

A decentralized Git version control system

- A proposed architecture and evaluation of decentralized Git using DAG-based distributed ledgers

Christian Habib
Ilian Ayoub

Supervisor : Manali Chakraborty
Examiner : Mikael Asplund

Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: <http://www.ep.liu.se/>.

© Christian Habib
Ilian Ayoub

Abstract

This thesis proposes an implementation for a decentralized version of the Git version control system. This is achieved using a simple distributed DAG ledger. The thesis analyzes how the decentralization of Git affects security. Use and misuse cases are used to compare and evaluate conventional Git web services and a decentralized version of Git. The proposed method for managing the state of the Git project is described as a voting system where participants in a Git project vote on changes to be made. The security evaluation found that the removal of privileged roles in the Git version control system, mitigated the possibility of malicious maintainers taking over the project. However, with the introduction of the DAG ledger and the decentralization, the possibility of a malicious actor taking over the network using Sybil attack arises, which in turn could cause the same issues as a malicious maintainer.

Acknowledgments

We would like to thank our examiner Mikael Asplund and our supervisor Manali Chakrabourty for their guidance throughout the thesis. We would also like to thank our family and friends for their continuous love and support during our studies.

Contents

Abstract	iii
Acknowledgments	iv
Contents	v
List of Figures	vii
List of Tables	viii
Glossary	1
1 Introduction	2
1.1 Motivation	2
1.2 Aim	3
1.3 Research questions	3
1.4 Approach	3
1.5 Delimitations	4
2 Background	5
2.1 Git	5
2.1.1 Git Branching Model	5
2.1.2 Git Internals	6
2.1.3 Permission Structure	7
2.2 Cryptography	7
2.2.1 Public-key Cryptography	7
2.2.2 Digital Signatures	8
2.2.3 Hash Functions	8
2.3 Merkle Trees	8
2.4 Blockchain	9
2.4.1 Consensus algorithms	9
2.4.2 Distributed DAG Ledger	9
2.5 DLT Vulnerabilities	10
2.5.1 Sybil Attacks	10
2.6 Threat Modeling	10
2.6.1 Misuse Cases	11
3 Related works	12
3.1 Distributed ledger technologies	12
3.2 Version control on the blockchain	13
3.3 Threat modeling	13
3.4 Motivation	14

4	Design choices	15
4.1	System design	15
4.2	Design Considerations	16
4.2.1	Distributed DAG ledger	17
4.2.2	Git	18
4.2.3	Distributed DAG ledger permissions	18
4.2.4	Consensus	18
4.3	System architecture	19
5	System architecture	20
5.1	System overview	20
5.2	Light-weight Git	20
5.3	Distributed DAG ledger	22
5.4	P2P network	23
5.5	Branch ownership	24
5.6	Voting	25
5.7	The architecture in detail	27
6	Evaluation	30
6.1	Evaluation Method	30
6.2	The Git project	30
6.3	Misuse cases in the Git project	32
6.3.1	Malicious maintainer	32
6.3.2	Malicious developer	32
6.3.3	Misuse cases	32
6.4	Misuse cases in decentralized Git	33
6.5	Vulnerabilities in a centralized Git web service	34
6.6	Role based access control	35
6.7	Decentralized Git security	35
6.8	Voting	35
7	Discussion	36
7.1	Design choices	36
7.2	RBAC	37
7.3	Consensus	37
7.4	Security	37
7.5	The work in a wider context	38
8	Conclusion	39
8.1	Aim and research questions	39
8.1.1	What security threats in Git web services can be mitigated using a distributed DAG ledger?	39
8.1.2	What security threats might occur when introducing a distributed DAG ledger to a decentralized Git implementation?	39
8.2	Future Work	40
	Bibliography	41

List of Figures

2.1	Git branching model	6
2.2	Git commit structure	7
2.3	Merkle tree	8
2.4	Directed acyclic graph	10
4.1	System overview	16
5.1	Commit hash structure	21
5.2	Git tree structure before and after merge	22
5.3	Ledger database structure	23
5.4	P2P network	24
5.5	Abstract view of ownership of branches	25
5.6	Flow of messages in the voting algorithm	26
5.7	Overview of node functionality	28
5.8	Flowchart of messages sent between peers	29
6.1	Overarching misuse case	33
6.2	Misuse case diagram of decentralized Git	34

List of Tables

4.1	Evaluation for DAG based distributed ledgers	17
4.2	Evaluation for git implementation	18
6.1	Use cases for the Gitlab web service	31
6.2	Malicious agents in the Gitlab web service	31
6.3	Narrowed down use cases for the Gitlab web service	31

Glossary

Asyncio - Asyncio is a python toolkit for developing concurrent single-threaded code.

P2P - Peer to peer network where peers share information between each other.

RocksDB - A key-value pair database implementation developed by Facebook.

DLT - Distributed Ledger Technology.

RBAC - Role Based Access Control.

PBFT - Practical Byzantine Fault Tolerance.

DAG - Directed Acyclic Graph.

Byzantine fault - Arbitrary fault.



1 Introduction

The introduction of blockchain technology by Satoshi Nakamoto[1] sparked an era of cryptocurrencies and blockchain applications. The technology was primarily developed to enable people to trade without the need for a centralized banking system, even so, this technology is not limited to developing currencies. There are numerous uses for the blockchain, one example of an application is the possibility of storing medical records on the blockchain[2]. The security benefits of a blockchain could be useful in numerous use cases, one of these is the version control system Git. To a Git project, availability and integrity might be significant factors, where the owner of the project would prefer to decentralize it to prevent breaches in the central system. In this scenario, a distributed ledger system could be a solution, and therefore it is of interest to observe if Git functionality can be modified so that it can be hosted on a distributed ledger. New advancements are made continuously to improve upon the first implementation of the blockchain. One of these advancements is the concept of a distributed Directed Acyclic Graphs(DAG) based ledgers[3], where blocks on the ledger can be reference multiple blocks instead of the conventional single chain where a block only references to the previous one. The DAG ledgers has structural similarities to Git's branching models and could therefore represent git commits and their branches in a much more logical way compared to conventional blockchains. This thesis will therefore research whether it is possible to implement a decentralized Git version control system with the help of a distributed DAG ledger and evaluate the security trade-offs of doing so.

1.1 Motivation

One of the main reasons behind the rise of cryptocurrency was the possibility a currency not regulated by a central bank. The users themselves could maintain the currency without the need for a centralized authority. Git web services such as Github and Gitlab, much like a centralized bank, need the trust of the users for them to consider storing their code on their platforms. These Git web services have security functionalities such as login credentials and levels of privileges for the participants in the project to help insure the integrity of the project. However, there are still cases of breaches in these Git web services.

In 2020 a popular chrome extension was exposed to a malicious maintainer. The original owner to the project stepped down as maintainer and gave the role to a unknown party[4]. A new malicious version of the software was published to the chrome extension store where all users that updated to the new version were now compromised. This incident did not only affect the integrity of the Git source code but also the integrity of the users information.

Security breaches may not always occur due to bad practice from individuals using Git. Since Git is hosted on centralized servers, attackers may get access due to security flaws on the servers or web interfaces. Such a flaw was discovered on the popular git hosting service GitLab[5], where due to the lacking image validation on the website, arbitrary code execution could be preformed.

The examples mentioned are some of the various cases of security flaws that can result in massive losses for its users. The usage of Distributed Ledger Technologies(DLT) in Git could mitigate attacks because of the lack of centralized servers that host the repositories, by decentralizing and distributing them. Furthermore, by accounting for the immutability property of DLT:s[6], attempts of data modification can be discovered and can help validate the integrity of the source code.

Hence, these above mentioned incidents motivates us to propose a solution that intendeds to introduce a way to mitigate breaches in the centralized servers and introduce new mitigation strategies against malicious participants of Git project repositories, using distributed DAG ledgers.

1.2 Aim

The aim of this thesis is twofold, first, we want to implement a trustless implementation of Git using distributed DAG ledger technology, and then to evaluate how this implementation can mitigate security risks in the conventional git implementation. The thesis will focus on a particular misuse case where a malicious agent gains access to a git project and how a Git implementation based on DLT could help mitigate it. The thesis will also evaluate how a decentralized git solution could mitigate common vulnerabilities that exist in Git web services.

1.3 Research questions

This thesis will focus on the following questions:

1. What security threats in Git web services can be mitigated using a distributed DAG ledger?
2. What security threats might occur when introducing a distributed DAG ledger to a decentralized Git implementation?

1.4 Approach

Firstly, to answer the research questions, a architecture for a decentralized Git is proposed. Secondly, the decentralized and conventional versions of Git are compared and evaluated

with the help of use and misuse cases, as well as common web vulnerabilities. The thesis will also evaluate new possible threats that might occur when using the decentralized version as described in the thesis.

1.5 Delimitations

This project will focus on implementing a minimal implementation of Git on a distributed DAG ledger, where some functionality of Git will not be implemented. Similarly, a simple implementation of a DAG DLT might not include all features usually found in DLT:s that would otherwise ensure more robust security as well as other properties. This is due to the scope of the thesis.



2 Background

The following chapter elaborates on the topic of this thesis and supplements the information not yet covered.

2.1 Git

Git is a Software Configuration Management (SCM) tool and is used for version management in various projects. Git uses branching and merging features to allow developers to simultaneously contribute to the same project. This allows developers to create branches for specific parts of the project to be worked on, without tampering with the overall project[7]. The Git project is separated into a local and a remote repository. The user commits their changes to the local repository which is stored on their machine and when the user is ready to share their changes, they push them to the remote repository. The remote repository being stored on a different server or somewhere on the internet[8].

2.1.1 Git Branching Model

In software development there is a need for a structured workflow and versioning of the software. Git is a versatile version control system that can be used in different ways. A popular way of using Git for versioning is to have long running branches. These branches are used to continuously merge into and represent a stable version of the source code. Most often large scale project will have a master branch representing a stable version of the source code ready for release. A develop branch is also very common to use as a long running branch that is used for development. To contribute to the source code a developer branches from the develop branch. These branches are called topic branches or more commonly feature branches. These branches are short lived, since when the feature is implemented and merged into the develop branch the feature branch can be discarded. When the develop branch is

stable and has the features needed for the next version of the source code it can be merged into the master branch[8].

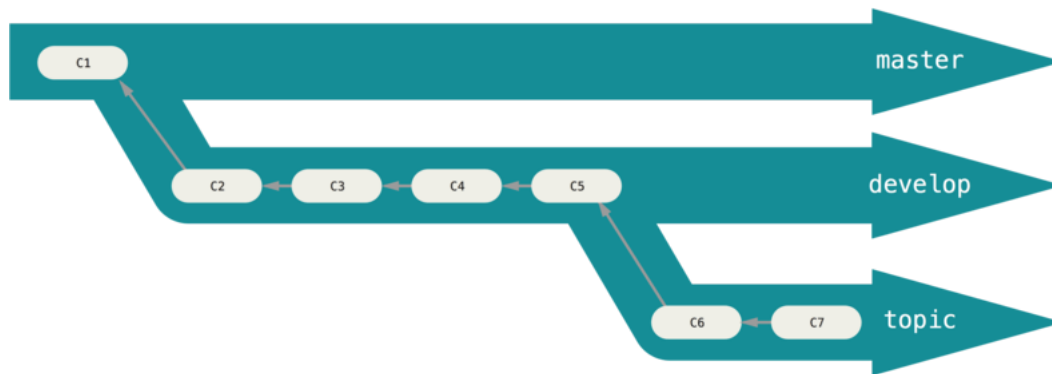


Figure 2.1: Git branching model, chapter 3.4 [8]

2.1.2 Git Internals

The Git repository consists of certain objects called Git objects and their relations to one another enables the overall git functionality. The objects are the following:

- The blob
- The tree
- The commit
- The tag

Git content, such as files, are stored as blobs in the Git repository. Whenever a change is committed, only the blobs that have been changed are included, files that are not changes are stored as references to past objects in previous commits. The blob is hashed using SHA-1 and is used to create tree hashes, in a similar fashion to a merkle tree. It is these trees that work as pointers to newly changed blobs or past unaltered blobs. Furthermore, the commit object tracks the progression of the project. In order to gauge the progression of the project the commit also points to a parent commit, which is the commit before it. In the case of merged branches, the next commit points to the branches it merges. The tag object associates metadata to a specific commit, for instance the name of the person that made the commit and the commit message[8]. Figure 2.2 illustrates an example of a series of commits and their relations to the trees and blobs.

