Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Applying Software Diversity to Hyperledger Fabric Smart Contracts

Bachelor's Thesis

Author
Zoltán Bende

Advisor
Attila Klenik

May 31, 2020

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Bende Zoltán*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2020. május 31.

Bende Zoltán
hallgató

Kivonat

A gyakorlatban a szoftver diverzitás mintákat gyakran alkalmazzák a rendszer hibatűrésének javítására. Ezen szakdolgozat alkalmából az új 2.0-as verziójú Hyperledger Fabric okosszerződés mechanizmusát felhasználva szoftver diverzitás mintákat fogunk használni, és méréseket hajtunk végre, hogy megvizsgáljuk a szoftver diverzitás hatását a Hyperledger Fabric legújabb verziójának teljesítményére.

A szakdolgozat során definiáljuk a szoftver diverzitást és megvizsgáljuk hogyan használják a szoftver diverzitás mintákat egy rendszer hibatűrésének javítására. Áttekintjük ezen belül a szoftver diverzitás néhány alkategóriáját és minden alkategóriához megvizsgálunk egy mintapéldát.

Ezután bemutatjuk a Hyperledger Fabricot és annak új okosszerződés mechanizmusát. Majd megnézzük hogyan tudunk megvalósítani szoftver diverzitás mintákat a Fabric 2.0-as verziójával hibatűrés javítása érdekében.

Emellett a szakdolgozat során három mérést végzünk el, melyekben összehasonlítjuk a régi Fabric teljesítményét az új teljesítményével, ugyanolyan hálózati felállás esetén. Majd megvizsgáljuk hogyan tudunk olyan méréseket megvalósítani, melyekben szoftver diverzitás mintákat használunk. Végül áttekintjük a témával kapcsolatos lehetséges jövőbeli munkákat.

Abstract

Software diversity patterns are widely used in the industry to provide fault tolerance to distributed systems. We are going to utilize the new Hyperledger Fabric smart contract mechanism introduced in Fabric version 2.0 to use software diversity patterns on the Fabric network. We run measurements to compare the effects of software diversity on the performance of the new version of Hyperledger Fabric.

The thesis explores what software diversity is and how software diversity patterns are used to improve the fault tolerance of a system. We will look at certain categories of software diversity and explore scenarios belonging to these categories.

Next, we introduce Hyperledger Fabric and the new smart contract mechanism in version 2.0. Then we examine how can we implement software diversity patterns in Fabric 2.0 to improve fault tolerance.

We also perform three measurements, comparing the old Fabric's performance with the new on networks with the same logical setup. We explore the possibilities of creating measurements that implement software diversity patterns in Fabric 2.0. Finally, we discuss the future research possibilities related to the thesis.

Chapter 1

Introduction

Distributed ledger technologies play an increasingly important role in business sectors as a distributed platform for collaboration. These platforms' application logic is called smart contract, and smart contracts are distributed across the network participants. The platform we are going to use is Hyperledger Fabric, the most widely used DLT in the industry. In Fabric the smart contracts (called chaincode in Hyperledger Fabric) execute the incoming transaction and use a voting mechanism to determine whether they should accept the transaction or not. It is very important that DLTs have a certain degree of fault tolerance, but in older versions of Hyperledger Fabric and in almost all other major DLT platforms, it wasn't possible to use different smart contracts on different nodes of the network. This makes the system vulnerable to faults in the chaincode, since all nodes might return the same faulty result. Starting from version 2.0, Hyperledger Fabric gives us the ability to use different chaincodes on different nodes of the network, which enables us to use software diversity patterns to improve the fault tolerance of the system.

First chapter explores what software diversity is, and how it relates to designing fault tolerant systems. The chapter also presents a few subcategories of software diversity and concrete applications of software diversity patterns. In particular, we define the concepts of performance, availability and robustness related to fault tolerance and explore the difference between managed diversity and artificial diversity.

Second chapter delves into Hyperledger Fabric, and the new chaincode mechanism introduced in version 2.0. We define what distributed ledger technologies are and the difference between permissioned and permissionless ledgers. Then we explore how Hyperledger Fabric works, and explain the difference between the old and the new Fabric version. This chapter also explores how software diversity patterns with the new chaincode mechanism can improve the fault tolerance of the system.

Third chapter describes the three measurements we took to test the viability of software diversity patterns in Hyperledger Fabric. In these measurements we try to compare the performance of the old version with the new one, and examine the possibilities for using software diversity patterns. We describe the technologies we used to create the measurements and how we configured Hyperledger Fabric to enable us to perform these measurements. In the end, we show the results we achieved running when we've run the measurements.

Chapter 2

Software diversity

In the following chapter we are going to discuss the theoretical aspects of software diversity, and how it is related to fault tolerance. To do this first we are going to define a few important concepts related to fault tolerance, such as performance, availability, and robustness. Then we'll define what software diversity is. Finally, we are going to look at a few examples of using software diversity patterns, and we'll examine these patterns and how they affect the fault tolerance, or more concretely, the performance, availability, or robustness of the system.

2.1 Fault tolerance

First, we have to explain the definition of fault in a system. A system failure is a condition when the system no longer complies with the specification. Certain parts of the system state might cause system failure, these are called errors. And the cause of such errors are called faults[7]. In software engineering it is very important to have a system that continues to function even if there are faults in the system, therefore we need to think about ways to ensure the system has a certain amount of tolerance against faults. Hence, fault tolerance becomes an important subject to study.

Fault tolerance is often achieved by replicating the same component, and creating a voting system with the majority of the components assumed to be giving the correct answer. We might also have redundancy, which is multiple copies of the same component in case one might fail. The third is software diversity, which is explained in the next section.

2.1.1 Performance

In a system that performs a certain number of operations, for example a web server, performance means the number of transactions the system can support in a given time, before the system fails. Failures can be partial all complete, that is after a certain number of transactions the system might slow down, but still function, or the entirety of the system might fail. The specification of a system often contains the expected performance for the use case it is intended to satisfy, and it is important to design our software to meet the specifications[7].

Using software diversity patterns might impair to a degree the performance if, for example, we have a system where multiple diverse components have to vote on what should be the next result of a computation.

2.1.2 Availability

Availability is the amount of time the system can perform its function according to its specification. This is often measured in the number of nines after 99 percent, that is the amount of time the system is working divided by the total time.

Software diversity often improves availability since the system has no single point of failure where software diversity patterns are introduced, because the code running on the components is diverse, therefore we can trust not necessary a single component, but the components that return the same result.

2.1.3 Robustness

Robustness is measuring how many errors can a system tolerate before system failure happens. Robustness might include fault tolerance in runtime execution or fault tolerance from the inputs of the system[27].

For the same reasons mentioned above software diversity also improves robustness, variety means it is less likely that all the components fail simultaneously in the same way.

2.2 Introduction to software diversity

Software diversity is a collection of different techniques in software engineering that improve the quality of software by introducing variety into some parts of the soft-

ware. This might mean artificially introducing variety to the system, the diversity that exists in open source software, or creating multiple implementations of the component while maintaining the same interface for the component. Similarly, to diversity in nature, software diversity improves the resilience and stability of the software.

2.2.1 Various uses cases of software diversity

For our purposes the most important use for software diversity patterns is creating fault tolerance for a given system. This happens for many reasons, one is that if we have a variety of components, it is unlikely they all contain the same bugs, therefore if the components return the same result, then the change that all the components had the same bug is very unlikely.

Software diversity can also be useful for software security. The main reason for this is that if an attacker wants to infiltrate a system then the variety in the components makes it more laborious and difficult to penetrate it, since the security bugs are different in each component.

Software diversity is also a useful concept when it comes to creating tests for a system. This is done by generating tests automatically and introducing variety into the test generation. This way we can increase our chances of finding bugs in the system as opposed to tests created by humans.[21]

2.3 Software diversity examples for fault tolerance

We can separate software diversity into two broad categories: managed software diversity and artificial software diversity. Managed diversity means diversity that is created by the way software developers interact with the software development process (like creating different implementations of the same specification), whereas artificial diversity refers to methods where we are artificially generating diversity mechanically. These are the two main categories we will classify the following examples where software diversity is used to enhance fault tolerance[4].

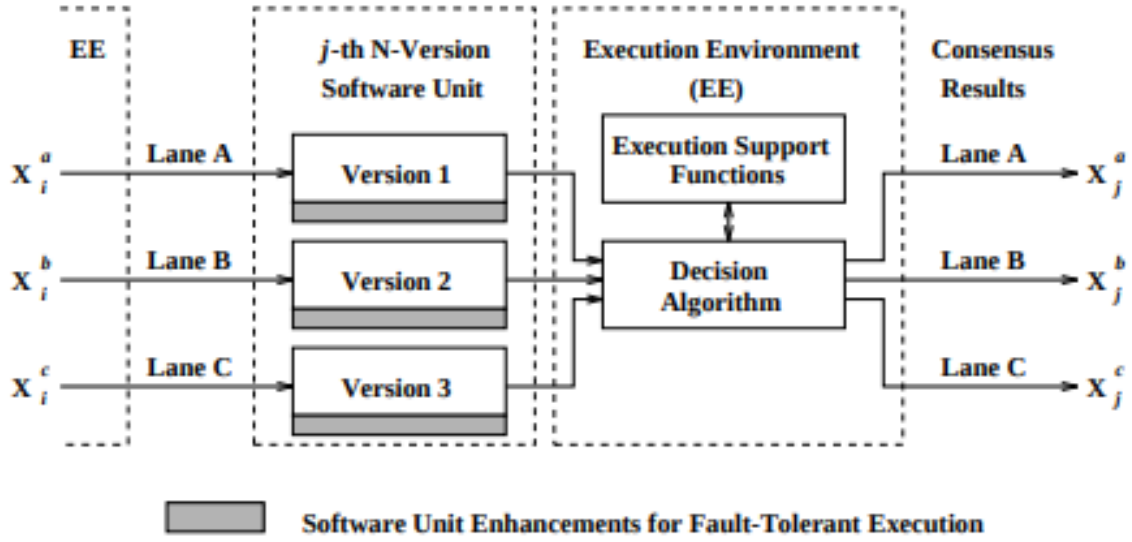

Figure 2.1: Diagram of the N-version pattern
[3]

2.3.1 Managed design diversity

One of the important branches of design diversity is N-version software illustrated in figure 2.1. This means the pattern of having two or more software implementations of a specification. These implementations are usually developed by software development teams independently so even if all teams introduce bugs into one of the implementations, it will be unlikely that these implementations will have the same bugs. The way they implement the specification can be very different from each other, and a voting mechanism decides what the output should be when we run the different implementations. The winning result of the voting is the most likely to be correct then[3].

