Evaluation of Blockchain Technologies for Connected Cars

Politecno Di Torino - Master's Thesis

Candidate: Mlađo Milunović

Advisors: Carla Fabiana Chiasserini, Paolo Giaccone

April 1, 2019

Contents

Chapter 1

Introduction

The goal of this thesis is to design and test a blockchain network for storage, verification, and analysis of Cooperative Awareness Messages generated by motor vehicles. Cooperative Awareness Message(CAM) is a message format defined in [1] which is a part of the Intelligent Transport System [2]. Each vehicle broadcasts periodically a CAM message for all vehicles to see and collect, and each vehicle uploads it's own and received CAMs to the network. The network is a collection of eNodeB stations and we propose for the network to be implemented with the blockchain technology. The uses of such network are many: determining the guilty party in case of a crash, insurance analysis, traffic analysis, etc. Vehicles can insert false information into their CAMs(either maliciously or erroneously). To accommodate this possibility the network will provide a way to verify the correctness of a CAM. This is done through position verification algorithms, which compare the properties of cars that received the CAM with the values found inside the CAM and checks if they are physically plausible.

Chapter 2

Description of Cooperative Basic Service within the Intelligent Transport Systems Architecture

2.1 Introduction

Intelligent Transport Systems (ITS) are systems which provide information and communication mechanisms that actors involved in transport(cars, trucks, buses, trains, etc.) can use to safely and efficiently use the transportation infrastructure. Cooperative Basic Service (CA basic service) is one of the foundational services of ITS whose goal is to provide cooperative awareness. Cooperative awareness(CA) in this context means that road users and roadside infrastructure are informed about each other's position, dynamics, and attributes [1]. Road users can be cars, trucks, buses, and even pedestrians; essentially anyone who uses the roads. Road infrastructure are the non moving members involved in transportations, such as: traffic lights, signs, barriers, gates, etc. [1] Cooperative awareness is the foundation for many road safety and traffic efficiency applications, such as: ADD EXAMPLES. CA is achieved through a regular exchange of information between road users and roadside infrastructure. Road users are also called vehicles. This regular exchange can happen between:

1. Vehicles to Vehicles (V2V messages)
2. Vehicles to Infrastructure (V2I messages)
3. Infrastructure to Vehicles (I2V messages)

The information exchange happens through a wireless network named V2X network [1].

The information to be exchanged is packed in periodically transmitted Cooperative Awareness Messages(CAMs). The generations, management, and processing of CAMs is the job of the CA basic service. The CA basic service is a mandatory facility layer for all kinds of ITS-Stations(ITS-S), which take part in the road traffic [1]. The focus of this chapter is on the CAM specification for vehicle ITS-S which participate in the V2X network.

2.2 Services Provided by CA basic service

Ca basic service is a facilities layer withing the ITS architecture whose purpose is to operate the CAM protocol. It provides two services: sending CAMs and receiving CAMs. To disseminate CAMs CA basic service uses entities of the ITS networking & transport layer.

2.2.1 Sending CAMs

The sending of a CAM includes the generation and transmission of the CAM. The generation is performed by the originating ITS-S, after which the CAM is given to the ITS networking & transport layer for dissemination. The dissemination of CAMs may depend on the applied communication system. In our context we will consider the ITS-G5A network[EN 302 663], where CAMs are disseminated from the originating ITS-S to all ITS-S in the direct communication range. The range can be manipulated by the originating ITS-S through transmission power modifications. CAMs are generated periodically with a frequency controlled by the CA basic service in the originating ITS-S. To generation frequency depends on the ITS-S status, *e.g.* sudden change of speed and/or position, as well as on the congestion of the channel.

2.2.2 Receiving CAMs

Upon receiving a CAM, the CA basic service makes the content of the CAM available to the ITS applications and/or to other facilities withing the receiving ITS-S.

2.3 CAM Dissemination

2.3.1 CAM dissemination requirements

Point-to-multipoint communication is used for transmitting CAMs. The CAM is transmitted only from the originating ITS-S in a single hop to the

receiving ITS-S that is in direct communication range from the originating ITS-S. A received CAM cannot be forwarded to other ITS-Ss.

2.3.2 CA basic service activation and termination

As long as the CA basic service is active, the CAM generation is triggered and managed. The activation of the service varies for different types of ITS-S. For vehicle ITS-S, the CA basic service is activated with the ITS-S activation. The CA basic service is terminated when ITS-S is deactivated.

2.3.3 CAM generation frequency

Cam generation frequency defines the interval between two consecutive CAM generations. The frequency is defined and managed by the CA basic service. The limits of the generation frequency depend on the type of ITS-S. For vehicle ITS-S:

- Minimum time between two consecutive CAM generations is 100ms. This corresponds to a generation frequency of 10 Hz.
- Maximum time between two consecutive CAM generations is 1000ms. This corresponds to a generation frequency of 1 Hz.