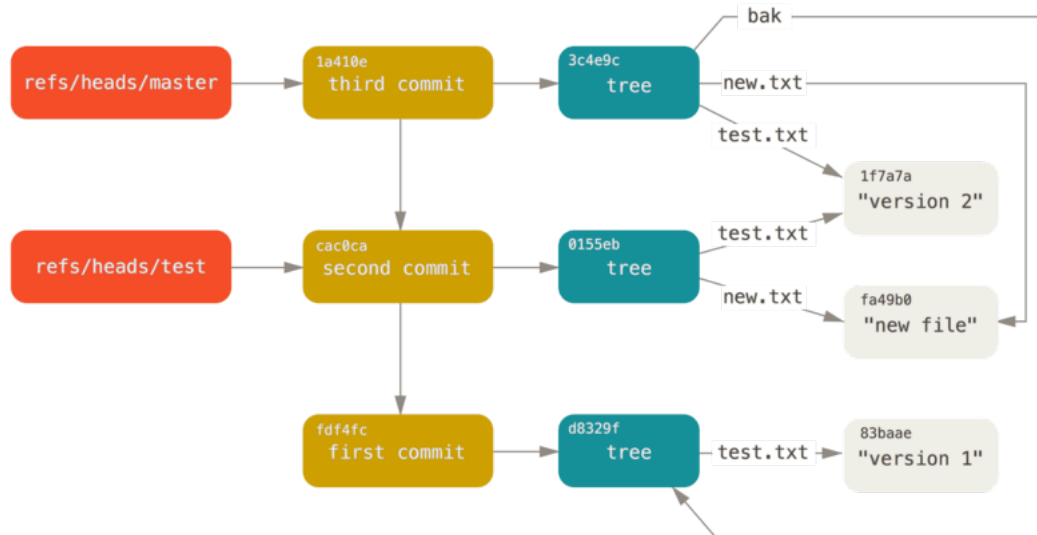


Figure 2.2: Git commit structure, chapter 10.3[8]

2.1.3 Permission Structure

The most commonly used git web services, such as gitlab, use Role Based Access Control(RBAC) to manage the access rights of different users. RBAC allows git web services to define different roles that have varying levels of permissions, be it read or write. Depending on what git web service, the roles defined can vary[9].

2.2 Cryptography

Cryptography is vastly used in computer science to ensure security for data. This is achieved using various methods and mathematical algorithms to create ciphers by encrypting plain and exposed data. This subsection will contain cryptographic concepts used in this project.

2.2.1 Public-key Cryptography

Cryptographic keys are specific data objects that are used to cipher or decipher data that is in need of confidentiality, just like the way a physical lock would need keys to lock and unlock access to whatever is locked. They can also be used to identify specific individuals in the system based on their unique value.

Public-key cryptography that is also known as asymmetric encryption is the concept of using unique key pairs for each sender/receiver instead of having one universal public/private key pair between participants, i.e using the same public and private keys to encrypt and decrypt. Instead each participant generate a key pair of their own and makes their public key known to all. In message encryption, if A then wants to send to B, then A will encrypt the message using B's public key, to which only B can decrypt using their private key. This strengthens confidentiality[10].

2.2.2 Digital Signatures

Digital Signatures (DS) ensure integrity with the use of public/private key pair, and are used to prevent fraudulent messages. DS consist of public/private key generation as well as signature and verification algorithms. First the sender signs the message using their own private key. The signature can then be sent and verified by others using the public key corresponding to that sender, which is known to the recipients. An example of a DS implementation is the RSA algorithm which is a cryptographical method of encrypting messages and signing signatures by using large prime numbers[10].

2.2.3 Hash Functions

Hash functions are cryptographic functions that work like one-way functions if implemented properly. This property makes it hard for anyone to deduce the input(raw data) based on the function and the output, the output being the cryptographic hash. This makes it a useful tool for storing sensitive of sensitive data, for example by hashing passwords before storing in a database. The Secure Hash Algorithm(SHA) is an example of a hash functions, where SHA-1 generate a 160-bit hash value[10].

2.3 Merkle Trees

The merkle tree is a data structure that use cryptographic hashes to identify nodes. The leaf nodes on the merkle tree containing data that are then recursively hashed further up the tree until we have a root node that is hashed using all the nodes and leafs bellow. This is illustrated in figure 2.3. This way the hash depends on the data stored in the leaf nodes. If the data is altered on the leaf nodes the change will cascade up to the root node's hash. Blockchain implementations commonly use this method to store data on the blocks, where each transaction on a block is represented as a leaf node. Git also uses similar merkle trees to represent the changes made on the repository[11][8].

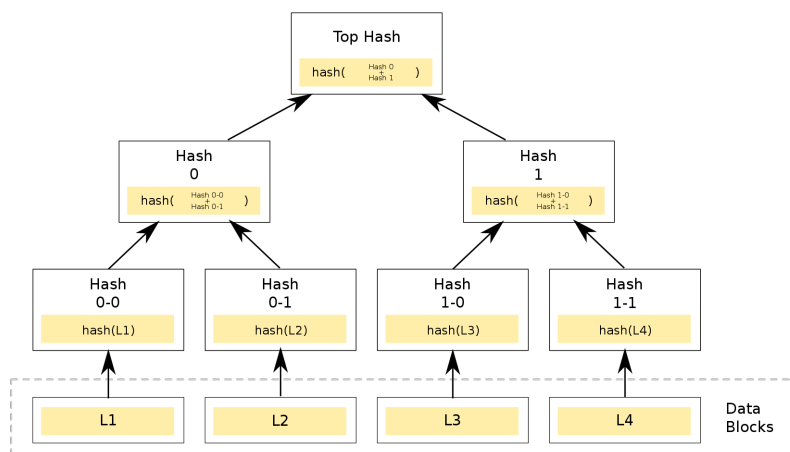


Figure 2.3: Merkle tree[12]

2.4 Blockchain

Blockchain can mostly be described as a distributed ledger that stores blocks. These blocks contain data, which in the case of cryptocurrency would be transactions. The blocks also contain metadata to identify and connect the blocks together.

The blocks are identified using a hash generated by a hash function. These hashes identify the blocks and are used to connect the blocks in a reverse linked-list. The last block references the previous block and all the way down to the first block, also referred to as the genesis block. Tampering with any of the blocks in the blockchain will affect the hashes in the whole chain leading to an avalanche effect[11].

2.4.1 Consensus algorithms

As in all distributed systems, the blockchain can suffer from byzantine faults, which means some of the participants in the blockchain might be acting in an arbitrary way or even maliciously[13]. The way the blockchain solves the issue of consensus in the blockchain depends on the implementation. The bitcoin blockchain for example uses Proof of Work(PoW) to achieve consensus in the network. PoW depends on "miners" to generate new blocks. They do this by generating a hash for the block. However, to generate a hash is trivial for modern systems. One way to make the finding of hashes non-trivial is to make the miners find a hash starting with some arbitrary amount of zeros. This way we can also control the difficulty of the mining by increasing or decreasing the number of zeros. Now if an attacker tries to take over the blockchain they have to outperform the rest of the network to gain control. This is because only the longest chain will be accepted by the network[6].

Practical byzantine fault tolerance (PBFT) is a protocol for reaching consensus, introduced by Castro et al. [13]. For the network to be seen as non faulty the number of malicious nodes in the network need to be limited. Maximum number of faulty nodes are described as $3f < n$, where f is the number of faulty nodes and n is the size of the network[14]. The algorithm relies heavily on sending messages back and forth between nodes.

2.4.2 Distributed DAG Ledger

A blockchain can be seen as a long continuous linked chain. However, new innovative ways of designing DLT have had success. One of these are the distributed directed acyclic graph ledger, where the ledger is designed as a directed acyclic graph as shown in the figure below.

In terms of data structures, instead of having individual blocks with list of transactions in it, a DAG ledger commonly has nodes that represent single transactions where a node can be referenced by multiple nodes[3].

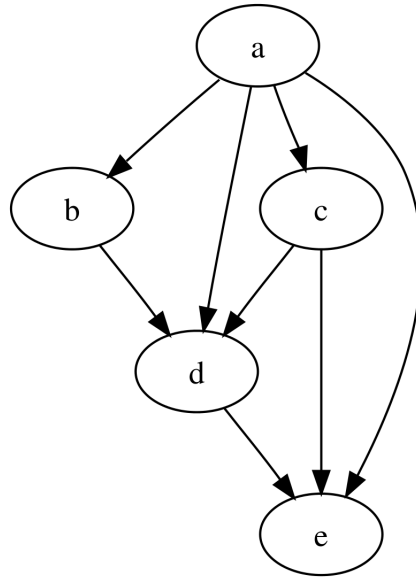


Figure 2.4: Directed acyclic graph[15]

2.5 DLT Vulnerabilities

DLT brings security to a system but there still exist vulnerabilities that may be exploited by attackers. There are various forms of attacks that could compromise a DLT system. A known attack to note is the sybil attacks.

2.5.1 Sybil Attacks

The sybil attack is when an attacker can create multiple identities on a network and by doing so gaining a majority. In the case of a blockchain a sybil attack can allow attackers to rewrite transaction history and even isolate honest participants by not relaying their transactions.[16]. PoW is widely used by cryptocurrencies such as ethereum and bitcoin. They are difficult to compromise through sybil attacks because of the number of miners mining their currency continuously. For an attacker to gain access to the network they would have to out perform all the miners on the network. The 51% attack is a well known attack within blockchains where malicious actors gain the majority of the computational power. This majority is used to verify blocks and place them on the blockchain. The majority vote enables them to agree upon malicious transactions and alteration of transaction history[16].

2.6 Threat Modeling

Threat modeling is a method of identifying threats in a system. With threat modeling, the process of identifying security risks is structurally implemented in the work flow. There are various approaches in doing so, where vulnerable components can be identified and classified according to severity. Also, possible attacks and attackers may be considered in the threat modeling process[17].

2.6.1 Misuse Cases

An example of a procedure that can be used in threat modeling is misuse cases. A misuse case is essentially an analysis of a bad scenario. These may be created for specific scenarios to identify and analyze vulnerabilities in usage of the system. It can provide a good understanding of what is needed to do for mitigation of said threats and can be represented in a diagram that visually show the dependencies and threats/mitigations of the scenario.[18].



3 Related works

The blockchain application is not a new concept in distributed technology, however there are still possibilities for re-imagining how a blockchain can be implemented. Since Nakamotos initial whitepaper, there have been plenty of research done in the field of blockchain applications. This section will present some of the research that relates to this paper.

3.1 Distributed ledger technologies

A paper by W. Yang et al.[19] describes the implementation of a lightweight DAG-based blockchain for vehicular social networks. The paper proposes the DAG-based blockchain network as a solution for security and efficiency for the Vehicle Social Network (VSN). This DAG-based blockchain, called LDV in the paper, helps the VSN ensure security through PoW but also enables the vehicles to dismiss information that is of no concern to the vehicle. This way a vehicle only needs to store part of the blockchain and makes the LDV much more storage efficient compared to bitcoin for example that demands that each node store the complete blockchain, which is currently 350 GB[20].