Another model for design diversity is recovery blocks illustrated in 2.2. Recovery blocks are similar to N-version software, they both have different versions of a given program, but the way recovery blocks decide which one to choose is different. Recovery blocks have an acceptance test meant to decide whether the result of a given software version had a fault in it, and if so, the system can recover the previous state and can try to run the next software version with the same input, and so on, until the result becomes acceptable.

Both of these methods improve the robustness of the system, but the first solution fails if the majority of results are faulty, and the second fails if the acceptance criteria can't detect the mistake. As for performance, the first method has to wait for all the versions to provide results, only then can the voting happen, while the

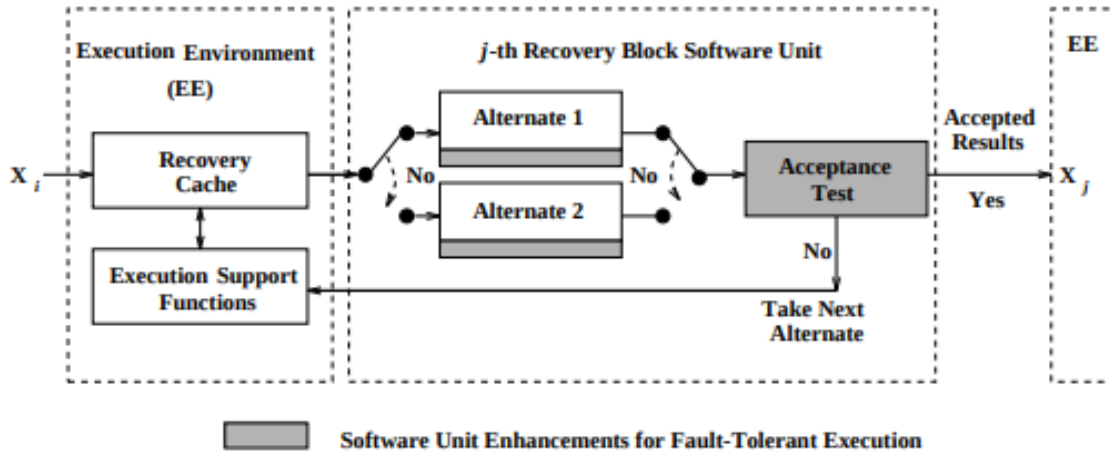

Figure 2.2: Diagram of recovery blocks
[3]

second method has to restart every time the acceptance test fails. This might hinder performance, but how to implement these solutions depends on the specific applications.

2.3.2 Managed natural diversity

Natural diversity means diversity that emerges from spontaneous processes. The biggest example is the open-source movement where similar functionalities are provided by a diverse set of solutions, often driven by market competition.

One example of natural software diversity is the Cactus approach[8] illustrated in 2.3. This framework is meant to add customizability to services in networked systems. Customizability is achieved by giving the users of the framework the ability to customize the components of the service as they see fit, letting the users add variety to the system. The framework has so-called Composite protocols, which is a form of middleware that accepts messages as inputs and produces messages as outputs. Inside these are the micro-protocols, which are event handlers, and they modify the message if an event triggers them. This is the part of the framework that is highly customizable by the users of the framework.

The way this approach enhances robustness is that different micro-protocols might provide a solution for different potential faults. Using software diversity also increases the availability of the system, since if the system is more robust than it will take more time for a system failure to occur. But there is a trade-off, namely between performance and robustness and availability, but it is possible to fine-tune the configuration so that we can balance the various considerations.

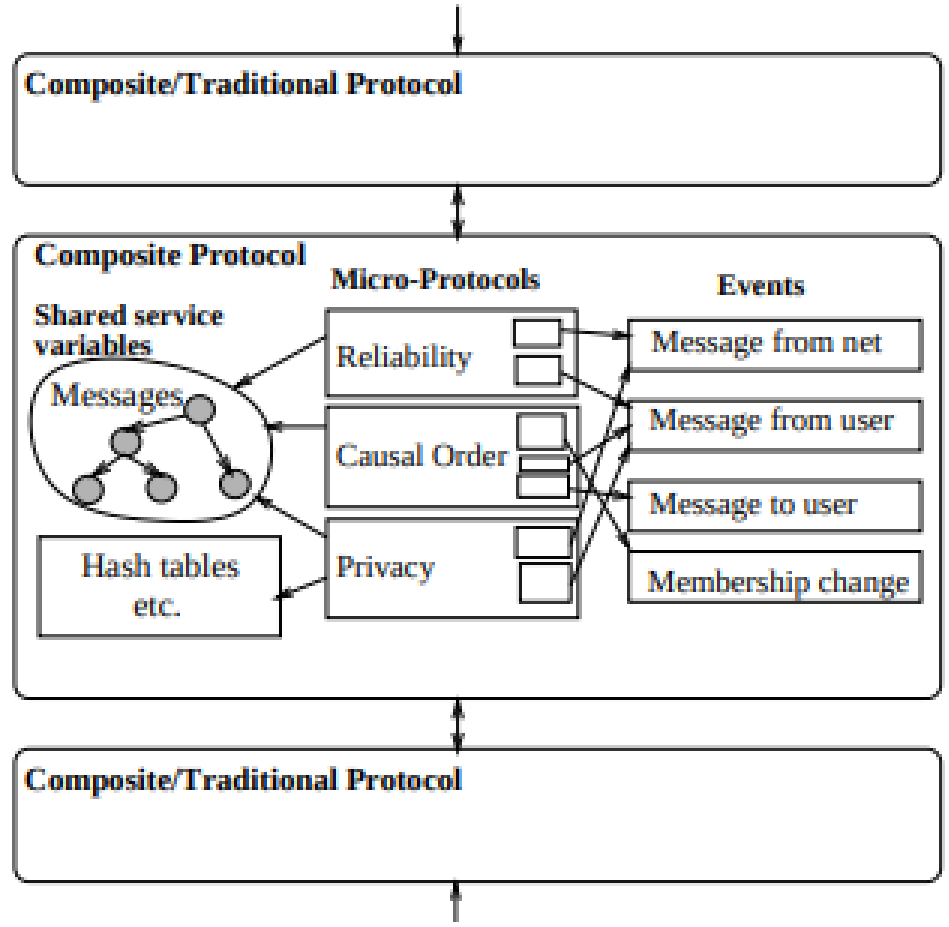

Figure 1. Composite protocol.

Figure 2.3: the composite protocol with the micro-protocols [8]

2.3.3 Automated execution diversity

Automated execution or randomization means creating diversity by introducing randomness to the execution of a certain program code automatically. There are many ways to do randomization, we might change the source code or the binary code of a program, or create diversity during runtime execution of the software.

An example of such diversity is the technique named data diversity[18]. This method involves modifying the input data of a system in case the original input causes errors in the system. This enables the user of the technique to create fault tolerance relatively inexpensively and on top of the fault tolerance of the original design.

The research paper on data diversity shows that with this method the performance varies depending on the parameters used, but performance is generally acceptable. As far as robustness goes, this method is very advantageous since it reduces the problems an error inducing input might produce.

Chapter 3

Software diversity in Hyperledger Fabric 2.0

This chapter introduces distributed ledgers and in general Hyperledger Fabric as the main technology we intend to use to implement software diversity patterns and conduct measurements. In particular, we are going to use the new smart contract mechanism implemented in the 2.0 version of Fabric.

3.1 Distributed Ledger Technologies (DLT)

Distributed Ledger Technologies is a novel approach with the main purpose of providing an alternative to centralized services. An example for a centralized service is the SWIFT global money transfer system for banks, which allows banks to transfer information between each other through a central participant. The reasons for using DLTs over centralized services is that the participants don't have to rely on trusting a centralized organization, secondly, DLTs allow the participants to hold each other accountable, and finally there is no single point of failure since the participants don't have to rely on the fault tolerance of the centralized entity. The difference between these approaches is illustrated in 3.1

The main way DLTs are implemented is by using blockchains. A blockchain- based implementation puts the transactions in sequential order and groups them in blocks. A blockchain always has a starting block, and the next block in a list of blocks always has the hash of the previous block inside it. Each participant holds the complete copy of this chain of blocks, and the hashes inside each block make sure nobody can alter the previous blocks without changing all the blocks after it. It also ensures that nobody can place their block into the list, or modify an existing block since

modification would change the original hash of a certain block. This chain of blocks is the ledger that replaces centralized bookkeepers.