The actual generation frequency depends on the dynamics of the originating ITS-S and congestion of the communication channel, but it's always between the defined limits.
For road side units(RSUs) ITS-S the frequency is defined in such a way, that at least one CAM is transmitted while a vehicle is in the communication zone of the RSU ITS-S. The time interval is always greater or equal to 1000ms(1 Hz). The probability for the reception of a CAM from an RSU by a passing vehicle depends on the generation frequency of the CAM and the time the vehicle is within the communication range, which depends on the vehicle speed and the RSU transmission power.

2.3.4 CAM time requirements

The time when the data reported in the CAM is taken is crucial for the applicability of that data in the receiving ITS-Ss. For this purposes, each CAM is timestamped. Of course, we expect a reasonable time synchronization between the different ITS-Ss.
The time required for CAM generation is guaranteed to be less than 50ms. This time refers to the time difference between time at which CAM generation

is triggered and the time at which the CAM is delivered to the networking & transport layer For the CAM timestamp, the following requirements are satisfied:

- For the vehicle ITS-S, the timestamp corresponds to the time instant at which the position of the ITS-S reported in the CAM was determined
- For the RSU ITS-S, the timestamp is the time of generation
- The difference between CAM generation time and the time stamp is guaranteed to be less than 32767ms.

2.3.5 Security Constraints

The security mechanism for ITS allow authentication of messages transferred between ITS-Ss with certificates. A certificate specifies the permissions and privileges of the certificate owner to send certain types of messages and optionally privileges for specific data elements within these messages. FIX!!!!!The format for security envelope is defined in [TS 103 097]
Inside the certificate the permissions and privileges are indicated by a pair of identifiers, ITS-AID and SSP. The ITS-Application Identifier(ITS-AID)(specified in TR 102 965) indicates the overall type of permissions granted: *e.g.* indicates that the owner is allowed to send CAMs. The Service Specific Permission(SSP) is a field that indicates special permissions withing the permissions indicated by the ITS-AID: *e.g.* indicates that the owner is allowed to send CAMs for a specific vehicle type.
An incoming signed CAM is accepted by the receiver if the certificate is valid and the CAM is consistent with the ITS-AID and SSP in its certificate.

2.4 CAM Format Specification

2.4.1 General structure of a CAM

A CAM is composed of one common ITS PDU header and multiple containers. The ITS PDU header contains information of the protocol version, the message type, and the ITS-S ID of the originating ITS-S. For vehicle ITS-Ss, a CAM includes:

- One basic container: contains essential information, such as the type of ITS-S
- One high frequency container: contains dynamic information of the originating ITS-S, such as speed, heading, acceleration

- May include one low frequency container: contains static and not highly dynamic information of the originating ITS-S, such as vehicle lights
- May include one or more special containers: contains information specific to the vehicle role of the originating vehicle ITS-S

All CAMs generated by a RSU ITS-S include a basic container and optionally more containers.

Each container is composed from a set of mandatory and a set of optional fields. The full specification can be found at [ETSI EN 302 637-2].
Since the format contains many mandatory fields, in the context of this thesis we will use and define a sub format containing only a subset of the fields. The fields that will be used are the following

- VehicleId: this field is the ITS-S ID field taken from the ITS PDU header. It is randomly generated and changes periodically for privacy reasons.
- Timestamp: field representing the instant when the CAM information was calculated
- Position: longitude and latitude of the vehicle
- Heading: expressed in degrees
- Speed
- Acceleration

2.4.2 Reports

The information required to perform position verification of CAM messages:

1. the CAM message itself
2. the reception information from other vehicles that received the CAM. Reception information contains physical characteristics of the receiving vehicles: time of reception, position, speed, etc.

A vehicle needs, in addition to the generated CAMs, to upload the reception information for every received CAM. In the context of this thesis a data structure called CAM Report is defined to allow uploading all the necessary information. Whenever a vehicle generates a CAM message it also generates a CAM Report. A CAM Report contains the generated CAM and the

reception information for all CAMs the vehicle received since the last CAM Report. Furthermore, each CAM Report contains cryptographic information that links it to the previous report making the reports tamper-proof.

CAM Report Format Specification

ReportId	A string that uniquely identifies the report derived from the generated CAM. It is obtained from the concatenation of the vehicle ID and the timestamp of the generated CAM.
GeneratedCam	The CAM message generated by the reports' owner.
GeneratedCamHash	
ListReceivedCams	A list containing reception information for each received CAM since the last report.
PreviousReportId	The id of the previous report generated by this report's owner.
PreviousReportHash	The hash of the previous report generated by this report's owner.