Y. Hashem, E. Zildzic, and A. Gurtov[21] propose a solution for secure location updates in a Unmanned Aircraft System (UAS) network. The paper describes using the blockchain framework Hyperledger Iroha, a permissioned blockchain, to store data specified by the Drone Remote Identification Protocol (DRIP). The solution is evaluated by simulating a network of drones, where each simulated drones sends updates to the blockchain and calculating response times. The paper concludes, given the number of drones and Iroha nodes, that the response time is acceptable for real time location updates.

Ding et al.[22] propose a decentralized database platform using a DAG DLT, called Dagbase, with the purpose of ensuring a higher level of security but also maintaining high efficiency. The proposed construction divides the product into three layers, where the first layer ensures a DAG structure of the database events. The second layer would be the database functionality, and the last layer provides the user interface. Similarly to other typical DAG based DLTs the

events that are represented as nodes point towards two parent nodes. Ding et.al use a DAG consensus in accordance to HashGraph that ensures Byzantine fault tolerance.

3.2 Version control on the blockchain

In a paper by S. Wang et al.[23], they describe a forkable storage engine called ForkBase and the applications that can be developed on top of this engine. The ForkBase engine has implemented features widely used in version control systems, for example merging, forking and versioning. These features make it simple to implement a version control system like git on top of the engine. In the paper they describe the implementation of a Hyperledger blockchain that uses ForkBase to store data. This blockchain was implemented to stress test the storage and compare it to the conventional way of storing data on a Hyperledger, which is using RockDB. The results were mixed however one could see a great improvement in latency when performing write operations.

A solution to data storage on the blockchain was proposed by Rouhani et al.[2], where an off-chain storage was used alongside the blockchain. The solution was applied to medical data storage, which naturally needed a level of security and integrity. The blockchain was implemented using the framework Hyperledger. Hashed URI:s of the medical records were stored on the blockchain, which was significantly less data than the medical record itself. This solution was based on permission, where each participant had to be known to the system, and with permission the user could access the URI from the blockchain and fetch data from the off-chain storage.

Nizamuddin et al.[24] introduce a distributed solution to file sharing and version control. This is done by utilizing the scripting language Solidity for writing smart contracts on the Ethereum blockchain. The system is designed to store the data off-chain by using the distributed file system IPFS(InterPlanetary File system). A smart contract is created in the Ethereum blockchain whenever a new document is uploaded. Then, developers may upload a new version of said document in the IPFS and request its approval to the blockchain. It is stored if at least two thirds of the approvers accept the request. The registration of developers and approvers follow the same method of approval.

3.3 Threat modeling

Karpati et al.[25] presented a comparison of Attack trees(AT) and Misuse cases(MUC). The goal was to discover which of the two methods, in an industrial setting, yield more findings in threats and mitigations, but also what type of threats and mitigations. By having software developers as participants, they were introduced to a software system of a familiar domain to them(the banking system), and the usage of both methods. The participants' work experience and knowledge were also taken into consideration while conducting these tests. The overall findings showed that AT yielded substantially more threat discoveries while MUC yielded slightly more mitigations. The type of threats that were discovered using these methods were connected to different parts of the software security life cycle, where MUC threats in the design and requirements phases and AT in the implementation and usage.

Ten et al.[26] propose a methodology for assessing vulnerabilities in cybersecurity for SCADA systems, using attack trees. By taking into account three different conditions for each attack scenario that is represented by a leaf in the attack tree, a variable between 0 to 1 is acquired.


the number 0 being invulnerable and 1 vulnerable. The conditions for each attack scenario are: there have been no attempt to intrude, at least one countermeasure has been implemented for each attack scenario and there exist at least one password policy for each attack scenario. All scenarios of the tree contribute to indices, and by evaluating them, the pivotal leafs with vulnerabilities can be determined. By evaluating all leaves, one can identify which leaves need further countermeasures based on their values.

A study conducted by Petrica et al.[27] illustrates the vulnerabilities of a WordPress web server using attack trees. Each leaf node in the attack tree corresponds to a exploit in the web server and has a probability assigned to it. By assigning probabilities for each exploit one can quantify the level of security in a WordPress web server by deriving the probability of each exploit and then calculate the probability for an attacker to successfully compromise the web server.

3.4 Motivation

The implementation described in the thesis is a novel approach to version control systems. Because of this, earlier works describing similar endeavours are limited. The related works described above can help guide the thesis in how the decentralized version control system can be implemented, analyzed and discussed. This is why the main focus of the related works are in the areas of distributed ledger, threat modeling and version control.

Analyzing what ways version control systems have been used in a decentralized Git, but with a difference in taking the DAG structure into account. One key element of storing large amounts of data on a blockchain is the off-chain storage as described by Rouhani et al[2]. The area regarding distributed ledgers helped shape an understanding of practical application of DLT. These were not particularly majorly inspired to the thesis in terms of application, but gave an understanding of how wide the use is for DLT and different approaches of implementing these applications. The topic of security is a central part of DLT and naturally the security of any DLT application should be researched. Threat modeling is a appropriate tool when researching any vulnerabilities in a system, Karpati et al[25] gives insight into how powerful threat modeling is as a tool for finding vulnerabilities and threats.



4 Design choices

The purpose of this section is to describe the general system design and the approaches that can be used to implement the system architecture. The method used to evaluate what approaches are best suited for the thesis is defined in section 4.2.

4.1 System design

The purpose of the system is to distribute the resources that would have otherwise been centralized on a Git web service. The remote repository, that is conventionally stored on a centralized server solution, will be distributed amongst all participants of the Git project, represented as peers in the P2P network. The peers will be able to commit their changes to their local repository which could then be pushed to the distributed remote repository. To ensure the integrity of the commit history and the commits data, the commits and their connection to each other will be stored in the distributed DAG ledger. The ledger will contain meta data for each commit and a reference to the remote repository file system. Figure 4.1 illustrates how a peer communicates with the network and what parts of the system are distributed and what part are local to the peer.

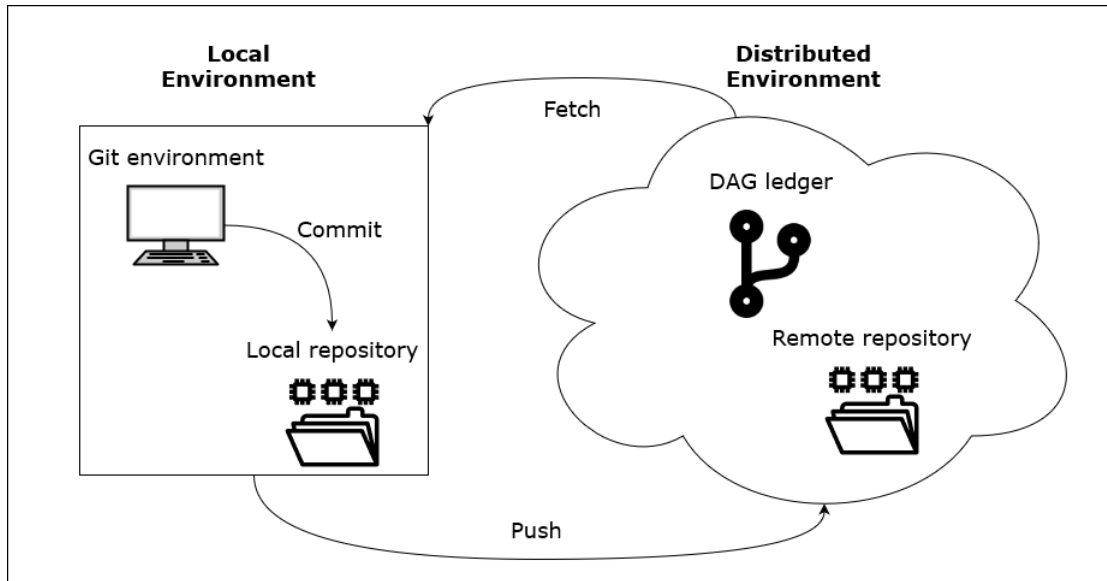


Figure 4.1: System overview

The general design of the system consists of sub-systems, where each sub-system is responsible for a different part of the architecture. The sub-systems in questions are listed below.

The DAG ledger is responsible for ensuring the integrity of the source code and to maintain the branching structure of the Git version control system. The distributed DAG ledger, much like the blockchain, needs to be immutable, decentralized and secure. In the system design the distributed DAG ledger will contain blocks, where each block on it represents a commit. **The Git environment** is responsible for the versioning of our source code and contains logic behind how commits are connected with each other. When a developer has made a commit and decides to push this commit, the peer then sends out this commit to the rest of the network. On a push the Git implementation will send the commit through the network and will update the distributed storage and DAG ledger. On a fetch the peer will update their local repository with the changes made by other peers. **The remote repository** will be hosted on a distributed data storage and will contain all git object and the commit tree structure. The remote repository is referenced by the distributed DAG ledger.

The P2P Network allows for a decentralized Git web service. The network is used to allow peers in the network to send their updated versions of the source code to each other and update the distributed Git environment.

4.2 Design Considerations

The system can be implemented in numerous way, therefore this chapter will evaluate which course of action is appropriate for this project. The categories used to evaluate the options for the different parts of the system are,

Documentation: How well documented is the implementation?

Compatibility: Is this technology compatible with the branching structure of Git?

Time estimation: Approximately how much effort is needed to start using this technology?

Network: Is there an existing peer to peer network implementation that can be used?

4.2.1 Distributed DAG ledger

The idea of DAG based ledgers is a relatively new concept in blockchain technology. This means that there is a limited community developing DAG based ledger solutions, where conventional blockchain technologies are already being used for industrial purposes. The distributed DAG ledger implementations that are to be considered for this project are the tangle and obyte. The option to develop a light-weight customized DAG ledger is also to be considered, where only necessary functionalities will be implemented to host Git.

The obyte, previously known as byteball, is a DAG based distributed ledger implementation where arbitrary data can be stored. The obyte has a relatively small community of developers and users compared to other similar blockchain implementations. When new blocks are introduced to the network they are validated by other users by referencing the block in their block. To incentivize validating blocks, the obyte introduces an internal currency named bytes. To publish a block on the obyte the publisher needs to pay a fee in bytes that are paid to the users that validate the block[28].

The tangle is a DAG based distributed ledger used by the IOTA cryptocurrency designed for the Internet of Things(IoT) industry. The tangle contains a DAG ledger made up of individual transactions instead of blocks containing multiple transaction like the blockchain. Each transaction in the tangle needs to validate two previous transactions for it to be admitted[29].

Table 4.1: Evaluation for DAG based distributed ledgers

DAG implementation	Documentation	Compatibility	Time estimation	Network
obyte	Minimal	Yes	Substantial	Yes
tangle	Well documented	No	Acceptable	Yes
light-weight DAG	None	Yes	Acceptable	No

As shown in the evaluation of the different DAG ledger technologies in table 4.1, the tangle can be immediately disregarded because of its incompatibility with the Git branching structure. The tangle requires that each new entry needs to validate two past entries which does not fit with Git branching structure since a commit should be able to only reference one past commit. The obyte does not have this issue. However, it is not as well documented and not as widely used for DLT applications compared to the tangle.

Implementing a light-weight DAG ledger requires implementing the minimal required functionalities for the ledger to still have the properties associated with DLT. These properties are already present in the obyte and the tangle. To distribute the DAG ledger the system needs a P2P network where all peers in the network share this DAG ledger. Both the tangle and obyte have ready to use implementation for a P2P network, however for the light-weight DAG there is no network to take advantage of. The proposed solution for the P2P network is to have an authentication server that handles the peers' connection to the network. When a peer connects to the authentication server it connects the peer to the rest of the network.