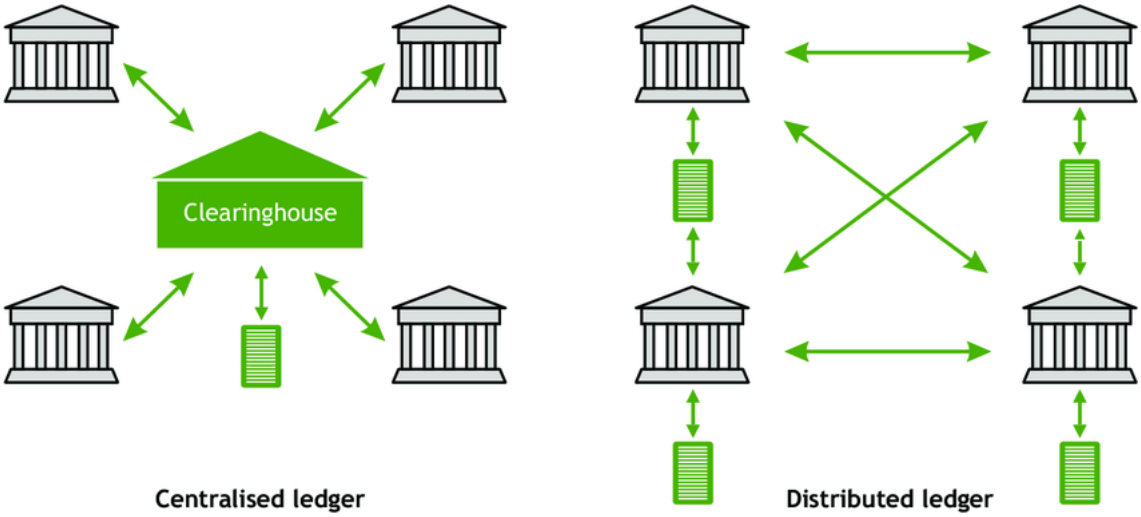

Figure 3.1: Centralized vs Decentralized ledger
[22]

Most ledger networks share a certain number of similarities:

- they all have a replica of the entire ledger
- it's only possible to append new blocks to the ledger
- there is no opportunity for modifying the ledger
- all DLTs have a method for verifying the validity of each block

There are, however, a few key differences, the most important being whether a network is permissionless or permissioned, or in other words public or private[1].

3.1.1 Permissionless ledgers

The main important feature of permissionless ledgers is that anybody can participate in this kind of network, and there are no restrictions on the number of participants. Permissionless ledgers can be used as decentralized currency, and on certain networks, it's even possible to perform computations. A famous example is Bitcoin[17], which allows participants to transfer virtual money to each other. Another important technology worth mentioning is Ethereum[5], which introduces the concept of smart contracts, allowing the network participants to set rules governing the transaction between them.

To secure a permissionless ledger, it is necessary for most currently working networks to perform expensive computations (like the proof of work[2] algorithm of the previously mentioned networks) that consume a lot of energy, so running the network tends to be very wasteful. The network throughput also tends to be very slow, which makes it unsuitable for large transaction volumes, and since it is an open network we can't design the network for a given throughput. However, permissionless ledgers are usually very secure, since behind each transaction there is a large amount of computation to prove the validity of the transaction.

3.1.2 Permissioned ledgers

Contrary to permissionless ledgers, the number of participants in a permissioned ledger is restricted to a smaller number. Permissioned ledgers are usually utilized by organizations to keep track of transactions between each other in a distributed fashion, like for instance food supply chains[14]. The primary resource traded in a private network usually isn't money, but a certain kind of token or asset that the network participants want to keep track of (for example agriculture products, or containers used in transportation).

Since the number of participants in the network is limited, the consensus mechanism for permissioned ledgers don't require expensive computations to keep the network secure, so it doesn't consume as much energy as public networks and the small number of participants makes it possible in distributed systems to use the well-known Byzantine fault-tolerant protocols[15]. Finally, the transaction throughput is much higher with permissioned ledgers, so for many organizations private ledgers are usually preferred.

3.2 The Hyperledger Fabric Platform

For demonstrating software diversity patterns we are going to use Hyperledger Fabric, which is an enterprise-grade software platform developed by the Linux Foundation. It is the most mature and well maintained permissioned distributed ledger currently available on the market, with industrial applications ranging from enabling tracking food supply chains, financial applications, and blockchain-based energy management[16][19]. Thanks to the design of the Fabric platform, there exist a multitude of options for customizing a network for a particular problem, according to the needs of the organizations using the system, making it a versatile choice for

its users. Finally, it is worth mentioning that a Hyperledger Fabric network is also a fast network, capable of 3500 transactions per second.[20]

The new 2.0 version has a certain set of newly added features that makes it possible to implement patterns for software diversity, the most important for this thesis being Hyperledger Fabric's new smart contract mechanism (for further reading see chapter 2.3.3). Fabric 2.0 enables multiple architecture patterns, making it ideal for using and measuring the performance of software diversity patterns[12].

Hyperledger Fabric in general has a highly modular architecture, a network that consists of many small pieces of distinct components, working together to create a functioning network. Between these components transactions take place, updating the ledger state according to a certain set of rules. The rules are smart contracts written in a general programming language, called chaincode in Hyperledger Fabric[11].

3.2.1 Network participants

Hyperledger Fabric is a closed network where the participants (usually government organizations or businesses) have cryptographic keys to identify them on the network. In Hyperledger Fabric these participants have a logical representation on the network, they are called organizations.

Organizations on the network can own several components illustrated in 3.2. First is the client application that initiates a given transaction on the network. Other components are peers that hold copies of the ledger and the chaincode associated with it and orderers that put the transactions into blocks sequentially.

The first important component involved in a Hyperledger Fabric network is the client. These are the components on a network that initiate the transactions that the organizations using the network want to execute. All components the client might interact with have unique identities and all the operations on the network are only executable with sufficient authorization.

The second component to mention is peers. The main purpose of peers is to store and update the ledger of the network, the shared database that the network is meant to keep track of. If clients want to interact with the private ledger, they have to do it through the peers in the network. They also host the chaincodes that determine the rules for how you can modify the ledger, though this is no longer the requirement in Fabric 2.0. An organization might have multiple peers, enabling scalability and fault tolerance.

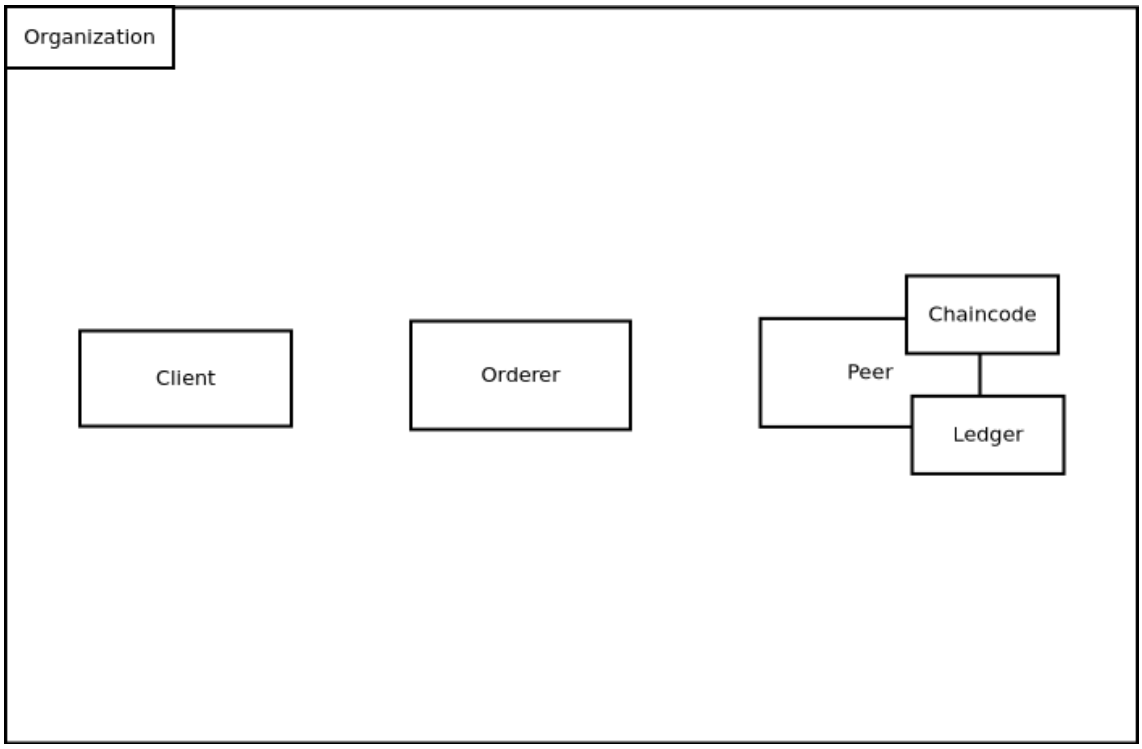

Figure 3.2: Components of an organization

Another important element in a Fabric network is orderers. When peers initiate transactions there has to be an order in which these transactions take place. This makes it necessary to have a mechanism in a network, to globally order the concurrently occurring transactions in the network.

Usually, when a client wants to create a transaction and change the state of the ledger, they want to set rules on how the ledger should be updated. These rules have to be agreed upon by the participating organizations, and the transaction has to be executed on multiple peers for the transaction to be valid. The main mechanism for creating executable logic is chaincodes that enable developers to create executable code in their preferred programming language.

It is worth mentioning that Fabric also has very sophisticated solutions for managing the identities of the users and the components participating in a network, but we are not going to dive into identity management in detail, since it is not relevant to the measurements we intend to perform.