In the **ListReceivedCams** each member consists of:

ReceivedCamId	The ID of the received CAM
ReceivedCamHash	The hash of the received CAM
ReceptionInfo	Physical characteristics of the report owner in the time instant when the CAM was received

Each **ReceptionInfo** consists of the following fields:

ReceptionTimestamp	The time instant in which the report owner has received the CAM
ReceptionLatitude	The latitude of the receiver in the time instant specified by ReceptionTimestamp
ReceptionLongitude	The longitude of the receiver in the time instant specified by ReceptionTimestamp
ReceptionSpeed	The speed of the receiver in the time instant specified by ReceptionTimestamp
ReceptionAcceleration	The acceleration of the receiver in the time instant specified by ReceptionTimestamp
ReceptionHeading	The heading of the receiver in the time instant specified by ReceptionTimestamp. Represent with respect to the North.

Tamper-Proof properties of CAM Reports

In the structure of the CAM Reports there are two pieces of information present that provide the tamper-proof properties:

1. Hash of the previous CAM report generated by the same vehicle
2. A hash of each received CAM in the List of Received Cams

Hash of the previous CAM report generated by the same vehicle
This property borrows it's idea from the blockchain itself. Where as each report is cryptographically linked with the previously generated reports. This following figure illustrates the property:

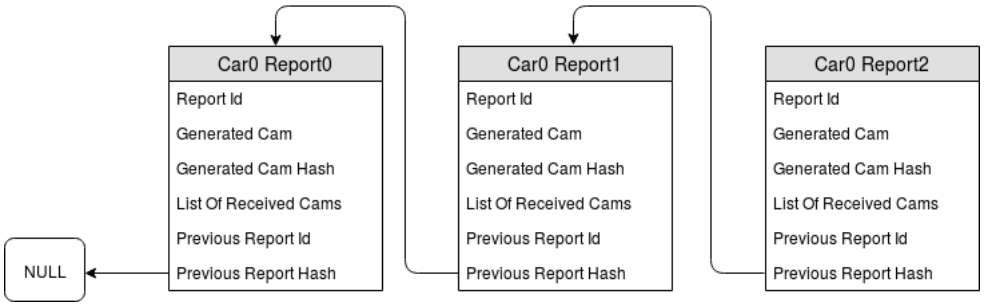


Figure 2.1: First three reports generated by a car with the id Car0

The first generated report by a car is called **Genesis Report**. Since there is no reports generated before, it's previous report hash and previous report id are set to null. After that each new report contains a hash of the previous one.
This property has two effects on the security of the reports:

1. It is more difficult for a vehicle to pretend to be another vehicle since it needs to have the last generated report of the original vehicle
2. It is more difficult for a vehicle to change the report after it has already been generated. Changing a report would imply that a vehicle needs to change all the reports that are generated after the report in question in order to match the hashes. This can easily be identified by a presence of a "fork" in the vehicle blockchain.

A hash of each received CAM
Whenever a car is generating a report it contains a list of the CAMs received since the last generated report. Instead of saving the whole received CAM a vehicle stores a hash of the received CAM.
This introduces the following security properties:

- The car doesn't store the whole received CAM. That it means it can't arbitrarily modify specific values. The only possible attack is to modify the hash, but that would not make the original CAM invalid since the hashes are checked before the position verification.
- It makes modifying a CAM inside a report after the report has been uploaded even more difficult. Changing the CAM inside a report would lead to the CAM hash being different than the values stored by other vehicles which received the CAM. This can be easily identified by comparing the hashes.

Tamper Proof Summary Illustration

2.4.3 Position Verification Algorithm

The position verification algorithm implemented in this thesis can be explained as following: To verify a given CAM, we take all the reports containing this CAM and extract the related reception information. For each report we take the reception information and information reported in the CAM and perform the verification. The verification means, for example checking if the receiver was in the transmission range of the sender, etc. The verification process return eithers valid or invalid. Each CAM will be associated with two fields: ValidVotes and InvalidVotes, which show how many other vehicles have declared this CAM as valid or invalid, respectively. How many votes needed, and how much the difference between ValidVotes and InvalidVotes to consider a CAM valid depends on the client's use case.

Chapter 3

Fabric

3.1 Introduction

Hyperledger fabric is an open source, private, and permissioned distributed ledger technology platform established under the Linux Foundation [3]. Like other blockchain technologies, it has a ledger, uses smart contracts, and is a system by which participants manage their transactions. It provides some differentiating capabilities with respect to other blockchain technologies.
Fabric emphasizes modularity by providing following modular components:

- A pluggable *ordering* service that allow defining different consensus protocols based on the user's requirements
- A pluggable *Membership Service Provider* to allow associating entities in the network with cryptographic identities
- A pluggable *Endorsement and Validation Policy* to allow to define different policies independently for each application
- A pluggable *State Database* that allows to use different DMBS models for efficiently accessing the blockchain data.