4.2.2 Git

Since Git is unique with its features, the design choices are limited to the existing open source Git implementation and developing lightweight Git implementation. The Git source code is open source project which has a large community of developers and provides the opportunity to fork the repository for development. This choice allows for tailored implementation for using it with a DAG ledger technology. However, in order to do so, one would have to properly understand the source code and adapt it for the usage of the DAG, this could be time consuming because of the scope and complexity of the Git source code. The light-weight Git implementation provides the opportunity to only develop features that are significant to the system, and can be tailored towards the DAG ledger functionality.

Table 4.2: Evaluation for git implementation

Git implementation	Documentation	Time estimation
Git	Well documented	Substantial
light-weight Git	None	Acceptable

4.2.3 Distributed DAG ledger permissions

With the introduction of a peer to peer version control system, the issue of access control needs to be addressed. Therefore a notion of ownership of branches will be introduced as a way of protecting the source code under development and to help enforce a structured way of developing software. The idea for ownership for branches is already a concept in software development, where each feature branch is "owned" by the developer developing that feature. The long running branches such as a develop branch and a master branch could potentially be seen as "ownerless" branches, where pushing commits to this branch requires some form of voting algorithm.

4.2.4 Consensus

One could speculate on what type of consensus algorithm would be appropriate for the decentralized Git described in the thesis. Given a Git project, where the number of participants can vary greatly, consensus algorithms such as PoW can be ineffective. Since the number of participants can be few in number and the number of entries to the DAG ledger are few and far in between. The aim of decentralizing Git was to possibly mitigate the risk of malicious actors. Proof of Stake(PoS) consensus elevates specific actors' influence on validation based on their stake[30], and thus using this consensus in an ordinary blockchain for Git might not mitigate this risk. DAG based consensus algorithms, such as the one used in the tangle, uses the DAG structure when reaching consensus. For example, the tangle requires that a new transaction validates two past transactions This is not suited for the context of a git branching structure. Since the DAG structure is a result of the consensus algorithm of the DAG based ledger, it is difficult to maintain the commit structure of Git and still use the DAG based consensus algorithm.

Possible consensus algorithm that could be effective for the DAG ledger are those similar to practical byzantine fault tolerance. PBFT is a effective consensus algorithm in a distributed system with a limited number of node. However, as the number of nodes increase the performance severely deteriorates[14]. However, in the context of a Git repository the user needs

to actively participate in the voting to accept or deny incoming commits to the protected branches. This voting algorithm can have its communication flow based on PBFT for a fair voting process.

4.3 System architecture

After evaluating the different options for the system implementations, the following sub-systems are chosen for the system architecture:

- Light-weight Git implementation
- Light-weight DAG implementation
- P2P network implementation using a gateway server
- Voting based consensus algorithm



5 System architecture

As described in section 4.1 the system consists of different sub-systems. This chapter will describe what choices are made in the implementation of these sub-systems.

5.1 System overview

The core components of the system are the DAG ledger, Git implementation and the network. The DAG ledger implemented is a light-weight custom DAG implementation, distributed using a rudimentary P2P network. The Git implementation is a light-weight implementation and made to integrate with the P2P network.

The part of the system that a user directly interacts with is the light-weight Git. It functions just as conventional Git, where the user can commit changes, branch from these changes and merge branches together. The major difference between the light-weight Git and Git is the remote repository. In Git the remote repository would be located on a centralized server or a web service such as Gitlab or Github. The light-weight Git distributes the remote repository between its peers where every peer in the network has a copy of the remote repository on their local machine. This repository is kept consistent between the peers. The user commits to the local repository and pushes these commits to the remote repository. These commit are sent over the network where all peers update their remote repository. Aside from updating all the remotes, the DAG ledger is also updated during a push. The user will append all commit metadata to the DAG ledger and when all the peers in the network receive the commits they will do the same.

5.2 Light-weight Git

The light-weight Git is a minimalist Git implementation written in python, where only important features are implemented to establish the overall functionality of the version control

system. Features such as compression of commits and objects are not implemented in light-weight Git. The implementation focuses on handling blobs, trees and commits similarly to Git where each Git object is hashed using SHA1. Each object hash that is not a blob is hashed using the objects that are connected to it, following the same structure as Git internals[8]. Similarly to a merkle tree, a hierarchy of hashes is created with the commit object hash as the root as seen in figure 5.1.

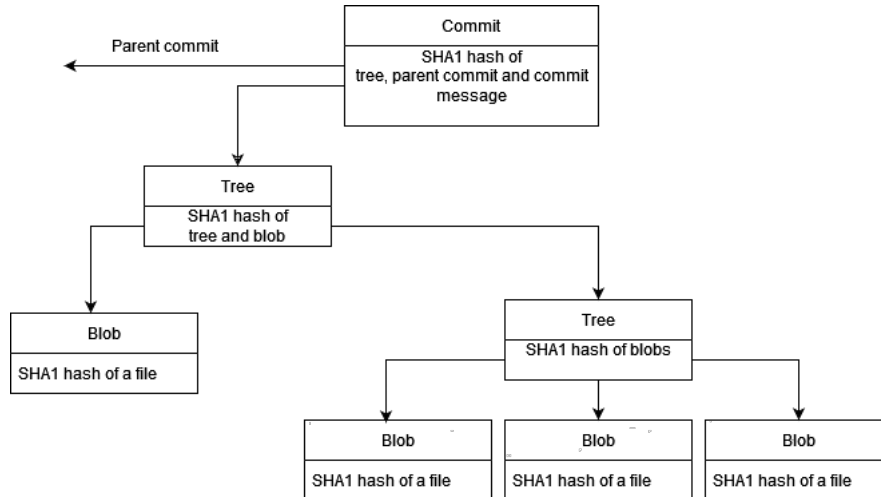


Figure 5.1: Commit hash structure

The light-weight git implementation has most Git commands available, such as init, status, log, commit etc. The Git commands of note that are implemented are branch and merge. In order to differentiate different branches from each other, branch references are used to point to that specific branches latest commit. Another usage for references is the HEAD reference, that points to the branch reference that the user is currently working on. Using the "checkout" command allows the user to change this reference to point to another branch reference. This can be seen in figure 5.2.

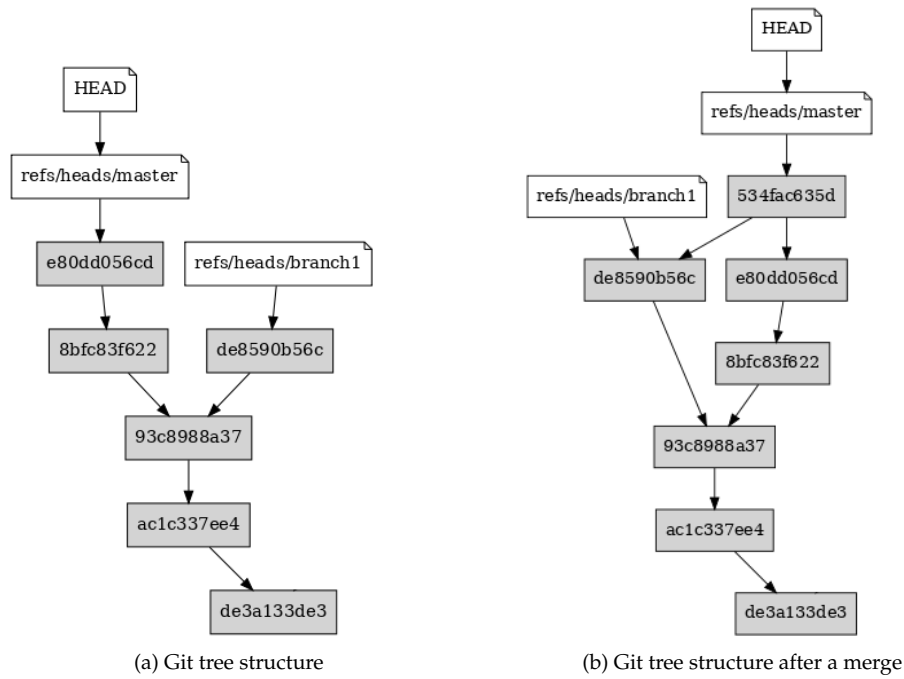


Figure 5.2: Git tree structure before and after merge

The merging of two branches is accomplished using the the merge command. By invoking it, the data that is stored in one of the branches is loaded into the current working directory. A commit is then created that points to the latest commits of the branches as its parent. This is illustrated in figure 5.2(b). Now the changes made in branch1 will be present in the master branch.

5.3 Distributed DAG ledger

The ledger is distributed on the network as a rocksdb database, a key-value pair database implementation developed by Facebook[31], where all the block data is stored. As seen in figure 5.3 each entry contain the block hash as well as the hashes of the parents. For the DAG structure to be possible, a block needs to be able to reference multiple parents. The block is separated into two layers where the first layer contains the commit metadata and the second layer contains the DAG block.

The DAG block contains all information needed to maintain the integrity of the DAG ledger. The DAG block hash is used to identify the block and is calculated by hashing the entire block using the hash function SHA256. To identify the peer that published a block, a public key is used as the user ID. The peer generates a public key and a private key using RSA key generation. These keys are later used in the digital signatures by signing blocks that are to be sent and verifying blocks that are received. The branch stored in the DAG block identifies which branch the commit is a part of. This branch relates to the same branching structure seen in Git. This way ownership of branches can be enforced. Using the public key of the peer and the branch reference in the block, the network can be sure that the only one that is able to publish to a branch is the peer that created the first reference to the branch.

The commit metadata contains a merkle root, described in the DAG block as commit data, that references a file that exists in the remote repository. This data is received via the light-weight Git remote push, where the peer stores the data on a remote repository located in the peers file system. The commit parents are stored to help dictate the branching structure of the DAG ledger, where connection between DAG blocks are the same as the connection between commits. This means that a DAG block containing a commit will reference the DAG blocks of that commits parents.

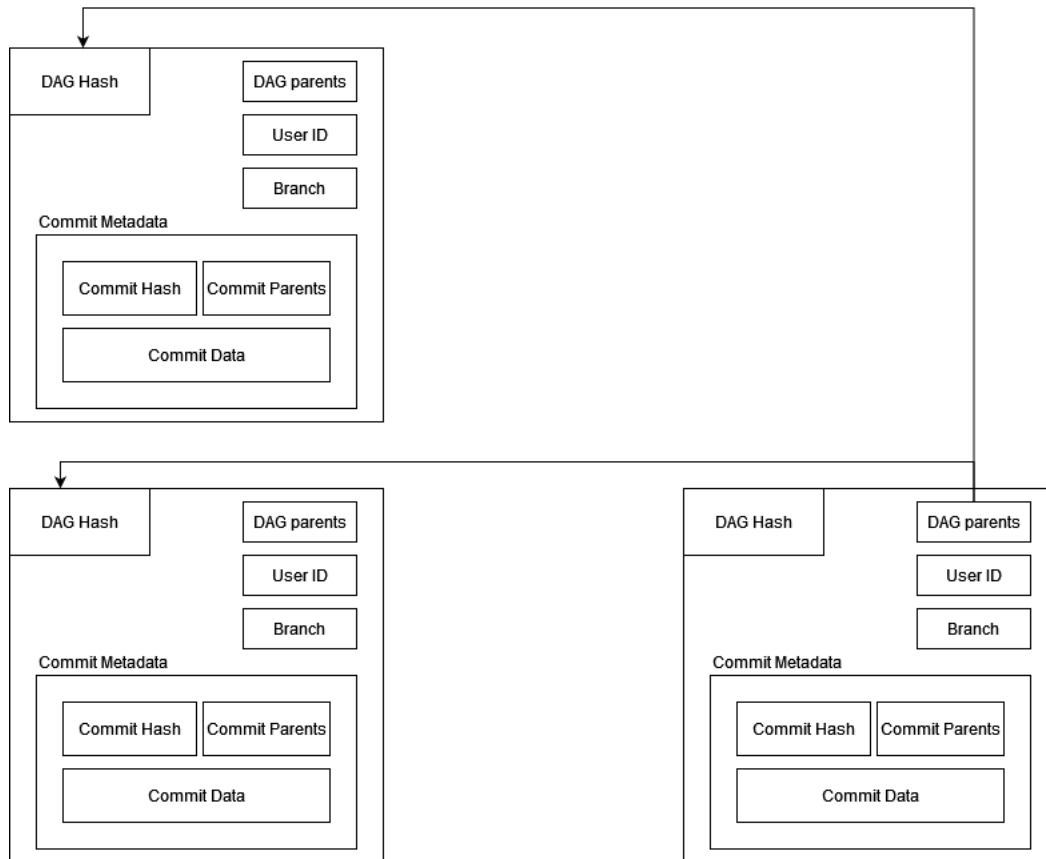


Figure 5.3: Ledger database structure

5.4 P2P network

To host the distributed DAG ledger a P2P network is introduced. The network is represented as nodes where each node is connected to all other nodes in the network. To connect to the network, a gateway server would connect a node by sending a list of peers to the new node and updating all the others on the network about the arrival of the new peer. An example of such a network is illustrated in figure 5.4, where one gateway server is responsible for connecting an entire network of an arbitrary number of nodes. When a new peer connects to the gateway server, it sends a the public key of the new peer and its address which the peers would use to connect to it.

