used for Oauth registrations by other nodes in the network.

6.4.2 *For developers*

Limitations here come from the implemented p2p system. For example, on IPFS it can be a challenge to keep track natively of the original file structure. Colligo allows for expansion on using a different p2p technology that can offer different features and overcome other systems.

Extension modules need to rely on HTTP API. Currently, there are no other ways to communicate with the framework. When there are multiple extension modules that each make frequent requests to the framework, the user's machine can encounter system slow down events. This can limit the complexity that the extension modules can have.

6.5 DISCUSSION

Colligo has a manual synchronization of the work documents. When a new person joins a project, the application replicates all the existing OrbitDB databases to give access to all the project-specific data. Collaborators can manually synchronize their work with the latest updates from others by fetching the latest entry from the repository database that contains IPFS hash. With the IPFS hash, the user can download file contents from the network nodes that have the specific hash's content.

When we are talking about either centralized or decentralized application, the ideal case is both. This also applies to this framework. In a real use case scenario, it is not reliable to have node instances only for people working. There might be times when only one person is online and that person wants to get the latest version of the work. This is not possible without an online node that holds that piece information. To overcome this, cloud instances can be employed. With minor tweaks, we can have ephemeral nodes, that are hosted on a cloud solution and for each submission in the database, it will retrieve its contents and have them available. In this fashion, we can always have data available for synchronization.

Migration

Colligo relies on existing tools that do not require complex set up. The file synchronization feature works the file management with the file-system of the operating system. Adding Colligo to an existing work-flow or removing it, would not affect the structure of the work documents. The version control extension component is based on Git, a tool that already exists on the machines of most of the developers. Colligo creates separate branches to synchronize the work through Git so that an existing repository can be used without interfering with an existing work-flow. For the repository synchronization, the Git module adds a separate remote for the framework, so the user can still push code to an existing remote of a centralized solution.

CONCLUSION

In this thesis, Colligo is introduced as a decentralized collaboration framework. The framework's implementation represents a snapshot of the current technologies that can be used for decentralized applications in various domains: P2P systems, authentication communication and versioning. The framework encompasses file synchronization, version control, issue tracking and messaging features. Colligo aims to represent available tools and also serve as a starting ground for the concept of a decentralized collaboration system. It allows the possibility to expand and explore alternatives for the existing tools and also encourages combinations of different tools' features.

Answering hypothesis questions:

- How feasible is a decentralized collaboration framework?

Currently, there are tools to create a suitable environment for a collaboration framework. At first, there were P2P systems with a focus on being a distributed file protocol. Nowadays we have tools like IPFS and Dat protocol that aims to provide a foundation for the development of decentralized applications. With tools that have distributed functionality and the aforementioned P2P systems, it is more feasible to build decentralized applications. For a collaboration tool, there are already libraries that achieve specific aspects of the collaboration and can be plugged together to form a complete experience.

- What are the requirements for a collaboration process in a decentralized manner?

An important aspect for a decentralized operation is continuous availability. If no peer is online, then there is no way to retrieve up-to-date data. This can be accomplished with either cloud peers or a hybrid architecture of both centralized and decentralized structure.

- What are the challenges of decentralized collaboration?

The tools used for Colligo are in active development. Versions move fast and there is a great chance there are breaking changes. This applies not just for the smaller tools but also the P2P system. The transfer speed for files can be problematic in some cases. Currently, people have high-speed internet but when two peers have an inferior connection, a central server with high speed and reliable connection can be an advantage.

In [Section 6.1.1.5](#) we presented different tests. For file processing on the testing machine, IPFS can process and resolve a 1 GB file in approximately 12 seconds excluding the time to retrieve the file from a remote peer. It has also been discussed how IPFS's efficiency on the network is greatly influenced by the access patterns.

7.1 TOOLS AND WORK ASSESSMENT

Developing Colligo gave great insights into the used tools. It starts with matching available tools with a P2P system that is flexible and meets the application's needs. Then, it continues with polishing performance and fixing technical errors that would not be obvious in the design phase.

IPFS has been an interesting tool to work with. It is presently under active development and maybe with no ideal performance. Currently, the main implementations are in Go and Javascript, while other implementations for Python and Rust are at the beginning. Colligo uses the Javascript implementation of IPFS. With the current state, it can be a reliable tool for a prototype of a decentralized application. Frequent updates might include breaking changes but also performance improvements. For the Javascript implementation at the moment of writing the thesis, there were issues with the DHT routing. There are some workarounds, but this can pose difficulties. The PubSub feature, a very useful tool to send instant messages to nodes, is in an experimental phase and it is not very reliable.

OrbitDB, the storage tool for Colligo is heavily relying on IPFS and especially on the PubSub feature. Because of this, sometimes, database replication can be delayed.

Both IPFS and OrbitDB are on their way for a fully stable release but they offer great features with a simple implementation. It is clear and easy to have different nodes to exchange data with IPFS and also to store data with OrbitDB, get stored records and have a lightweight database with a distributed architecture.

In Javascript, there are plenty of choices for Git wrappers and implementations. This offers great variety in choices. For Colligo, a Git wrapper is used. Using a wrapper instead of implementation decreases the chance to stumble upon some feature that is needed but has not been implemented. In the worst case, command-line calls to the Git can be used. A disadvantage is that the user is required to have Git installed.

For issue tracking, Colligo uses `git-bug`. It has plenty of features but most of them are focused on command-line interaction. It has a web interface that aligns excellent with Colligo functionality, but the lack of some important features resulted in some hacky modifications to allow a basic functionality in Colligo's interface.