Fabric is a permissioned network where all the participants are known to each other. Therefore the network can operate on the partial trust that exists between the participants, such as a legal agreement or framework for handling disputes.
For this reason Fabric can leverage consensus protocols that do not require native cryptocurrency for providing incentives. The absence of cryptographic mining operations means that the platform can be deployed with roughly the same operational cost as any other distributed system.

3.2 Channels

Channels are used to provide private ledgers between a subset of participants in the blockchain network. Channels provide an efficient sharing of infrastructure while maintaining data and communications privacy. Each channel represents a logical blockchain network, *i.e.* each channel has its own ledger. Only the participants that are members of the channel can modify (according to established policies) and access the ledger. This is an especially important property if we want to transact with some participants without the knowledge of others in the network(such as offering a special price to the competitor).

3.3 Nodes

In hyperledger fabric two node types are used to construct and operate a blockchain:

1. Peers
2. Orderers

3.4 Peers

Peers are the fundamental building blocks of a Hyperledger Fabric Network. Peers hosts ledgers and smart contracts. All the interaction with the ledgers must be done through peers.
A peer contains a copy of the ledger for each channel that it belongs to, meaning that a peer can host multiple ledgers.
If a client wishes to perform a transaction, the client needs to interact with a peer by invoking the smart contract previously installed on the peer.

3.5 Distributed Ledger

A ledger records all the transactions executed in the channel.

3.5.1 Assets

In Fabric all transactions modify properties of objects called **Assets**.
Each asset consists of three parts:

1. A unique identifier: **Key**

2. A **Value**: modeled either in binary or JSON encoding
3. **Version**: a counter which is incremented each time the asset is updated

3.5.2 Ledger Structure

In Fabric, the ledger consists of two related parts [4]5:

1. The World State: holds latest value for all defined assets
2. The Blockchain: contains all transactions executed on the channel

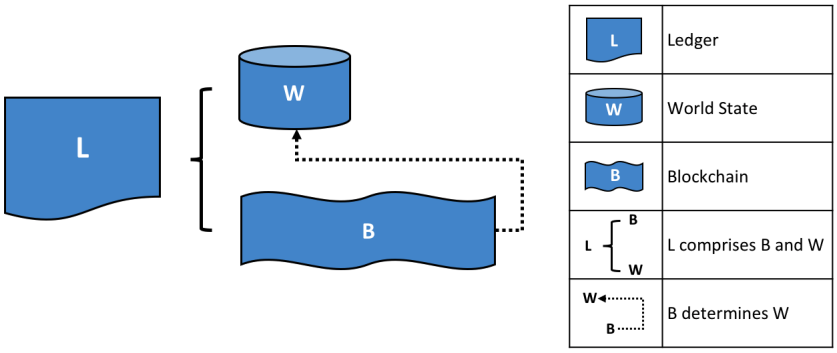


Figure 3.1: Two parts of the ledger

World State

World state stores for each asset:

- The key of the asset
- Current version of the asset
- Current value of the asset

World state is implemented as a database. This is logical since database solutions provide efficient methods of querying data.
The world state is very practical because the majority of transaction deal with the current value of the asset. If there was no world state, whenever a transaction would need to read a value of an asset, it would have to traverse the blockchain to retrieve the value.
As seen in the figure 3.1 the world state is derived from the blockchain. This

means that the world state can always be rebuilt from the blockchain. This is useful in cases such as if a peer fails abnormally, the world state can be re-generated from the blockchain on peer restart. The world state is not necessary for the fabric network to be functional; it just allows practical efficiency by acting as an asset cache.

World State Database Options

As we have already said, fabric supports two option for the database implementation: LevelDB(default) and CouchDB.
Both of the options support core operations such as getting and setting a key(asset), and querying based on keys. Keys can be queried by range and composite keys are supported.
CouchDB is an appropriate choice when ledger states are structured as JSON documents because CouchDB enables rich queries for JSON documents. This means that we can query based on the JSON field values in the asset.
To run queries based on JSON fields we need to define indexes. An index simply constructs an efficient data structure for searching based on the specified fields.