The private key is used by the peer to sign any blocks that the peer would like to append to the DAG ledger. Since the public key is known to the rest of the network the peers can verify

that the block truly came from that specific peer. The public key is also stored on the ledger to identify the user.

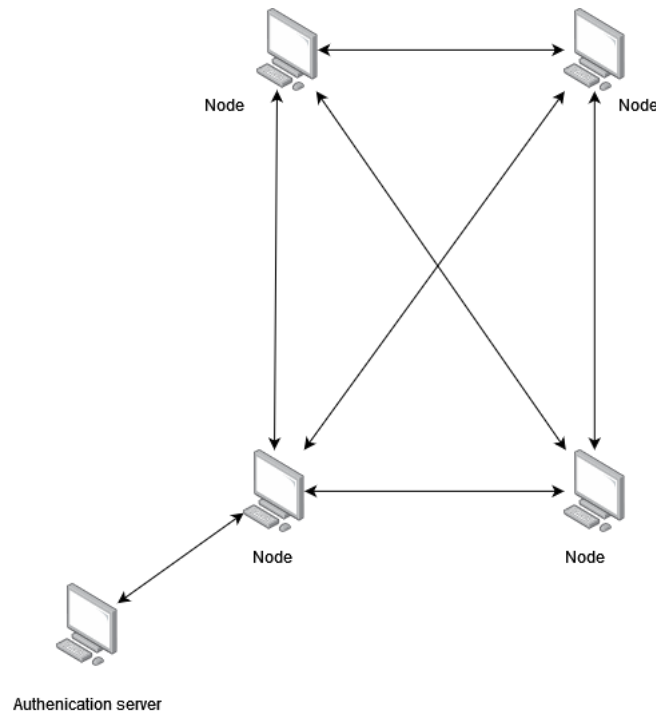


Figure 5.4: P2P network

5.5 Branch ownership

To be able to implement a notion of branch ownership, user ID and branch reference are introduced to the DAG ledger. The user ID identifies who has made the commit with the help of the users public key and the branch reference identifies what branch the commit is a part of. The user can add their branch to the DAG ledger by creating a branch reference, when the reference is added to the DAG ledger there is a clear history of who created the branch and therefore who owns it. Using the public keys to identify the user, allows the rest of the network to verify who appended the block using RSA key signatures. This way it is harder to impersonate a user unless they have their private key.

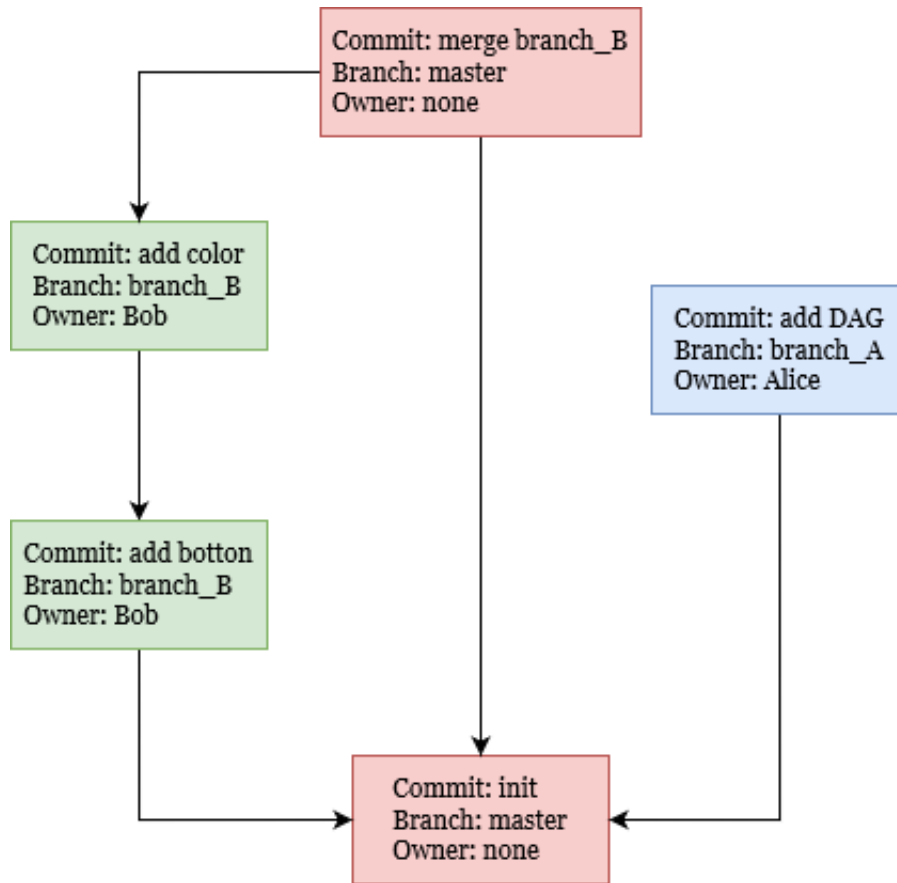


Figure 5.5: Abstract view of ownership of branches

Figure 5.5 illustrates how Bob and Alice commit their changes to the distributed DAG ledger. Alice has made her own branch and makes a commit that she pushes to the rest of the network. Since Alice is signed as the owner of that branch Bob can not commit to the same branch. Bob makes his own branch and makes two commits and pushes them to the network. Bob then makes a merge request to the master branch which is accepted by the network where Bob's changes are now made permanent on the master branch. The merge request is processed by the network with the help of a voting algorithm which is described in section 5.6.

5.6 Voting

If a user would like to merge their changes to a protected branch, they create a merge request. To reach decisions concerning what merge requests are admitted to the protected branches, a voting algorithm is implemented. This algorithm's message flow is inspired by PBFT[13]. When a branch wants to be merged into the protected branch a voting event is instantiated at all peers, where each peer either accepts or denies this request. For the merge request to be accepted into the protected branch, the request need a majority vote.

The proposed voting algorithm consist of four stages:

- **Request**

The client node in figure 5.6 is the peer that initializes the voting by requesting a merge. The request stage is when the local server node signs the request, using digital signatures, and relays it to the rest of the network.

- **Voting**

Once the request has been received by the other peers, the voting stage is started, where the signature and the blocks are validated. Thereafter the voting starts, where the peers are prompted with a choice of either accepting the changes to the master branch or declining them.

- **Counting**

After each node has placed their vote, they broadcast their answer to the network, and thereafter the counting starts. A node will receive vote answers from its peers. When a "Yes" is received it will add one to the total sum. If a majority is not reached, the node will only sent a "OK" response. When a majority has accepted the changes the merge is made permanent on the DAG.

- **Result**

Once the voting has been finished, a result message is sent back to the local server node.

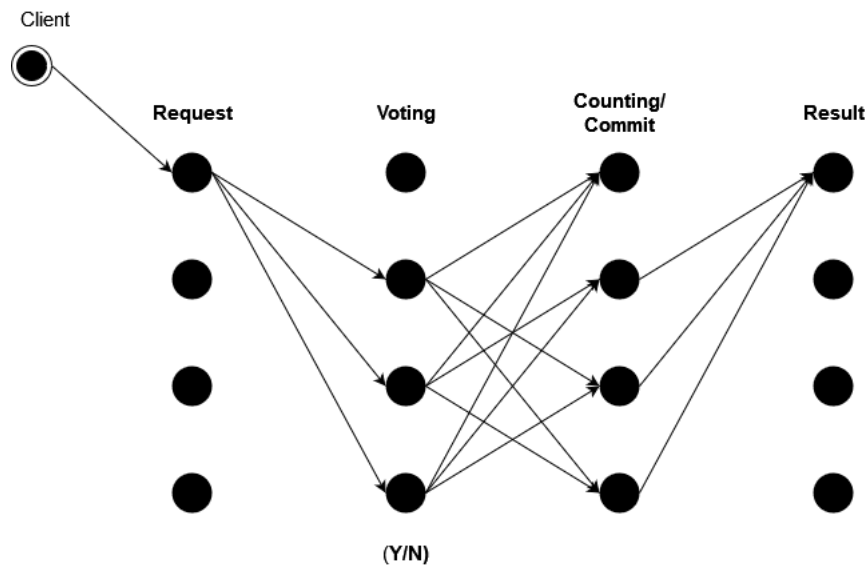


Figure 5.6: Flow of messages in the voting algorithm

How a node processes voting messages is presented in the pseudo code 1. Should the received message contain a commit to a protected branch without a vote in the message, then it means that the sending peer has started a merge request. This results in a prompt opening up for the peer to vote and broadcast its answers. Should the message contain a vote, then it is counted and compared to the overall number of peers. In this context an arbitrary majority number of 50% was chosen, but it can be altered to a preferred number. If the number is still beneath the required value, then the function returns and does not continue with processing the message. Once enough votes have been received does the function continue with security checks prior to making permanent changes in the distributed DAG ledger.

Algorithm 1 Pseudo code for voting algorithm

```

receive(message):
  if vote exist in received commit message then
    if message["vote"] == "Yes" then
      Increment vote_count with 1
      if vote_count < peers * 0.5 then
        Send socket message that vote is received
        return
      end if
    end if
  end if
  if master branch reference in commit message, with no vote in message then
    "" Merge request is initiated by sending peer ""
    Prompt y/n to user to receive answer
    Save answer as a vote in message
    Send my answer to the rest of the peers
    return
  end if
  . . .
  checks signature and validity of attempted commit
  Push commit to Distributed DAG ledger

```

5.7 The architecture in detail

This section give an overview of the architecture and its different components and to give some context of how these components are tied together. Figure 5.7 shows an simplified version of how these systems are tied together on a single node. As mentioned previously in the report, the Git implementation uses the DAG based ledger as a remote repository, represented as a RocksDB[31] database. The component responsible for what is admitted into the DAG ledger is represented in figure 5.7 as the Python DAG logic. This component, other than being responsible what gets admitted to the DAG ledger, is also responsible for the communication sent to and received by peers. With a network of multiple contributing nodes, the sending and receiving functionality is separated and accessed through different ports on the node computer. Therefore, a node can be represented with a sending node and a receiving node due to the functionality associated with each of the two.