With the development of Colligo, I got the chance to experiment first-hand how a P2P application works, what are the technical challenges and how to overcome them. With multiple independent components running at the same time I also had the chance to see different methods of communication between application and trigger errors that were results of an unstable implementation and/or unreliable data structures. Having IPFS relying completely on asynchronous functions it gave me great insights on how to manage the code flow and race conditions.

7.2 USER TESTING

A missing aspect from the thesis is a discussion and feedback from a Colligo usage by other people than the author. The thesis is a feasibility analysis for a decentralized application. It analyses different aspects and challenges that come with developing decentralized applications using libraries that are in active development.

Apart from the discussion about the design, implementation is important to also get insights from users and developers. Feedback of the general usability and stability from users can help the development of Colligo to reach more mainstream and popular use. Since Colligo is fundamentally a framework, there is huge potential on the features that can be developed and used in a decentralized manner. With the general implementation of the extension modules, I think Colligo can be used to experiment with other features and can also be seen as a testbed for decentralized applications.

7.3 FUTURE WORK

From an implementation point of view, there are several different roads that the development can go. Apart from IPFS, as presented, there is also the Dat protocol. Implementing both P2P systems can give some insights for a more in-depth comparison between those two and also can give Colligo extra features that are not available with IPFS.

Different libraries can be implemented in Colligo. For example, tools like Matrix, require a central server implemented to be used. This is an opportunity to analyse a hybrid application that uses a decentralized identity and data with the communication protocols offered by Matrix.

7.4 CLOSING THOUGHTS

The open-source community has various tools that are built with a distributed architecture and are easy to be plugged with others. There

are also different alternatives for P2P systems which represent the backbones of distributed applications.

From this thesis, we can learn what are the current tools for implementing a decentralized tool, what are the requirements and the limitations for it.

Colligo is an example of how a decentralized collaboration can be built, what are the uses and how it compares to mainstream centralized solutions. It is not a perfect solution and it can be seen as a foundation for decentralized applications. With all the limitations and challenges identified and discussed in the thesis, Colligo provides space for improvements and future work that can bring Colligo to be a significant alternative to popular choices for file synchronization and collaboration.

BIBLIOGRAPHY

- [1] Richard L Daft and Robert H Lengel. "Organizational information requirements, media richness and structural design." In: *Management science* 32.5 (1986), pp. 554–571.
- [2] Christoph Piechula. "Sicherheitskonzepte und Evaluation dezentraler Dateisynchronisationssysteme am Beispiel »brig«." MA thesis. Hochschule Augsburg University of Applied Sciences.
- [3] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.
- [4] Juan Benet. "Ipfs-content addressed, versioned, p2p file system." In: *arXiv preprint arXiv:1407.3561* (2014).
- [5] Juan Benet. "IPFS: The Distributed, Permanent Web or how I learned to stop worrying and love the Merkle Web." EE380 Computer Systems Colloquium. 2015. URL: <https://ee.stanford.edu/event/seminar/ee380-computer-systems-colloquium-12>.
- [6] Shangping Wang, Yinglong Zhang, and Yaling Zhang. "A blockchain-based framework for data sharing with fine-grained access control in decentralized storage systems." In: *Ieee Access* 6 (2018), pp. 38437–38450.
- [7] Muhammad Salek Ali, Koustabh Dolui, and Fabio Antonelli. "IoT data privacy via blockchains and IPFS." In: *Proceedings of the Seventh International Conference on the Internet of Things*. 2017, pp. 1–7.
- [8] Maxwell Ogden, Karissa McKelvey, Mathias Buus Madsen, et al. "Dat-distributed dataset synchronization and versioning." In: *Open Science Framework* 10 (2017).
- [9] Dominic Tarr, Erick Lavoie, Aljoscha Meyer, and Christian Tschudin. "Secure Scuttlebutt: An Identity-Centric Protocol for Subjective and Decentralized Applications." In: *Proceedings of the 6th ACM Conference on Information-Centric Networking*. 2019, pp. 1–11.
- [10] Michael Mure. *Git-bug*. <https://github.com/MichaelMure/git-bug>. 2020.
- [11] Dick Hardt et al. "The OAuth 2.0 authorization framework." In: *RFC 6749, October* (2012).

- [12] Shibasis Patel, Anisha Sahoo, Bhabendu Kumar Mohanta, Soumyashree S Panda, and Debasish Jena. "DAuth: A Decentralized Web Authentication System using Ethereum based Blockchain." In: *2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN)*. IEEE. 2019, pp. 1–5.
- [13] Mohamed Tahar Hammi, Patrick Bellot, and Ahmed Serhrouchni. "BCTrust: A decentralized authentication blockchain-based mechanism." In: *2018 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE. 2018, pp. 1–6.
- [14] Mohamed Tahar Hammi, Badis Hammi, Patrick Bellot, and Ahmed Serhrouchni. "Bubbles of Trust: A decentralized blockchain-based authentication system for IoT." In: *Computers & Security* 78 (2018), pp. 126–142.
- [15] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system." In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.
- [16] Jochen Dinger and Oliver P Waldhorst. "Decentralized bootstrapping of P2P systems: A practical view." In: *International Conference on Research in Networking*. Springer. 2009, pp. 703–715.
- [17] Jiajie Shen, Yi Li, Yangfan Zhou, and Xin Wang. "Understanding I/O performance of IPFS storage: a client's perspective." In: Jan. 2019, pp. 1–10. DOI: [10.1145/3326285.3329052](https://doi.org/10.1145/3326285.3329052).