The Blockchain

Whereas the world state contains the current state for all assets, the blockchain contains all the state changes(CRUD operations) for all assets. It is a historical record of how all the assets have arrived to their current state.
The blockchain is made from a sequence of interconnected blocks. Each block contains a sequence of transactions, where each transaction represent a query or update to the world state.
Each block's header contains a hash of block's transactions. Additionally each block's contains a hash from the prior block's header. In this way, all the blocks and all the transactions withing the blocks are sequenced and cryptographically linked together.
Unlinke the world state, the blockchain is always implemented as a file instead of a database. This is reasonable since the blockchain is biased toward a very small set of operations. Appending to the end of the blockchain is the primary operation, and query is an infrequent operation(because in most cases the current asset value is used, which is stored in the world state).
The following picture illustrates the blockchain structure:

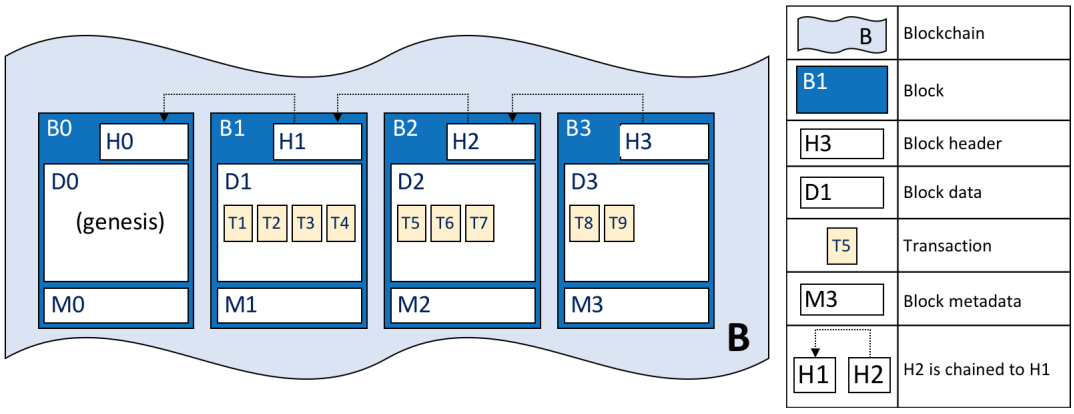


Figure 3.2: Blockchain Structure

We see that the first block in the blockchain is the so called **Genesis** block. The most important part is the links, that is the headers containing the hash of the previous block's header.

Blocks

In the figure 3.2 we see that a block in the blockchain consists of three parts, namely:

1. **Block header**: the section used for block identification, and to provide cryptographic linking between the blocks. These fields are used to ensure the immutability of the blockchain.
   Block header consists of:
   - **Block Number**: An integer starting at 0 (the genesis block), and increased by 1 fore very new block appended to the blockchain.
   - **Block Hash**: The hash of the block data(transactions in the current block)
   - **Previous Block Hash**: A copy of the hash from the previous block
2. **Block data**: Contains a list of transactions arranged in order. The order is determined by the ordering service when it creates the block. The transaction format is explained in the next section.
3. **Block metadata**: Contains the time when the block was written, as well as the certificate, public key and signature of the block writer.

The block committer(every peer in the channel) adds a valid/invalid indicator for every transaction. This indicator is not included in the block hash since the hash is calculated when the block is created.

Transactions

A transaction captures changes to the world states.
A transaction consists of the following parts:

1. **Header**: Contains metadata about the transaction, such as the information about the chaincode which was invoked by the transaction.

2. **Signature**: Contains a cryptographic signature by the client application which invoked the transaction. This is used to ensure that there was no tampering with the transaction

3. **Proposal**: Contains the set of input parameters to the chaincode. The input parameters are provided by the client application.

4. **Response**: The output of the smart contract execution. Contains the Read Write set(RW-set) which captures the before and after value of the world state through:

   - Read Set: keys and versions for all assets that were read during the smart contract execution
   - Write Set: keys and new values for all assets which were updated during the smart contract execution

   If the transaction is deemed valid, the write set will be applied to update the world state.

5. **Endorsements**: A list of signed transaction responses. This list needs to contains the signatures of all organizations required by the chaincode endorsement policy.

3.6 Smart Contracts and Chaincodes

A Smart Contract defines the executable business logic that generates new transaction that are added to the ledger. Smart contracts define the rules between different organizations. Applications invoke smart contracts to generate transactions that are recorded on the ledger.
A smart contract is defined within a chaincode. Multiple smart contracts

can be defined within the same chaincode. When a chaincode is deployed, all smart contracts within it are made available to applications.
Smart contracts typically perform CRUD operations on the world state, but can also query the blockchain.
Chaincodes can be implemented in multiple standard programming languages such as: Java, NodeJS, Go. To prevent infinite execution the fabric network has a parameter limiting the execution duration of a chaincode.
Smart contracts are hosted and executed on the peers of the network.