As can be seen in figure 5.8 when commits are pushed, it is done through the local client. This client relays the message from the Git implementation to the local server peer. Illustrated in the figure, the blue squares are listening ports that receive messages from other nodes or clients. Before any changes are made the local peer checks the validity, which includes if the commits are referencing existing previous commits. Then, if the commits are being pushed to a protected branch, the message is relayed to the rest of the network, which in turn initiates a voting event on the receiving nodes. Otherwise the commit is signed using digital signatures and stored in the database, if the referenced branch is owned by this peer. Thereafter the commit message is broadcasted with the signature included.

Once the sending peer has broadcasted the commit messages to the network, the receiving peers will receive them and start to process them. If the branch reference is not a protected branch, same validity procedures are performed, with the addition of verifying the signature. Once this is performed the commits are made permanent in the receiving nodes databases. Should the branch reference be a protected branch, the receiving node would start to vote on

whether this should be allowed or not. The user on the reviving node is given a prompt which asks if this change should be made permanent, and then sends the answer to the rest of the peers. When a receiving node receives the answer it processes vote messages and checks if the vote is "yes". If so it increments its vote count and checks for majority vote. If a majority has been reached, the validity and signature are checked. Thereafter, if the commit is valid it is stored on the database.

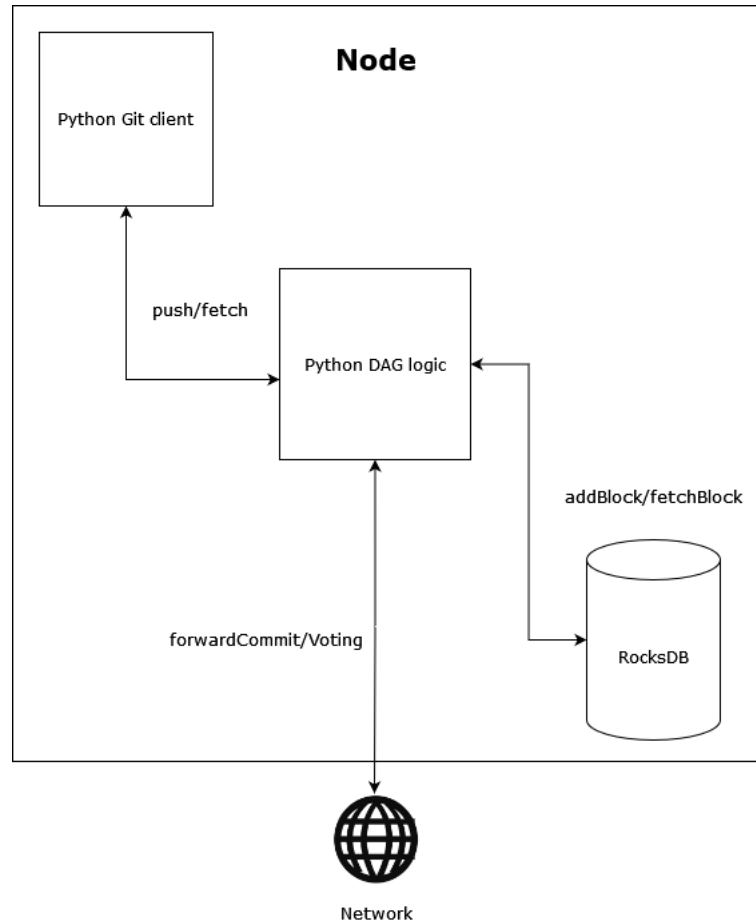


Figure 5.7: Overview of node functionality

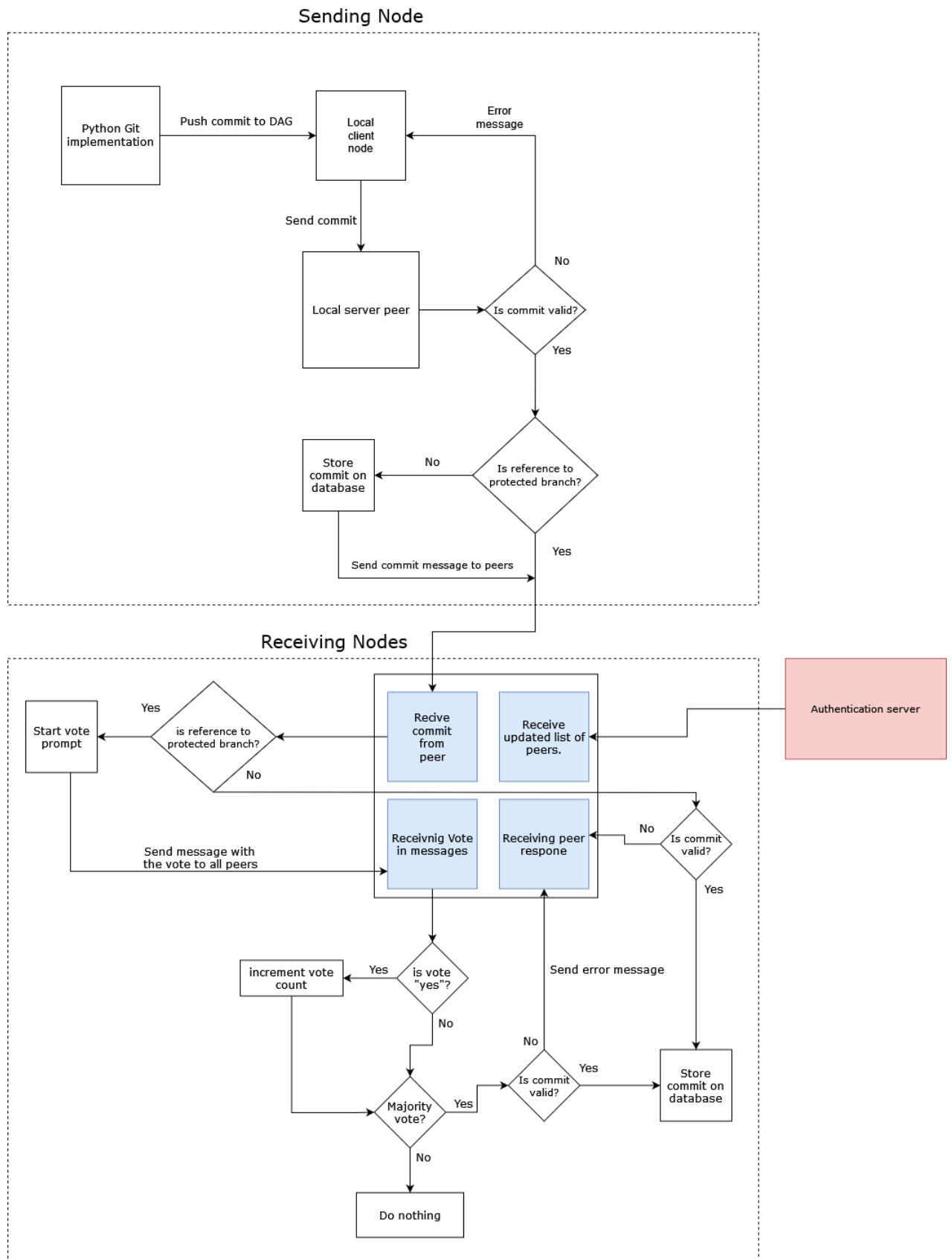


Figure 5.8: Flowchart of messages sent between peers



6 Evaluation

The purpose of this chapter is to present possible security risks of Git web services and evaluate the decentralized Git solution described in chapter 5 in terms of security. This is done by proposing misuse and use cases, using actors with different access privileges.

6.1 Evaluation Method

In order to evaluate security threats that might occur in a conventional Git web service there is a need to first identify these threats. In order to do so the actors in the Git web service Gitlab will be identified and their privileges in the project will be defined. Each use case for an actor is presented in table 6.1, where they are classified in accordance to what security property they violate if the use case is misused. The security properties in question are Confidentiality, Integrity and Availability. Where confidentiality represent not disclosing information to unauthorized actors, or in other words being able to protect data from being seen by parties that should not have access. Integrity represents how trustworthy the data is, for example not letting any unauthorized party edit the data. Availability represent accessibility to the data when an authorized party requests it[10]. After having classified these use cases, the ones that are within the scope and relevant to the research questions will be selected. These will then be evaluated for potential misuse cases for both conventional Git and the decentralized Git. The misuse case diagrams will be compared to each other to see what misuse cases is present in one but not the other.

6.2 The Git project

Given a Git project with more than one members and the protected branches, master and develop, roles are introduced to control the access to these branches. One or more members of the Git project have elevated privileges compared to normal developers, such as a maintainer or owner. The maintainer has free reign over the protected branches and changing the set-

tings for these branches. The owner has the same privileges as a maintainer, with the owner also having the option to delete the project among other privileges[9]. Developer is a role with less access rights than the maintainer, where the developers are free to branch from the protected branches and make changes to the code. However, when the developer wants to commit these changes to one of the protected branches they need to make a merge request. The request need the approval of other members in the project for the changes to be committed to the protected branch. With these actors described, we propose the idea of malicious maintainers, malicious developers and malicious reporter as listed in table 6.2. Table 6.1 lists the common use cases[9] for reporters, developers and maintainers.

Table 6.1: Use cases for the Gitlab web service

Usecase	User	Threat
Create merge request	Developer, Maintainer	I
Approve merge request	Developer, Maintainer	I
Manage merge approval rules	Maintainer	IA
Create, edit and delete releases	Developer, Maintainer	I
Pull & view project code	Reporter, Developer, Maintainer	C
Create New branches	Developer, Maintainer	I
Download project	Reporter, Developer, Maintainer	C
View project code	Reporter, Developer, Maintainer	C
Add new team members	Developer, Maintainer	I

Table 6.2 lists the actors that are studied, what privileges they have and what threat they pose to the three security properties.

Table 6.2: Malicious agents in the Gitlab web service

Attacker	Privileges	Threat
Malicious Maintainer	Read/write	CIA
Malicious Developer	Read/write	CIA
Malicious Reporter	Read	C

To narrow the scope of the thesis, only the integrity and availability security traits are being considered for this analysis. As the malicious reporter role does not pose a threat to these traits it will not be taken into account when proposing possible misuse cases. After narrowing down the use cases the remaining ones are listed in table 6.3.

Table 6.3: Narrowed down use cases for the Gitlab web service

Use case	User	Threat
Create merge request	Developer, Maintainer	I
Approve merge request	Developer, Maintainer	I
Manage merge approval rules	Maintainer	IA
Create New branches	Developer, Maintainer	I
Add new team members	Developer, Maintainer	I

6.3 Misuse cases in the Git project

To properly research possible attack vectors for the Git project, a subset of malicious actors are analysed.

6.3.1 Malicious maintainer

The maintainer role has elevated privileges compared to a developer. If the maintainer of the project is malicious in some way it can have grave consequences not only for the source code, but for the users of the software as well. The maintainer can make changes the code and introduce any malicious code they like. Taking the maintainers privileges into consideration, the misuse cases of note are the following for a malicious maintainer:

- Changes settings for access of protected branches and merge approval rules
- Commits malicious code to branches
- Approve malicious or poorly made commits

6.3.2 Malicious developer

Although the role of the developer does not have the privileges of a maintainer, a malicious developer can still be harmful to a project. The developer is able to approve merge requests in accordance with the rules applied by the maintainers, which can allow for vulnerabilities to be admitted to the protected branches if the commits in question are not properly checked. There is also the possibility of the malicious developer trying to commit malicious code to the protected branches. Thereof, the misuse cases in which the malicious developer can cause damage are:

- Approval of malicious merge requests
- Commit malicious code covertly for it to be approved to the protected branches

6.3.3 Misuse cases

The misuse case diagram in figure 6.1 visually describes the use cases of the systems and the misuse cases that a malicious agent can use to attack the project. The figure shows the misuse and use cases that were mentioned earlier in the chapter and how they relate to each other.