3.2.2 The Fabric Consensus

In general, to create a transaction multiple steps need to happen: first, the client has to initiate the transaction, then send it to the peers, where the transaction gets

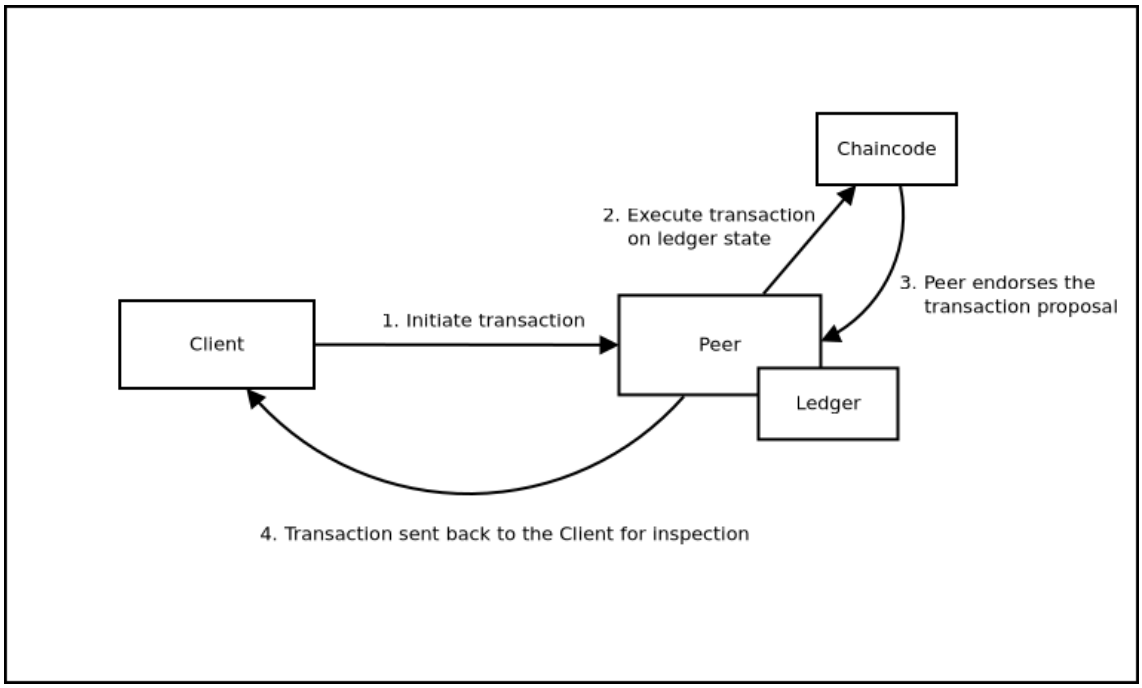

Figure 3.3: Transaction execution and endorsement

executed and the proposals get signed. Then the transaction is sent back to the client, the client inspects it, then sends it to the orderer. The orderer packs the transaction with others into a block, and sends it back to the peers. The peers validate the transactions and updates the ledger state. These steps are illustrated in figure 3.3 and 3.4.

The first step in the transaction lifecycle starts with the client. The client has to assemble a transaction proposal to create a transaction, then the transaction proposal is sent to the peers for the next step.

After the transaction proposals arrive at the peers, they inspect the proposal and check whether it is well-formed and whether the client has the authority to initiate the transaction, which is customizable in a Hyperledger Fabric network. In case all the conditions are met then the transactions will be executed in the chaincode on the current state which is accessed through the peer. This produces transaction results consistent with the inputs, but the ledger does not get updated. It is worth mentioning that a chaincode can have multiple transactions under execution at the same time. After the execution, the peer has to endorse the transaction, which means the peer signs the transaction with its signatures.

The transaction results then get sent back to the client who inspects the results and checks whether all peers necessary for the transaction approved the transaction and sent back the same state modifications to the ledger. It is not necessary though to

inspect the transaction results since in later stages of the transaction flow the peers will check whether the proper endorsements have taken place before updating the ledger.

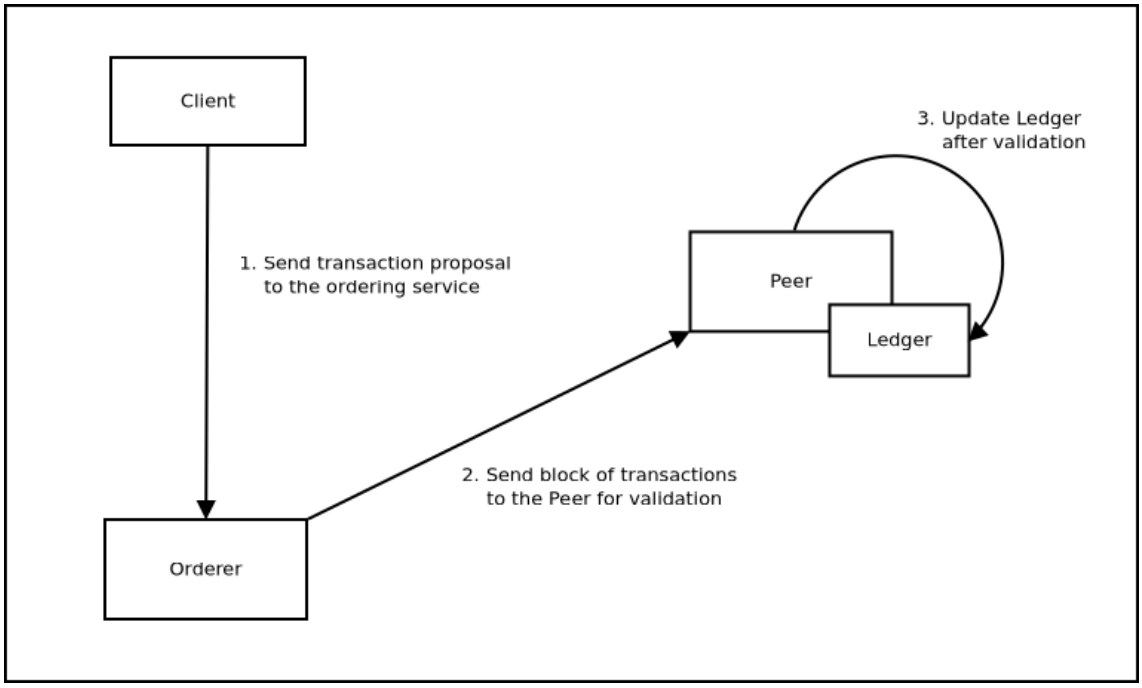

Figure 3.4: Transaction ordering and validation

The client then sends the results to an orderer node whose responsibility is to take this transaction and other transactions sent to it by other clients and package them into blocks. Then these blocks are sent to the peers for validation. One thing to note is that the orderers don't inspect the contents of the transactions, only order them.

The next phase in the transaction flow is validation, which first means making sure the transactions in the block have the proper endorsements necessary specified by the endorsement policies of the chaincode corresponding to the transaction. The ledger state is stored in versioned key-value pairs in Hyperledger Fabric, and since a block contains many updates to the state of the ledger, it might be necessary to tag certain intended changes to a ledger state invalid, since the modification of a variable might depend on a key-value pair other transactions already modified. Therefore, the peers have to check the validity of a state change to avoid writing into the database values.

Finally, after the validation phase, the ledger is updated with the new values by adding the new block to the end of the chain and applying the valid transactions. The peers then notify the client whether the ledger update was successful or not.

3.2.3 Channels and chaincodes

For the transactions between the peers to be possible, it is necessary to have a Channel between the peers and the orderers involved in transactions. A Channel is the mechanism that allows communication between these components in Hyperledger Fabric. A Channel usually has a configuration file defining its properties and from that, a software kit creates the first block, called the genesis block for the Channel. After that, the Channel is established and a client can initiate transactions on the behalf of an organization.

While permissionless ledgers usually have one ledger that records all transactions on the network, Hyperledger Fabric can run multiple ledgers on the same infrastructure. Therefore we can designate a certain subset of components on the same network (peers and orderers) to maintain a ledger. This is a form of partitioning, and at the same time, it allows the separation of ledgers by providing separation by data and authorization. Channels also allow chaincodes to be separated from each other since chaincodes on different channels are separated.

It is also useful to have the ability to update the ledger according to certain logic the participants agree to. Therefore Hyperledger Fabric has the concept called chaincodes to enable developers to specify rules on how clients of the network are able to change the state of the ledger. Chaincodes are usually defined using a popular programming language such as Java, Javascript, or Go, and if an organization through a client wants to change the state of a ledger they have to invoke one of the functions of a chaincode to start a transaction. The clients don't have access to the ledger state directly.

There are two steps involved in using the chaincode mechanism in Fabric illustrated in 3.5. First, we have to install the chaincodes on the peer, which means copying the source code of the chaincode to the peer. The next is instantiate, which starts the chaincode on the channel, this step has a channel scope. The same chaincode source can be instantiated on multiple channels. In the older versions of Fabric, each peer on a channel creates a container for the instantiated chaincode. This means creating exactly one new container in Docker. This can be very wasteful since one chaincode container could satisfy the role instead of one each for all peers. Fabric has endorsement policies for chaincode, which determines how many peers have to endorse the transaction and which peers. This can be scaled from all peers have to endorse or any peer can. If a chaincode container fails, then the peer can't endorse, this doesn't mean the system becomes unavailable but reduces redundancy. It is also not possible to have different chaincodes on the same channel on different peers. This

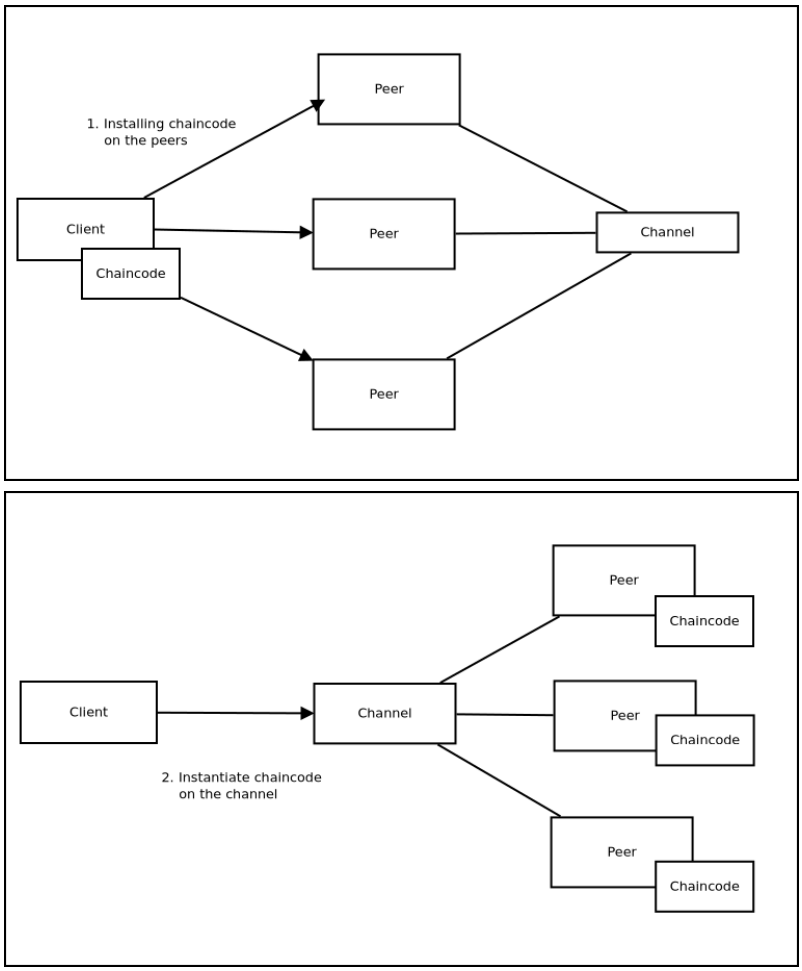

Figure 3.5: Installing and instantiating the chaincode

is a strong development dependence between organizations, and makes the service vulnerable against common mode failure since a single fault in the chaincode can result in the failure of multiple components. This problem is usually solved by software diversity patterns.