Endorsement

Every chaincode has an endorsement policy defines for it. The endorsement policy applies to all smart contracts withing the chaincode.
An endorsement policy specifies which organizations must sign a transaction generated by a smart contract withing the chaincode. If the endorsement policy is satisfied the transaction is considered **valid**, otherwise it is **invalid**. Only **valid** transactions update the world state, whereas both **valid** and **invalid** transactions are recorded on the ledger.

Transaction Validation Process

When a smart contract execution is invoked on the peer, a set of input parameters is provided. This set of input parameters is called **Transaction Proposal**. The peer then uses the **transaction proposal** and executes the smart contract requested by the client. The output of this execution is called the **Transaction Response**. The **transaction response** contains a **read-write set** specifying which data has been read during the execution, and which new data should be written to the ledger if the transaction is deemed valid. Note that the world state is not yet modified. It can only be modified after the transaction is validated.
The client which requested the invocation of the smart contract receives the transaction response signed by the peer that executed it. It's the client's job to collect the other required signatures specified by the chaincode endorsement policy. After the signatures have been collected the transaction is distributed to all peers in the channel.
The distributed transaction is validated in two steps by each peer in the network:

1. First, the transaction signatures are checked to see if they satisfy the chaincode endorsement policy.

2. Secondly, the read set of the transaction is compared with the current

value of the world state. This is a check to ensure that no intermediate update has happened to the values used in the execution of the transaction.

If a transaction passes both checks it is declared **valid**, otherwise it is **invalid**.

3.7 Orderers

Orderers are nodes that collect transactions executed by peers and order them into blocks. These blocks are distributed to all the peers in the channel. Orderers do not maintain a copy of the ledger and do interact directly with the ledger.

3.8 Permissioned and Private

Fabric is a permissioned platform, meaning that each participants of the network has a verifiable identity, *i.e.* there is no anonymous members.
Fabric is a private platform, meaning that only the participants of the blockchain network, more specifically of the channel[ref] can access the network.

3.8.1 Identity

Any entity that hosts or interacts with a hyperledger fabric blockchain network must have a valid digital identity. A digital identity is encapsulated in a X.509 certificate.
In fabric there are some additional attributes on top of the digital identity. The union of the digital identity and these additional attributes is called a **Principal**. Additional attributes can contain a wide range of properties, such as: actors' role, actors' organization unit, etc.
In general a principal determines all of actors' permissions over resources and access to information in the blockchain network.
For an identity to be valid it must come from a trusted authority. In Fabric, the component which is used to verify identities and govern the rules regarding identity verification is called the **Membership Service Provider(MSP)**. The default MSP implementation in Fabric uses X.509 certificates as identities, adopting a traditional Public Key Infrastructure (PKI) hierarchical model.

3.8.2 Membership Service Provider

Membership Service Provider(MSP) is a component of the fabric that turns verifiable identities into members of the blockchain network. Each organization participating in the fabric network is mapped to a unique MSP.

Membership

MSP's job is to identify which Root CAs and Intermediate CAs are trusted for each organization. This can be done in two ways:

1. Listing the identities of the members of an organization
2. Identifying which CAs are authorized to issue identities for the organization's members

In most cases a combination of both options is implemented.
An MSP can identify specific roles that an actor can play in both the scope of the organization the MSP represents(admins, members,...) and the scope of the network and channel(channel admins, writers, readers).
The MSP configuration is propagated through all the channels where the members of the organization that the MSP is representing are participating. This MSP configuration is called the **channel MSP**. On top of this, each actor maintains a **local MSP** in order to authenticate member messages outside the context of the channel and to define the permissions over particular components.
MSPs can also be used for the identification of a list of revoked identities.

Local and Channel MSPs

Local MSPs are defined for every node(peers and orderers) and user(client, admin,...). On nodes the local MSPs defines the permissions for that node. For example, on a peer node it may define who is the administrator for this peer. On users the local MSPs allows the user to authenticate himself as a member of the a channel or as the owner of a specific role in the system(*e.g.* organization admin).
Every node and user must have a local MSP defined.
Local MSPs are defined on the file system of the node or the user to which they apply. Therefore logically and physically, there is only one local MSP per node or user.

Channel MSPs define administrative and participatory rights at the channel level. Every organization that participates in a channel must have an

MSP defined for the channel. All peers and orderers in the channel share the same view of the channel MSP, and are able to authenticate the channel participants.
Channel MSP is logically defined in the channel configuration. However a copy of the channel MSP is physically maintained on the file system of every node in the channel and kept synchronized via consensus.