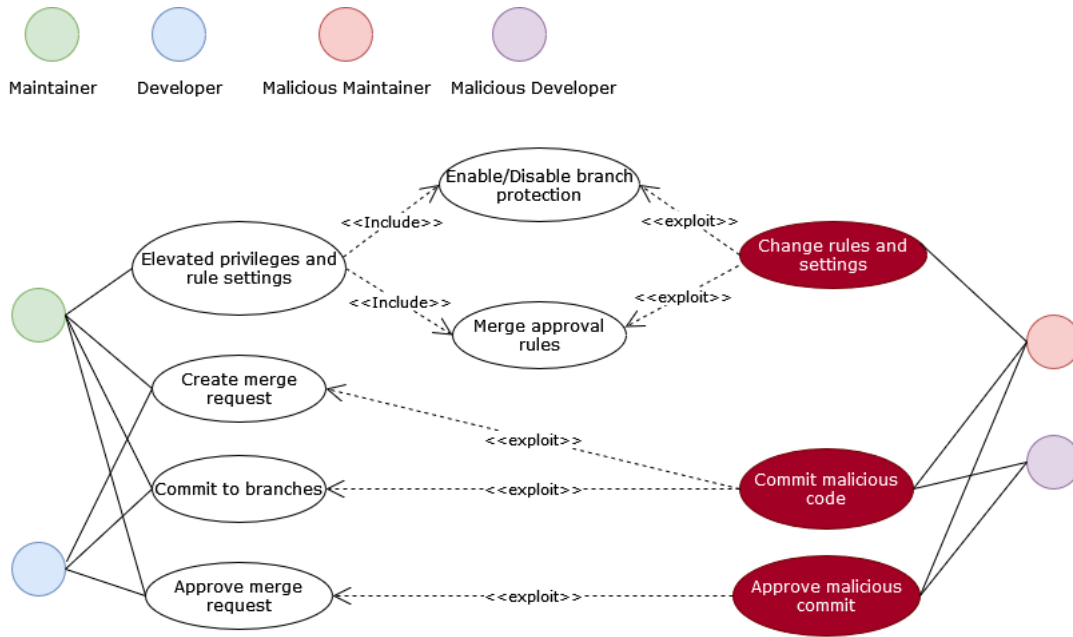


Figure 6.1: Overarching misuse case

The use cases, illustrated by the white nodes, show how a non-malicious agent can use the system. The red nodes show how a malicious agent might exploit these use cases to do some sort of harm to the system, these are what we call misuse cases.

6.4 Misuse cases in decentralized Git

The use cases and misuse cases of the Git project as presented in figure 6.1, are essentially the same for the implementation presented in chapter 5. With the introduction of the distributed DAG ledger and the removal of privileged access, some of the misuse cases present in the conventional Git implementation are mitigated or completely removed. However, the architecture as described in chapter 5 will present its own vulnerabilities which can be used by a malicious actor in misuse cases not present in figure 6.1. The vulnerability when the gateway server is breached and a malicious actor has access to the network, introduces another misuse case to the Git project which is the Sybil attack. By creating these virtual node identities, the attacker would gain a majority vote and essentially rendering the voting algorithm useless.

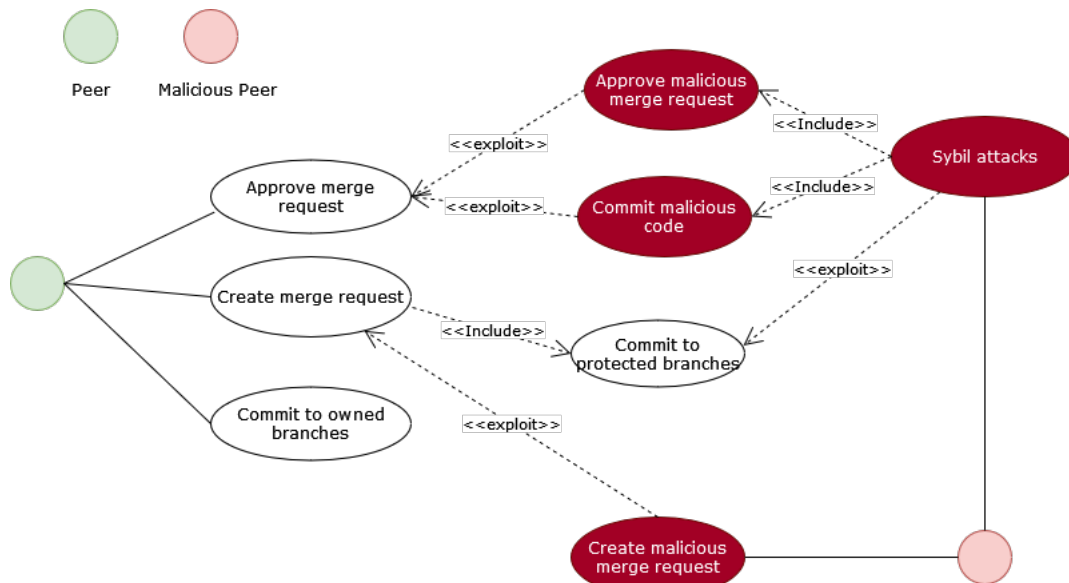


Figure 6.2: Misuse case diagram of decentralized Git

Considering the misuse case diagram in figure 6.1 we compare

Figure 6.2 illustrates how a peer can use the software and how a malicious peer might exploit these use cases. The option for changing rules is completely removed from the functionality of the system, because of this the exploits that a malicious maintainer might utilize is completely eliminated. However, the exploits that a malicious developer might utilize is still present in the decentralized version of Git. Though a malicious peer can create a malicious merge request, the rest of the network need to approve the merge request. In the case where a malicious peer utilizes a Sybil attack, where the peer takes over the network by creating multiple new peers under their control, the malicious peer has then control over anything that is made permanent in the protected branches. For example the peer can stop any commits from being merged into the protected branches or commit any malicious code the protected branches.

6.5 Vulnerabilities in a centralized Git web service

Centralized Git web services face the same type of threats as most web services since the Git repositories are hosted on web servers. This introduces threats that are common with web services, the OWASPTOP10[32] describes the 10 most critical security threats to web applications. These security threats vary in what type of breach they cause in the web application. Some exploits may enable an attacker to snoop information, while some allow for arbitrary code execution. Important to note is, if sensitive information is acquired by hackers, said information may be further used to perform damaging attacks, such as gaining the login credentials of an administrator. The most common vulnerability described in OWASPTOP10 from 2021 is broken access control, where users can access parts of the application that is outside of their access rights. For instance, an attacker may manually write in the URL to access password information.

6.6 Role based access control

The Git implementation described in chapter 5 collaborators in the project are referred to as peers. A central part of decentralizing the Git project is the security benefits of removing singular points where an attacker can gain access to the project and do harm. Since the decentralized Git does not include elevated access rights to protected branches or any rule setting regarding branch access, the maintainer role does not exist in the context of a decentralized version of Git. After the creation of the project and the protected branches have been set, the participants of the project may create branches of their own.

6.7 Decentralized Git security

An advantage using DLT is that the data stored is verified and distributed by all nodes in the network. If an attacker were to gain access to a peer in the network, the damage they can cause is limited. For an attacker to make any changes to the distributed ledger they would have to gain access to a majority of the nodes on the network. A possible attack vector for a malicious peer is to submit a malicious merge request. A malicious merge request made by a malicious agent can only be intercepted if the collaborators in the project can successfully identify the commit as malicious. Because of this need for human intervention in the code reviewing process, mitigating the risk of malicious code being committed to protected branches relies on the reviewers and their competence.

A possible point of entry to the decentralized Git for an attacker is the gateway server described in chapter 5. Given that the gateway server is the only entry to the network, an attacker could possibly deny entry to new users through denial of service attacks. Furthermore, should an attacker bypass the gateway server and access the network, they can simply create a branch of their own that may not be altered by others and start to commit large amount of data to the branch repeatedly. This would cause the receiving peers to fill their storage with useless data, and may eventually overwhelm the peers.

6.8 Voting

The voting system enables the network to reach decisions regarding the Git project without the need for a centralized authority. This allows for a more collaborative approach to maintaining a Git project. Removing the maintainer role also removes the threat of malicious agents exploiting the access privileges of a maintainer. However, this method of maintaining a project is vulnerable to sybil attacks. If an attacker can control a majority of the network it would in practice have the same privileges as a maintainer since all votes would go in favor of the attacker.



7 Discussion

This chapter discusses the design choices made, the security of these choices and the maintainability of the decentralized Git project. The chapter also mention how DLT affect society in a wider context.

7.1 Design choices

There are many possibilities for designing a Git application using distributed ledger technologies. Given the overall design described in chapter 4, the ledger, the network and the distributed storage, a best fitting implementation for these systems is up to discussion. Considering the network implementation presented in the thesis, the concept of an gateway server allowing peers to connect to network has its benefits. The implementation allows for a private network, where the gateway could deny access to any undeclared participants. Given that all participants are known to the system, the participants can be seen as honest. However, in the case where the gateway server is compromised the whole network would also be compromised. An attacker could, with control over the gateway server, admit multiple new malicious participants into the network, allowing for a sybil attack.

The distributed storage described in the thesis is a basic implementation where all participants in the project update their local file system. There are many flaws with implementing the distributed storage in this way. For instance one can not be certain that the file system is are kept consistent between all peers, considering possible faults in writing to the local machine. One major drawback to the system is the fact that data from all branches are stored in all peers. Since the only branches of note on the DAG ledger are the protected branches, any other branches that are not important to the participant only fill disk space needlessly. This also allows for any malicious participants in the network to fill the DAG ledger with garbage and could possibly deplete all the participants' free storage. One way to remedy this is by implementing a DAG based ledger similar to the LDV described in the paper by W. Yang et al,[19] where only branches that are important to the participant are stored.

The DAG based DLT is a relatively new concept when compared to conventional blockchains, where even blockchains are just recently being used in industry. Because of this there are limited options for developing applications on top of an existing DAG based ledger network, compared to ethereum that already has many blockchain applications running on it. Furthermore, the majority of DAG based ledger networks that are developed or being developed, use the DAG structure as a way of reaching consensus. An example of this is the tangle ledger used by IOTA, the DAG structure itself is a result of its consensus algorithm where each transaction is a node on the DAG ledger and each of these nodes need to validate two previous nodes to be admitted. This breaks the git branching structure and is therefore not an optimal solution. A possible way of implementing a decentralized git as described in the thesis is to have a blockchain for each individual branch. This solution could be implemented using a blockchain for each protected branch by, for example using the hyperledger framework.

7.2 RBAC

The thesis tries to research a way of maintaining a decentralized Git. The method suggested in the thesis is a voting algorithm where the participants in the project can hold votes for changes to be made to the project. However, this is not always an option for many types of projects. For example in project developed at a company where there are defined roles for the employees and the chain of responsibility needs to be clear. An appropriate use case for a decentralized Git could be open source software developed by a community.

7.3 Consensus

The voting algorithm described in section 5.6 could be seen as a consensus algorithm but it is important to note that the voting algorithm is not as complete or secure as for example the PBFT consensus algorithm. A key difference between the conventional PBFT consensus algorithm and the voting algorithm described in the thesis, is that the user at each node that receives the request needs to manually make the choice to approve or deny the merge request. Furthermore, the approval threshold for being committed to the DAG ledger can be set to any amount, similar to conventional Git web services it can be set to two or three approval votes.