3.3 Hyperledger Fabric 2.0

Most of the new features are related to chaincode management and deployment in Fabric 2.0. The new version departs from the old in a few ways. First, in version 2.0, the peer manages the creation of chaincodes. Second, it is no longer required for all peers to have the same chaincode, with the introduction of external chaincodes.

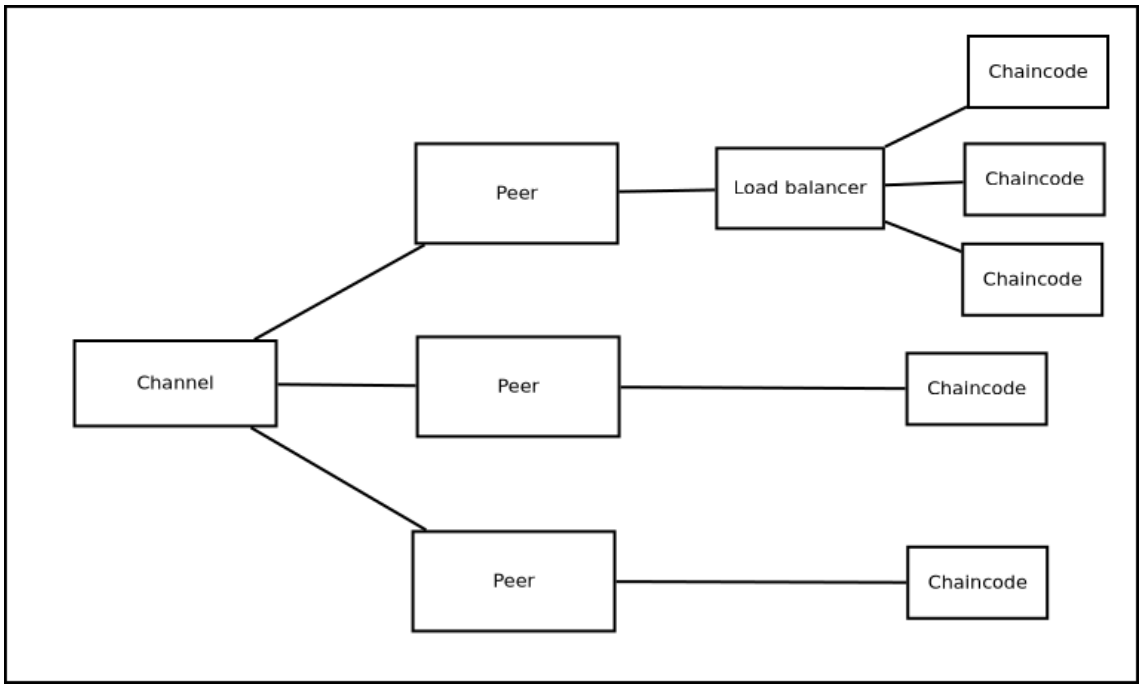

Figure 3.6: External chaincodes in Fabric 2.0

3.3.1 External chaincodes

An important development in the new Fabric version is external chaincodes. External chaincodes doesn't constrain the Fabric users to only have chaincodes in Docker containers, it allows executing chaincodes in the operator's environment of choice. It allows the chaincode to be run as an external service, so the chaincode execution becomes more separated from the peers. This means it is possible to create multiple copies of a chaincode for load balancing or increased crash fault tolerance as illustrated in figure 3.6.

Fabric v1.0	Fabric v2.0
same chaincodes on channel	allows different chaincodes
chaincode runs only in Docker	chaincode can run externally

Table 3.1: Differences between Hyperledger Fabric versions

Another feature is that chaincodes involved in a transaction don't need to be the same, they only have to return the same ledger state modification. This adds further flexibility to Fabric, and enables organizations to create their own implementations, and to fix their implementations without all participants having the need to replace their chaincodes.

3.3.2 Software diversity patterns in Fabric 2.0

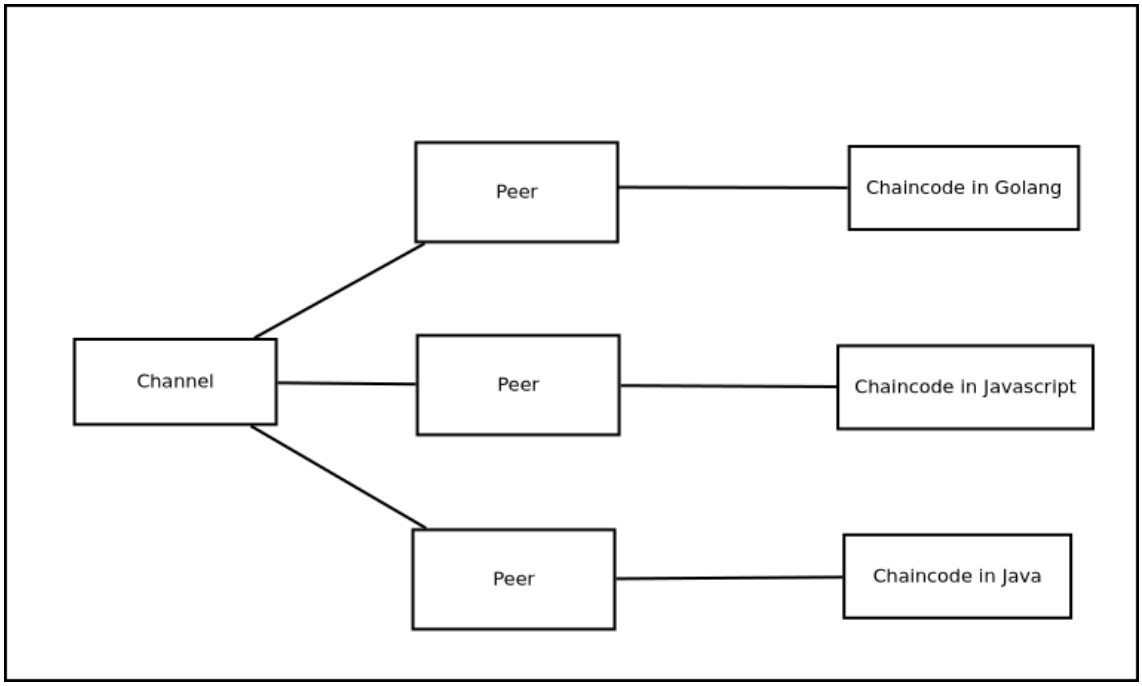

Figure 3.7: N-version diversity in Hyperledger Fabric

The new features introduced in Hyperledger Fabric 2.0 allows us to implement many different software diversity patterns. One of them is to create chaincodes for each peer that modify the ledger state in exactly the way, but have different underlying implementations as illustrated in 3.7. For example, one chaincode may be written in Golang, another implementation might be in Java, or Javascript, and they only have to match in their output when a function is invoked on the chaincodes. This is advantageous because it is unlikely that all chaincode implementations will contain the same bug, so if for example, we have a policy that expects all chaincodes to return the same result, we can be confident that we will not introduce faulty states into the ledger. Unfortunately, it is not yet possible to take advantage of this feature yet, since currently, it is only possible to create Golang chaincodes in 2.0 as external service, but other languages will be added soon.

Fabric 2.0 also encourages natural diversity, since every organization participating in the network can create their implementation independently, spontaneously, without previously agreeing to each other what the chaincode implementation should be. The other organization only have to agree on whether they accept the implementation or not.

Another option is to use the flexibility provided by external chaincodes to set up different configurations of peers and chaincodes. We take into our hands the chain-

code deployment process and the way we pair peers with chaincodes. We can create a configuration where all peers use a single address for executing code on external chaincodes, but behind this address, there are multiple copies of the diverse chaincode implementations. This could make the entire system more fault-tolerant, since if one chaincode container fails, others can take its place, and the failed containers can be restarted, so a fixed number of replicas always remain in operation. Moreover, it makes the system have better performance since if the chaincode, in general, is too busy handling transactions, we can have a load balancer with replicas behind it, so it is easy to scale the computation capacity if necessary. These possibilities make the entire Hyperledger Fabric transaction mechanism more robust and making Fabric better suited for its users.

Chapter 4

Measurements

Now that we introduced software diversity and Hyperledger Fabric, we are ready to implement software diversity patterns in Hyperledger Fabric and test the performance of the network. We are going to do three measurements, the first one is going to be the baseline, and we are going to implement the chaincode in the old version of Hyperledger Fabric. Next, we are going to do the same setup but with the new version of Hyperledger Fabric. Finally, we are going to create a setup with two chaincode for each org participating in the network. The goal is to compare the performance of the setups and test whether the new setups will provide better performance.