Chapter 4

Architecture

4.1 Physical architecture

The physical architecture is made from two components: **Vehicles** and **Base Stations**

4.1.1 Vehicles

Vehicles are data providers for the blockchain. Each vehicle sends its' data using the internet. Therefore, each vehicle must have an USIM with an enabled internet connection. The format used when uploading data is the CAM Report format specified in *section 2.4.2*.
A vehicle performs the following functions:

1. Generate and disseminate its' own CAM periodically.
2. Receive the CAMs from nearby vehicles and collect them into Cam Reports.
3. Upload the Cam Reports to the blockchain network, through base stations, using the internet connection.
4. Store a cryptographically linked list of all its' reports.

The generation, dissemination and reception of CAMs is provided by CA basic service as described in *chapter 2*. The uploading of reports is done through the USIM internet connection.

Vehicles do not host the blockchain network for the following reasons:

1. Resource requirements: each member of the blockchain needs to store a complete replica of the ledger. This puts an ever growing strain on resource requirements for each vehicle since a ledger is an append-only structure.

2. Number of vehicles: storing the ledger on a large number of vehicles implies that a transaction may take a long time since all of the vehicles have to update their ledger

3. Trust: A vehicle cannot be considered as a trusted node

Vehicles also do not upload their CAM Reports directly to the blockchain network for the following reason:

1. Transaction frequency: a vehicle generates a CAM Report every time it generates a CAM. The CAM generation frequency is between $1Hz$ and $10Hz$. Taking into the account the number of vehicles, the number of transactions would be enormous.

2. Authentication: To issue transactions each vehicle needs to posses a valid certificate and the certificate has to be verified for each transactions. This creates an enormous computational overhead.

If the vehicle is in a zone where internet connection is impossible, the reports are stored locally and uploaded as soon as the connection is restored.

4.1.2 Base Stations

Base Stations represent blockchain hosts and blockchain access points.

A base stations performs the following functions:

1. Authenticating vehicles which send the CAM Reports

2. Receiving CAM Reports from vehicles

3. Storing the CAM Reports into the blockchain

4. Hosting and maintaining the blockchain network

5. Providing an access point for authorized clients to read the contents of the distributed ledger

In the proposed architecture Base Stations will be eNodeBs of mobile operators.

E-UTRAN Node B

E-UTRAN Node B, also known as eNodeB is the element of the LTE network. It is the hardware that is connected to the mobile phone network and communicates directly through the wireless with user equipment(mobile phones, laptops with a mobile broadband adapter,...).

Mobile network operators form a conglomerate by sharing their eNodeBs and constructing the blockchain network. ENodeBs can authenticate vehicles since each vehicle sends data using through an USIM provided by the mobile network operators.

4.1.3 Putting it together

The physical architecture is illustrated by the following picture:

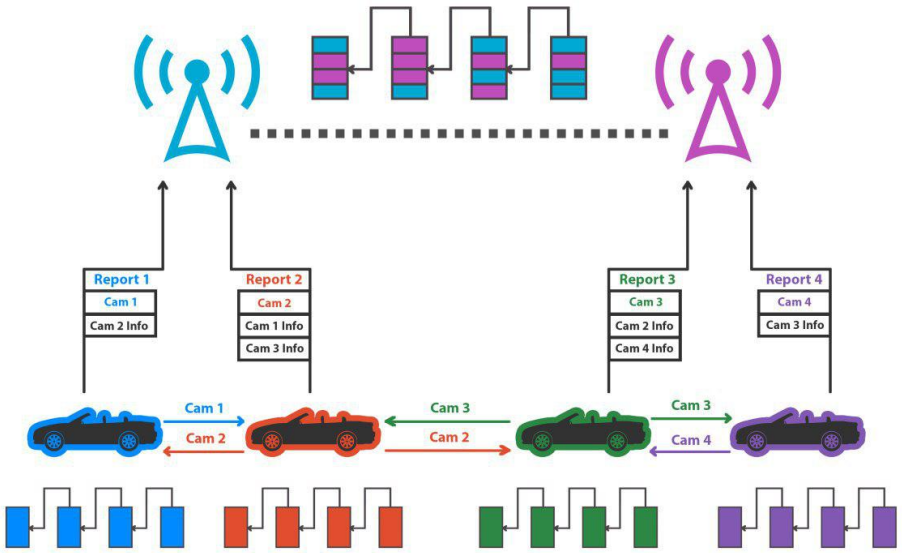


Figure 4.1: Physical Architecture

4.2 Blockchain network architecture

The goal of this section is to understand the hyperledger blockchain network architecture and main operating principles. A way to do this is to go through an example of building a network from the ground up and explaining each step along the way. In our case we have multiple organizations(mobile network providers) that will form a blockchain network which will allow storage of CAMs and any relevant information.