7.4 Security

The threat modeling of the Git web service as presented in chapter 6, has been scoped down to a specific focus in role privileges and how they can be used against the system by malicious actors. This grants a good insight in that regard, but does not take into account other overarching vulnerabilities. By following a threat modeling process one could discover more vulnerabilities in both web based and in our proposed solution. Considering Petricia et al.[27], that modeled attack trees for their web service, one could see the typical web vulnerabilities known to the software industry. However, rather than creating new attack trees for potential web vulnerabilities for Git, this thesis focuses more on the consequences of malicious actors with elevated privileges.

Since DAG based DLT are rather new, it would have been beneficial to create an attack tree as well, to have an even greater overview of vulnerabilities for the DAG ledger solution. Simi-

larly to the experiences noted by Karpati et al. [25], where developers used misuse cases and attack trees and compared both modeling methods. The study found that more threats were discovered with attack trees and more mitigations were found using misuse case. Since the focus of this thesis was to find possible mitigations for a malicious maintainer, Karpati et al. does validate its use in the thesis.

7.5 The work in a wider context

Blockchain technologies have been widely discussed due to their enormous energy consumption seen in PoW consensus algorithms such as bitcoin. For DLT using PoW the energy consumption is a factor not to be taken lightly considering the effect it has on the global warming. Bitcoin is estimated to consume 204.50 TWh of electrical energy which would be similar to the amount that the entire country of Thailand is consuming[33]. For DLT using DAG based consensus algorithms such as the tangle, it does not rely on miners and because of that could be an alternative to the more power consuming PoW consensus algorithm.

Beyond the question of consensus algorithm, moving away from the conventional idea of having a centralized authority has its own consequence. In the case of banking, as DLT is mostly used for cryptocurrencies, there is always a central authority responsible for the assets that are stored at the bank. The user can be certain that if anything happens to the assets, such as a bank going bankrupt or a robbery, that their money is insured. In the case of DLT if anything goes wrong, for instance in a hacking incident or otherwise, there is nothing protecting the user. Hence, the decentralization could result in a lack of clear responsibility in the case where something goes wrong. The same can be applied to the application described in the thesis, where the lack of a maintainer also removes the chain of responsibility.



8 Conclusion

This chapter presents the answers to the research questions.

8.1 Aim and research questions

The aim of this thesis was to research and evaluate the security of a decentralized Git compared to conventional Git web services. The thesis proposed an architecture for a decentralized version of Git, which was later compared to conventional Git using misuse case diagrams 6.2 and 6.1 respectively.

8.1.1 What security threats in Git web services can be mitigated using a distributed DAG ledger?

The DAG ledger would be able to mitigate vulnerabilities in Git roles that have certain elevated privileges, as seen in diagram 6.1 and 6.2. It is apparent from these two misuse diagrams that the misuse cases available to a malicious maintainer is eliminated. The DAG blocks also has the property of being immutable and thus increasing the level of integrity of the system, where fraudulent commits are detected and discarded. Changing anything on the DAG block, such as the user ID, the resulting hash would be invalidated, which further increase the possibility to detect fraudulence.

8.1.2 What security threats might occur when introducing a distributed DAG ledger to a decentralized Git implementation?

The vulnerabilities that this solution introduces is the malicious actors' possibility of bypassing the gateway server and entering the network, where the malicious actor may overwhelm

the system by committing gibberish to the DAG ledger. Furthermore, there exists the possibility that an attacker can perform a Sybil attack and take over the network since a Git project has significantly fewer participants than for example cryptocurrencies. What misuse cases a malicious peer can perform with the help of a Sybil attack is illustrated in the misuse diagram 6.2. Gaining a majority would be significantly easier if the attacker gains access to the network.

8.2 Future Work

Given that the DAG based consensus algorithm uses the DAG structure to reach consensus, using a DAG based ledger could be difficult in the context of a Git project. A possible solution to decentralizing the Git project could be to implementing each protected branches as its own blockchain, where merge commits references a second blockchain. A suitable implementation for the blockchains could be the permissioned blockchain Hyperledger Iroha. To implement a DAG ledger as described in the thesis one would need a consensus algorithm that is not based on the DAG structure itself, since this would effect the Git branching structure.



Bibliography

- [1] Satoshi Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: *Decentralized Business Review* (2008), p. 21260.
- [2] Sara Rouhani, Luke Butterworth, Adam D. Simmons, Darryl G. Humphery, and Ralph Deters. "MediChainTM: A Secure Decentralized Medical Data Asset Management System". In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. 2018, pp. 1533–1538. DOI: 10.1109/Cybermatics_2018.2018.00258.
- [3] Federico Matteo Benčić and Ivana Podnar Žarko. "Distributed Ledger Technology: Blockchain Compared to Directed Acyclic Graph". In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. 2018, pp. 1569–1570. DOI: 10.1109/ICDCS.2018.00171.
- [4] Calum McConnell. *URGENT: SECURITY: New maintainer is probably malicious*. 2020. URL: <https://github.com/greatsuspender/thegreatsuspender/issues/1263> (visited on 02/09/2022).
- [5] Ravie Lakshmanan. "Alert! Hackers Exploiting GitLab Unauthenticated RCE Flaw in the Wild". In: *The Hacker News* (Nov. 2, 2021). URL: <https://thehackernews.com/2021/11/alert-hackers-exploiting-gitlab.html> (visited on 11/19/2021).
- [6] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends". In: *2017 IEEE International Congress on Big Data (BigData Congress)*. 2017, pp. 557–564. DOI: 10.1109/BigDataCongress.2017.85.
- [7] Git. *About- branching and merging*. <https://git-scm.com/about>.
- [8] S. Chacon and B. Straub. *Pro Git*. The expert's voice. License Creative Commons (BY-NC-SA). Apress, 2014. ISBN: 9781484200766.
- [9] *Permissions and roles*. URL: <https://docs.gitlab.com/ee/user/permissions.html> (visited on 12/20/2021).
- [10] D. Gollmann. *Computer Security*. Wiley, 2011. ISBN: 9780470741153. URL: <https://books.google.se/books?id=KTYxTfyjiOQC>.

- [11] Andreas M Antonopoulos. *Mastering Bitcoin: unlocking digital cryptocurrencies*. "O'Reilly Media, Inc.", 2014. ISBN: 978-1-449-37404-4.
- [12] Sjbrown. "Hash tree". In: (2012). License Creative Commons (CC0).
- [13] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance". In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. OSDI '99. New Orleans, Louisiana, USA: USENIX Association, 1999, pp. 173–186. ISBN: 1880446391.
- [14] Harish Sukhwani, José M Martinez, Xiaolin Chang, Kishor S Trivedi, and Andy Rindos. "Performance modeling of PBFT consensus process for permissioned blockchain network (hyperledger fabric)". In: *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2017, pp. 253–255. DOI: 10.1109/SRDS.2017.36.
- [15] Dmitry Dzhus. "Tred-G". In: (2007). License Creative Commons (CC0).
- [16] Niclas Kannengießer, Sebastian Lins, Tobias Dehling, and Ali Sunyaev. "Trade-Offs between Distributed Ledger Technology Characteristics". In: *ACM Comput. Surv.* 53.2 (May 2020). ISSN: 0360-0300. DOI: 10.1145/3379463. URL: <https://doi.org/10.1145/3379463>.
- [17] *Threat Modeling Cheat Sheet*. URL: https://cheatsheetseries.owasp.org/cheatsheets/Threat_Modeling_Cheat_Sheet.html (visited on 01/21/2022).
- [18] I. Alexander. "Misuse cases: use cases with hostile intent". In: *IEEE Software* 20.1 (2003), pp. 58–66. DOI: 10.1109/MS.2003.1159030.
- [19] Wenhui Yang, Xiaohai Dai, Jiang Xiao, and Hai Jin. "LDV: A Lightweight DAG-Based Blockchain for Vehicular Social Networks". In: *IEEE Transactions on Vehicular Technology* 69.6 (2020), pp. 5749–5759. DOI: 10.1109/TVT.2020.2963906.
- [20] Statista. *Size of the Bitcoin blockchain*. 2021. URL: <https://www.statista.com/statistics/647523/worldwide-bitcoin-blockchain-size/> (visited on 09/06/2021).
- [21] Yousef Hashem, Elmedin Zildzic, and Andrei Gurtov. "Secure Drone Identification with Hyperledger Iroha". In: *DIVANet '21*. Alicante, Spain: Association for Computing Machinery, 2021, pp. 11–18. ISBN: 9781450390811. DOI: 10.1145/3479243.3487305. URL: <https://doi.org/10.1145/3479243.3487305>.
- [22] Yepeng Ding and Hiroyuki Sato. "Dagbase: A Decentralized Database Platform Using DAG-Based Consensus". In: *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. 2020, pp. 798–807. DOI: 10.1109/COMPSAC48688.2020.0-164.
- [23] Sheng Wang, Tien Tuan Anh Dinh, Qian Lin, Zhongle Xie, Meihui Zhang, Qingchao Cai, Gang Chen, Wanzeng Fu, Beng Chin Ooi, and Pingcheng Ruan. "Forkbase: An efficient storage engine for blockchain and forkable applications". In: *arXiv preprint arXiv:1802.04949* (2018).
- [24] N. Nizamuddin, K. Salah, M. Ajmal Azad, J. Arshad, and M.H. Rehman. "Decentralized document version control using ethereum blockchain and IPFS". In: *Computers Electrical Engineering* 76 (2019), pp. 183–197. ISSN: 0045-7906. DOI: <https://doi.org/10.1016/j.compeleceng.2019.03.014>.
- [25] Peter Karpati, Yonathan Redda, Andreas L. Opdahl, and Guttorm Sindre. "Comparing attack trees and misuse cases in an industrial setting". In: *Information and Software Technology* 56.3 (2014), pp. 294–308. ISSN: 0950-5849. DOI: <https://doi.org/10.1016/j.infsof.2013.10.004>.
- [26] Chee-Wooi Ten, Chen-Ching Liu, and Manimaran Govindarasu. "Vulnerability Assessment of Cybersecurity for SCADA Systems Using Attack Trees". In: *2007 IEEE Power Engineering Society General Meeting*. 2007, pp. 1–8. DOI: 10.1109/PES.2007.385876.

- [27] Gabriel Petrică, Sabina-Daniela Axinte, Ioan C. Bacivarov, Marian Firoiu, and Ioan-Cosmin Mihai. "Studying cyber security threats to web platforms using attack tree diagrams". In: *2017 9th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*. 2017, pp. 1–6. DOI: 10.1109/ECAI.2017.8166456.
- [28] Anton Churyumov. "Byteball: A decentralized system for storage and transfer of value". In: URL <https://byteball.org/Byteball.pdf> (2016).
- [29] Serguei Popov. "The tangle". In: *White paper 1.3* (2018). URL: https://assets.ctfassets.net/rldr6vzfxhev/2t4uxvsIqk0EUau6g2sw0g/45eae33637ca92f85dd9f4a3a218elec/iota1_4_3.pdf.
- [30] Bin Cao, Zhenghui Zhang, Daquan Feng, Shengli Zhang, Lei Zhang, Mugen Peng, and Yun Li. "Performance analysis and comparison of PoW, PoS and DAG based blockchains". In: *Digital Communications and Networks* 6.4 (2020), pp. 480–485. ISSN: 2352-8648. DOI: <https://doi.org/10.1016/j.dcan.2019.12.001>.
- [31] Dhruba Borthakur et al. *RocksDB Overview*. URL: <https://github.com/facebook/rocksdb/wiki/RocksDB-Overview#author-dhruba-borthakur-et-al> (visited on 03/17/2022).
- [32] "OWASP Top Ten". In: OWASP (2021). URL: <https://owasp.org/www-project-top-ten/> (visited on 12/21/2021).
- [33] *Bitcoin Energy Consumption Index*. URL: <https://digiconomist.net/bitcoin-energy-consumption> (visited on 02/02/2022).