4.1 The test infrastructure

Our test infrastructure runs on 19 virtual machines and has two high-level components: the SUT (system under test) and the workload generator. All components on the test infrastructure are deployed in Docker containers, on a Docker swarm cluster.

4.1.1 Docker Swarm cluster

Docker swarm is a tool that enables us to deploy containers on a network that can communicate with each other, even though they might physically exist on different virtual machines. Containers provide a small execution environment so that we are able to run services under such containers and enable us to create our test infrastructure and perform our measurements.

4.1.2 Fabric topology

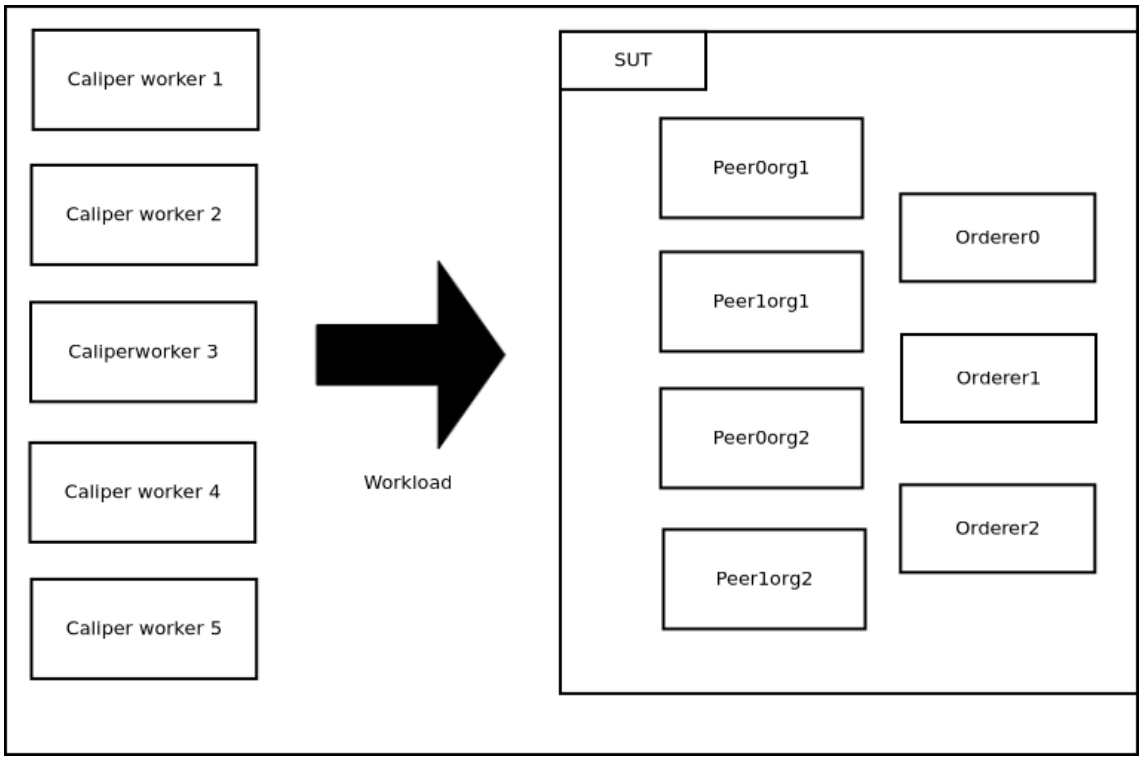

Figure 4.1: The test infrastructure

The network, excluding the chaincodes, has two organizations, and each organization has two peers. The network also has three orderers, so that the system has redundancy as far as the ordering service goes. The network topology is illustrated in Figure 4.1, as the SUT part of the diagram.

4.1.3 Workload characteristics

The tool we are going to use for our measurements is called Caliper[26], which enables us to run arbitrary workloads agains a Hyperledger Fabric deployment. It allows us to create rounds of transactions and set the transaction per second (tps) to bombard the network with the required rate of requests. Caliper, as a result, returns how many transactions succeed and how many fail, what is the average, minimum and maximum latency, and what is the effective tps of the network.

Caliper has five worker containers in the test infrastructure, they provide the workload for the measurements. This illustrated in Figure 4.1.

We are going to use the same workload setup in all three cases. The setup involves an initialization round and six transaction rounds, first round with 100 tps, sixth

round with 200 tps, and all other rounds progress from 100 tps to 200 by 20 tps per round. These transaction rounds are going to use the same chaincode function, so we can compare the results.

We are using a sample chaincode called marbles for our tests. We are going to use a function of the chaincode that performs range queries on the ledger and iterates on the results, so this operation is costly enough to invoke and suits our measurements.

4.2 Measurement setup and results

4.2.1 Measurement setup

We have three different measurement setups. First is M1, it is the setup with local chaincodes, second is called M2, this is the setup with the shared external chaincodes, and finally we have M3, the setup with separate external chaincodes. M1, M2, M3 are shorthand notations for measurement one, two and three.

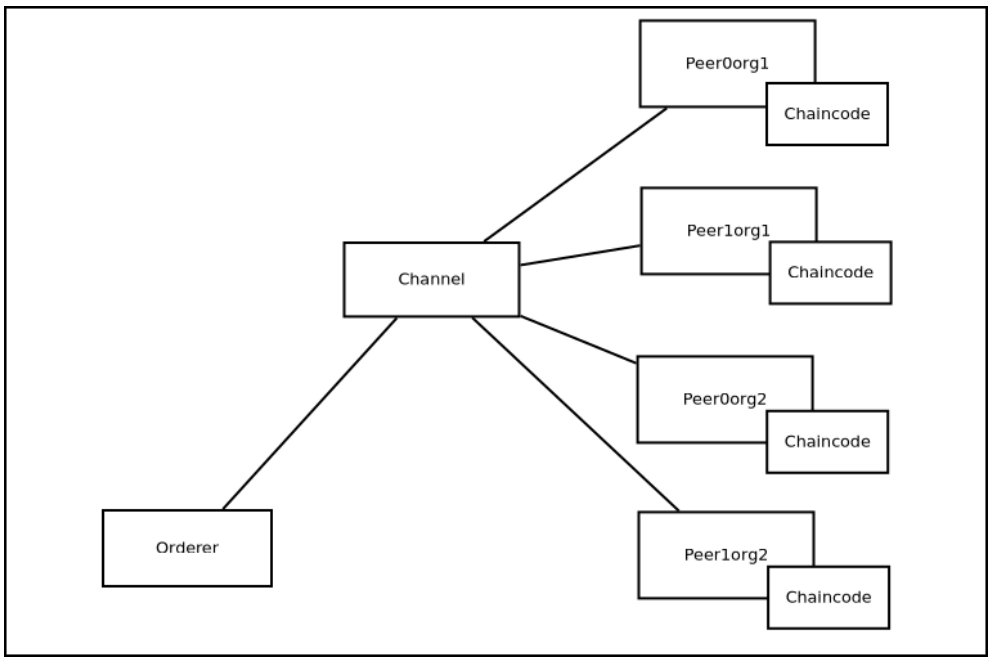

Figure 4.2: Measurement setup for the first version of Fabric

We perform the measurements on three measurement setups. The first one is performed on the 1.4.7 version of Fabric. This setup, like all the others, has three orderers for ordering the transactions, and four peers, two peers each for the two organizations of the network. The chaincodes of this organization are configured in the only way possible for Fabric v1, they are Docker containers, and all peers have the same chaincode attached to them. This example is illustrated in Figure 4.2.

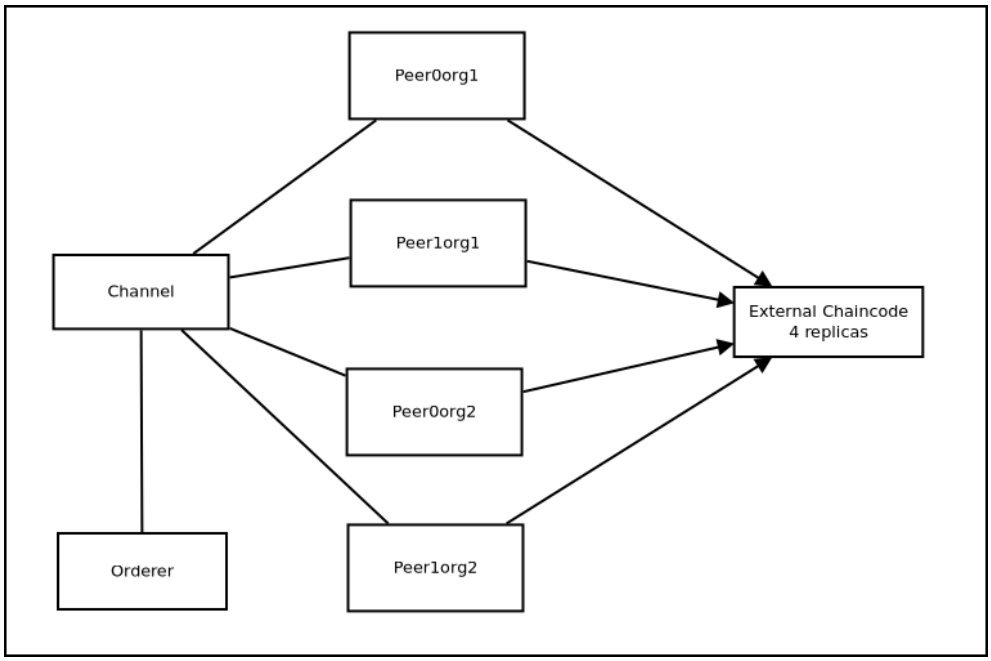

Figure 4.3: Hyperledger v2.0 measurement with the same chaincode replicated four times

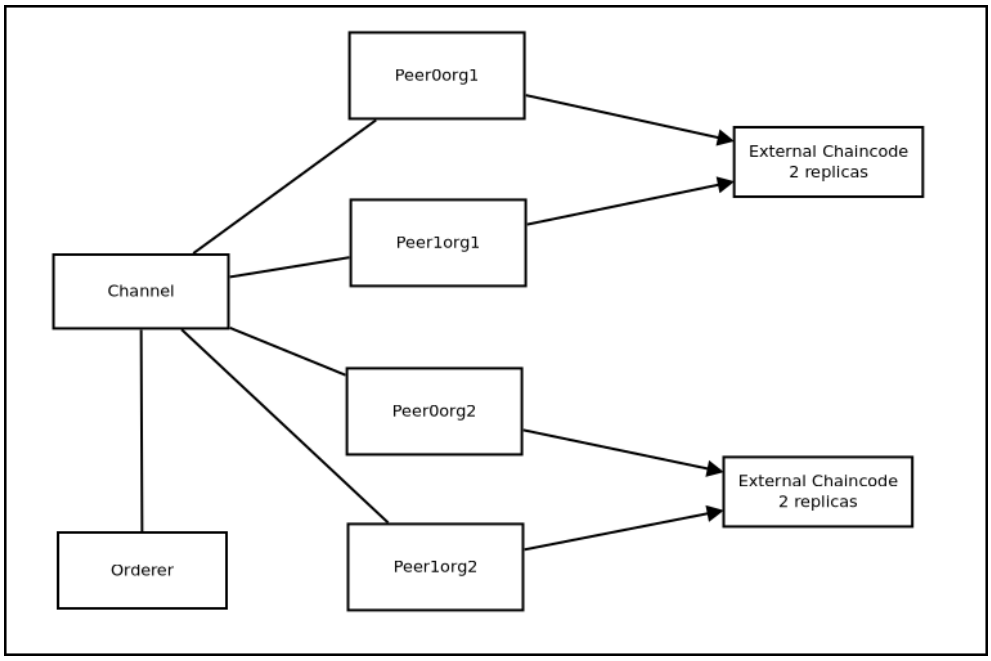

Figure 4.4: Hyperledger v2.0 measurement with two different chaincode for each organization

The second setup uses the new Hyperledger Fabric's external chaincode mechanism. The chaincodes are not attached to the peers anymore, and they don't have to be Docker containers, but the situation is similar to version 1 because all chaincodes are the same. So this measurement tries to compare the old version's performance with the new, using a similar setup of chaincode services. The external chaincodes are

behind one service in this case, and we create four replicas of Docker containers as external chaincodes. Docker Swarm ensures that round-robing style load balancing is performed between the replicas when a peer accesses them. This example is illustrated in 4.3.

The third setup also uses the new Hyperledger Fabric, but instead of using the same chaincodes everywhere, we introduce software diversity to the measurements by allowing each organization to have its own chaincode that is different from the other's chaincode. This setup has two replicas of the same chaincode for each organization. We might be able to further refine this setup by having different implementations inside an organization.

4.2.2 Measurement results

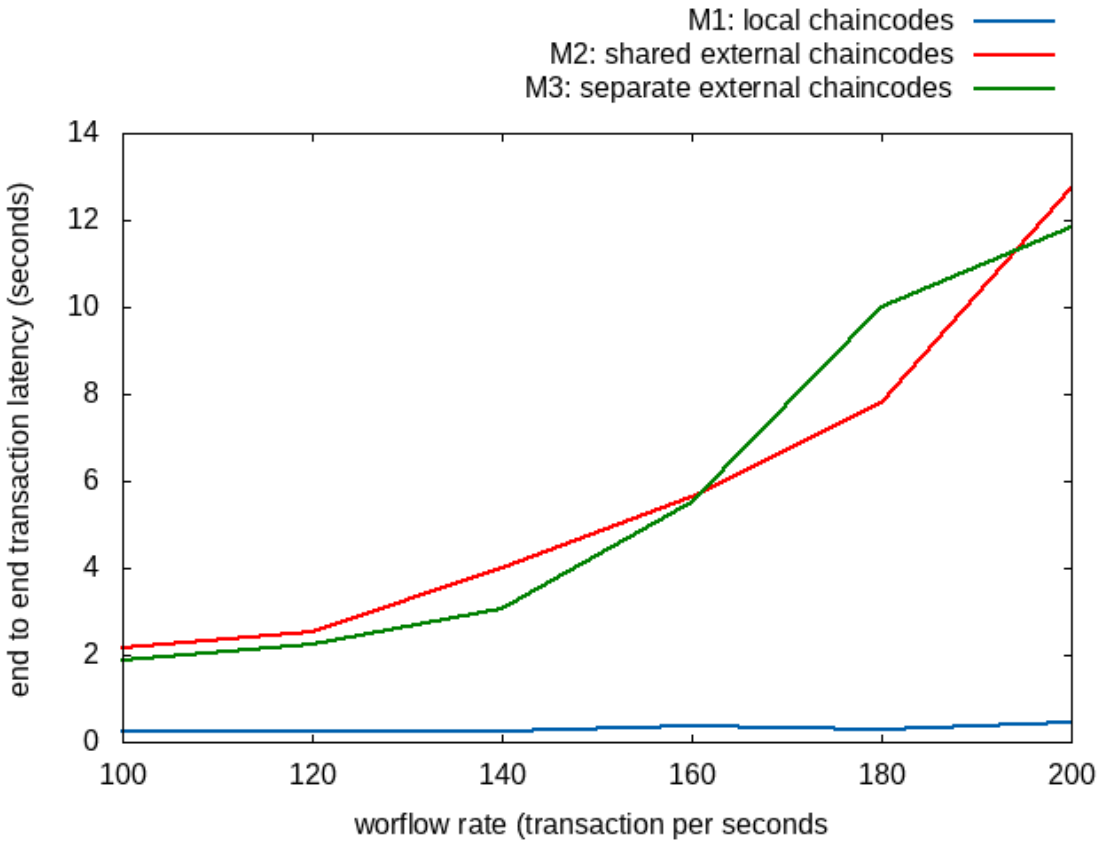

Figure 4.5: Minimum latencies of the three measurements

We created three plots from the measurement results, plot 4.5 ,4.6,and 4.7. The first plot contains the minimum latencies of each round for all three measurements, the

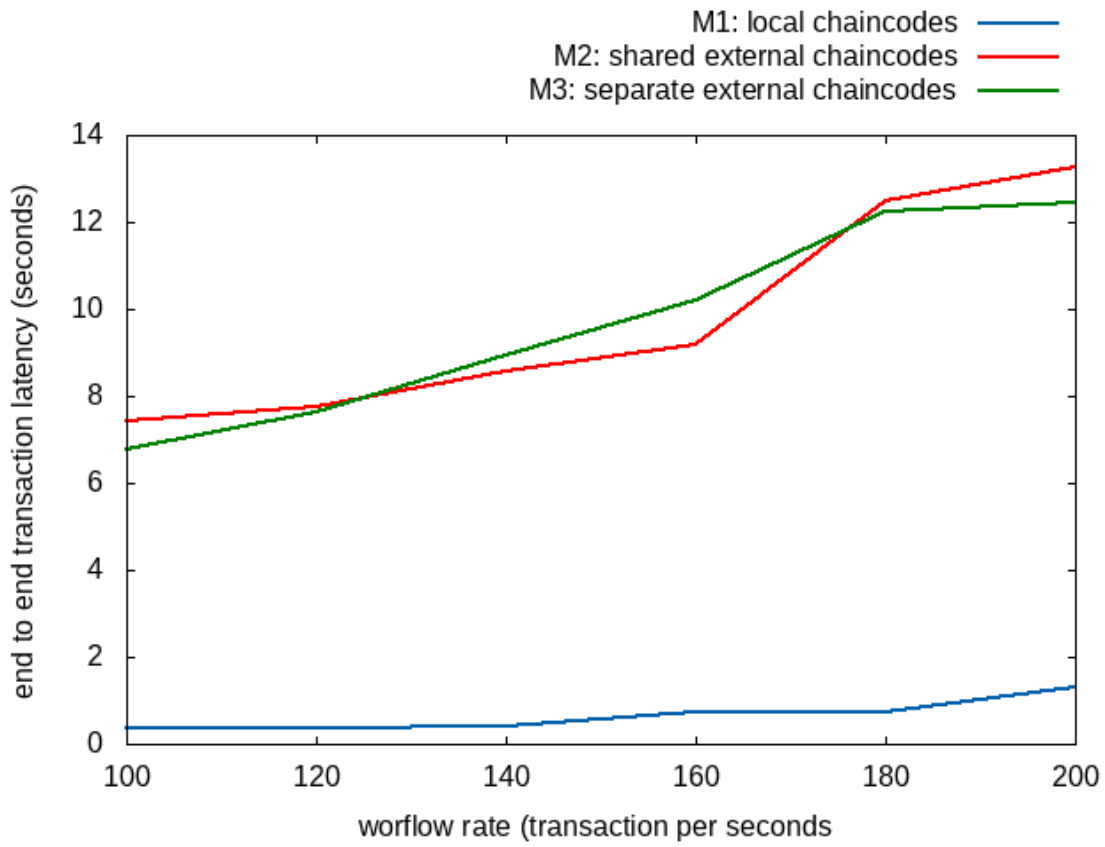

Figure 4.6: Average latencies of the three measurements

second and third contain the average and maximum latency. The table represent the same data. We can see that as we increase the number of transactions the latency increases in all cases since the network can't handle the increased traffic.

The results show, as expected, that the performance of the network is going to decrease if we use external chaincodes. The reason for this is slower connection between the chaincodes in the new version. This is the cost of having chaincode using external chaincode and the ability to run chaincodes externally.

The data shows that the external chaincodes are more sensitive to changes in Further measurements are required for a more precise characterization of the relation between the workload rate and the performance metrics. However, such detailed sensitivity analysis is left as future work.

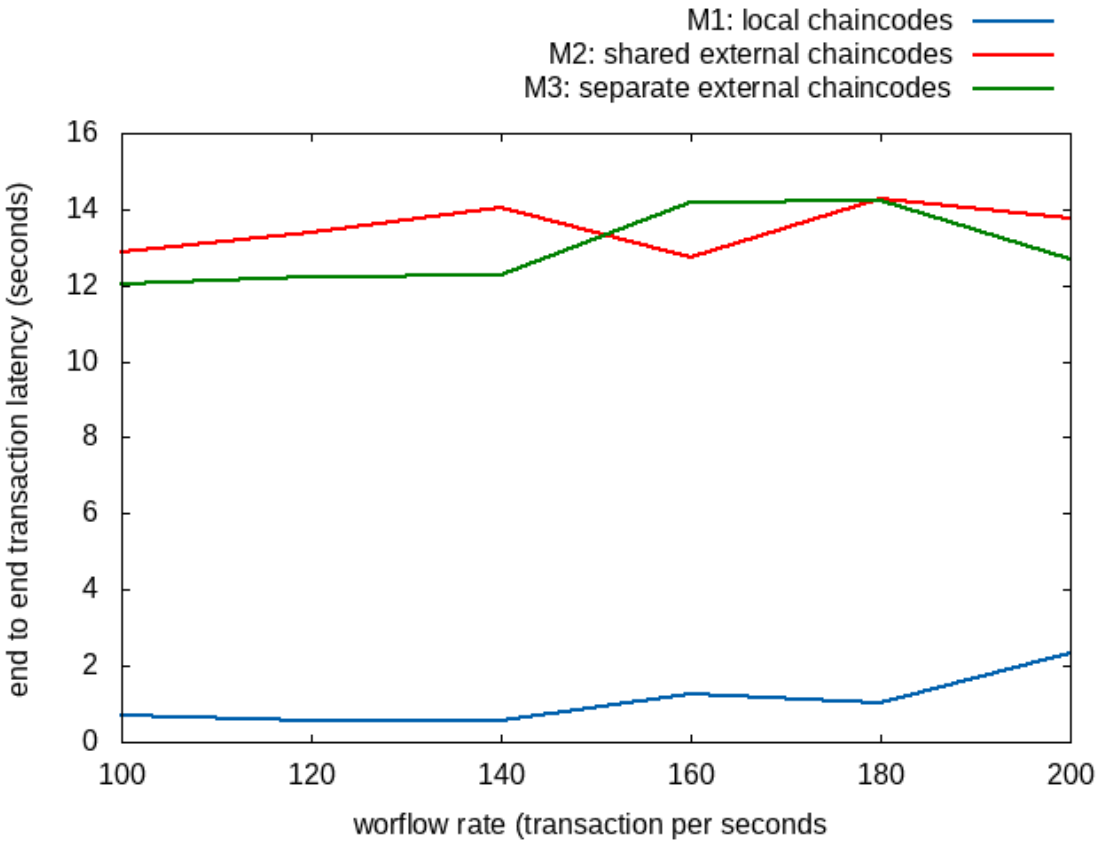

Figure 4.7: Maximum latencies of the three measurements

	minimum latency			average latency			maximum latency		
TPS	M1	M2	M3	M1	M2	M3	M1	M2	M3
100	0.24	2.15	1.87	0.38	7.43	6.77	0.71	12.86	12.04
120	0.25	2.55	2.23	0.38	7.74	7.64	0.56	13.37	12.20
140	0.26	3.98	3.05	0.42	8.59	8.94	0.56	14.03	12.27
160	0.38	5.65	5.52	0.72	9.20	10.20	1.26	12.75	14.17
180	0.28	7.80	9.99	0.75	12.49	12.25	1.01	14.27	14.22
200	0.44	12.72	11.85	1.29	13.28	12.43	2.31	13.75	12.70

Table 4.1: Measured latencies by transactions per second

Chapter 5

Conclusion and future work

This thesis explored the various use cases of software diversity for fault-tolerance. Then discussed the new version of Hyperledger Fabric and its new smart contract mechanism, and how software diversity patterns might provide fault tolerance to the system. Finally, we established a Hyperledger Fabric network, and performed several measurements comparing the old Fabric version with the new, and created a network that enables us to use software diversity patterns to the smart contract mechanism of Hyperledger Fabric.

There are multiple possibilities for continuing the work done in this thesis. First, we might measure chaincodes that are implemented in different languages, not just in Go, and analyze their performance. This enables us to introduce more diversity, and hopefully better fault tolerance into the system. It is also possible to introduce several bugs to the chaincodes, to test how well can the system tolerate them.

We created the conditions for using software diversity, but we didn't have the chance to modify the chaincodes to test software diversity in Hyperledger Fabric, but we tested external chaincodes and created a network setup that is easy to modify for implementing software diversity patterns in Hyperledger Fabric in the future.

we might test chaincode implementations used in real-world examples with our setup, and ensure they perform according to their specification. This could also provide us with valuable real world data, as opposed to the artificial examples we performed measurements on.

We can also create a detailed transaction and resource analysis examining external chaincodes. This includes the sensitivity analysis mentioned above, measuring how sensitive external chaincodes are to changes in the transactions per second they have to process.

We should note that it is not necessary to have external chaincodes to enable software diversity, the new Fabric supports the old chaincode mechanism too. An interesting possibility would be to combine external chaincodes with normal chaincodes and hopefully strike an acceptable compromise between performance and robustness.

Bibliography

[1] Hasib Anwar. Public vs private blockchains: how do they differ. *101 Blockchains*, 03 2020. https://101blockchains.com/public-vs-private-blockchain/.

[2] Karl Wüst Vasileios Glykantzis Arthur Gervais, Ghassan Karame. On the security and performance of proof of work blockchains, 2016. https://www.researchgate.net/publication/309451429_On_the_Security_and_Performance_of_Proof_of_Work_Blockchains.

[3] Algirdas Avizienis. The methodology of n-version programming, 1995. https://www.researchgate.net/profile/Algirdas_Avizienis/publication/200031514_The_Methodology_of_N-Version_Programming/links/00b49539a3cd7be0af000000/The-Methodology-of-N-Version-Programming.pdf.

[4] Martin Monperrus Benoit Baudry. The multiple facets of software diversity: Recent developments in year 2000 and beyond, 2000. https://dl.acm.org/doi/pdf/10.1145/2807593.

[5] Vitalik Buterin. Ethereum: A next generation smart contract and decentralized application platform. https://whitepaper.io/document/5/ethereum-whitepaper, 2013.

[6] Docker. Docker swarm documentation. https://docs.docker.com/engine/swarm/, 2020.

[7] Robert Hanmer. *Patterns for Fault Tolerant Software*. John Wiley and sons Ltd, 2007.

[8] Matti A. Hiltunen. Survivability through customization and adaptability: the cactus approach, 2000. https://www.researchgate.net/profile/Matti_Hiltunen2/publication/3837716_Survivability_through_customization_and_adaptability_

the_Cactus_approach/links/00b7d518934bb2b3ac000000/Survivability-through-customization-and-adaptability-the-Cactus-approach.pdf.

[9] Hyperledger. Hyperledger fabric documentation for external chaincodes. https://hyperledger-fabric.readthedocs.io/en/release-2.0/cc_launcher.html, 2020.

[10] Hyperledger. Hyperledger fabric documentation for external chaincodes. https://hyperledger-fabric.readthedocs.io/en/release-2.0/cc_service.html, 2020.

[11] Hyperledger. Hyperledger fabric introduction. https://hyperledger-fabric.readthedocs.io/en/release-2.0/whatis.html, 2020.

[12] Hyperledger. What's new in hyperledger fabric version 2.0. https://hyperledger-fabric.readthedocs.io/en/release-2.0/whatsnew.html, 2020.

[13] Hyperledger. Starting files for the measurements. https://github.com/hyperledger/fabric-samples, 2020.

[14] IBM. Ibm food trust. https://www.ibm.com/blockchain/solutions/food-trust, 2020.

[15] Barbara Liskov Miguel Castro. Practical byzantine fault tolerance, 1999. https://www.usenix.org/legacy/events/osdi99/full_papers/castro/castro_html/castro.html.

[16] Bobbi Muscara. Hyperledger fabric use cases. https://wiki.hyperledger.org/display/LMDWG/Use+Cases, 2020.

[17] Sathoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2018.

[18] John C. Knight Paul E. Ammani. Data diversity: An approach to software fault tolerance, 1988. https://pdfs.semanticscholar.org/cc9e/0993beb6400ee614826cded2b409278974ae.pdf.

[19] Toshendra Kumar Sharma. Hyperledger fabric - top use cases. *Blockchain Council*, 11 2019. https://www.blockchain-council.org/blockchain/hyperledger-fabric-top-use-cases/.

[20] IBM Research Editorial Staff. Behind the architecture of hyperledger fabric. *IBM*, 02 2018. https://www.ibm.com/blogs/research/2018/02/architecture-hyperledger-fabric/.

[21] Chen TY; Kuo FC; Merkel RG; Tse TH. Adaptive random testing: The art of test case diversity, 2010. http://hdl.handle.net/10722/89054.

[22] Mischa Tripoli. Centralized vs decentralized networks. https://www.researchgate.net/figure/Traditional-centralised-ledger-and-a-distributed-ledger_fig1_327867089, 2018.

[23] Docker. Docker documentation. https://www.docker.com/, 2020.

[24] Docker2. Docker compose documentation. https://docs.docker.com/compose/, 2020.

[25] Klenik Attila. Cluster admin tool https://www.npmjs.com/package/@klenik/cluster-admin, 2020.

[26] Hyperledger. Caliper https://www.hyperledger.org/use/caliper, 2020.

[27] IEEE. Standard Glossary of Software Engineering Terminology. *IEEE*, 12 1990. https://ieeexplore.ieee.org/document/159342.