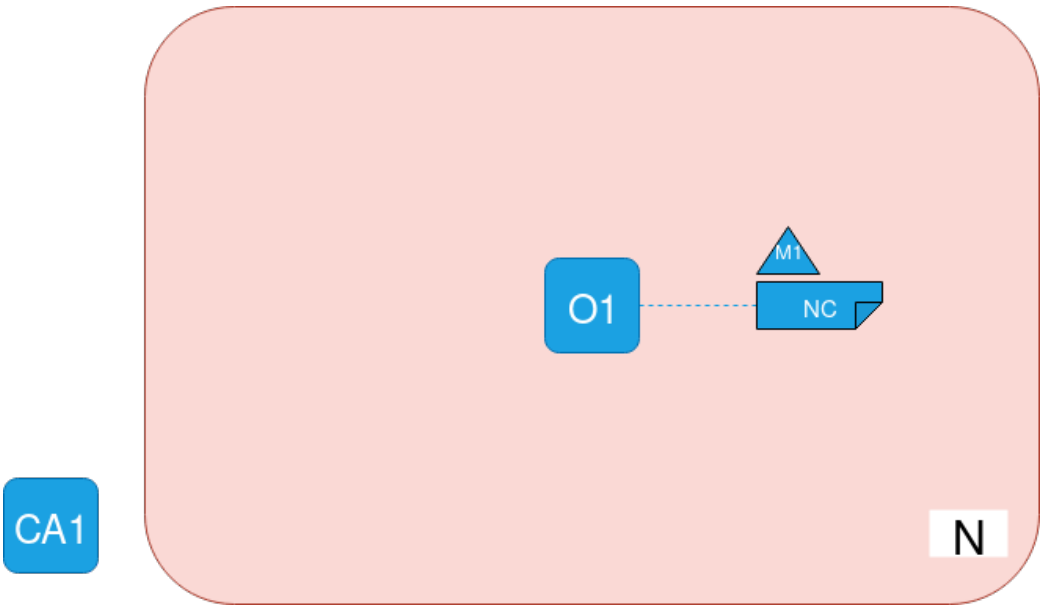
The network

The network will be built by two organizations:

1. Organization M1, whose elements will have blue color
2. Organization M2, whose elements will have red color

Creating the network

First we create the basis for the network:



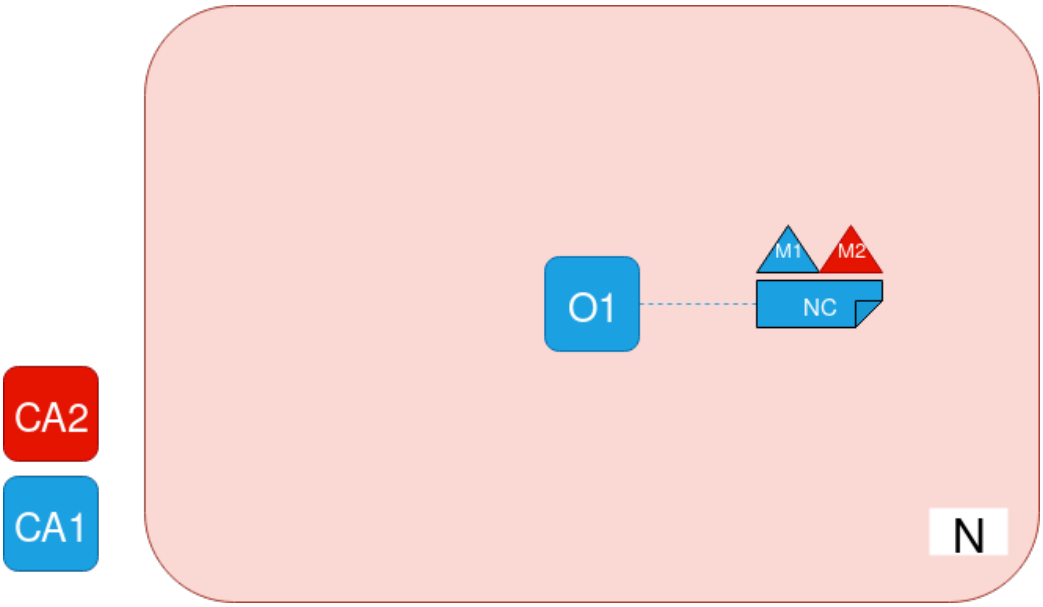
To start our network N, we need the ordering service(O1). For now we can consider the ordering service as a network access point. The ordering service contains the network configuration(NC) which defines a set of policies. These policies determine who has administrative power over the network.

Since for now only organization M1 is involved in the network, only it has the administrative power. The network configuration policies can change as we will see.

Different components of the network will use certificates to identify themselves as being part of an organization. Usually each organization has it's own certificate authority which issues the certificates for the member of the organization. Note how CA1 in the picture, the certificate authority for organization M1, is not a part of the network.

Adding an organization

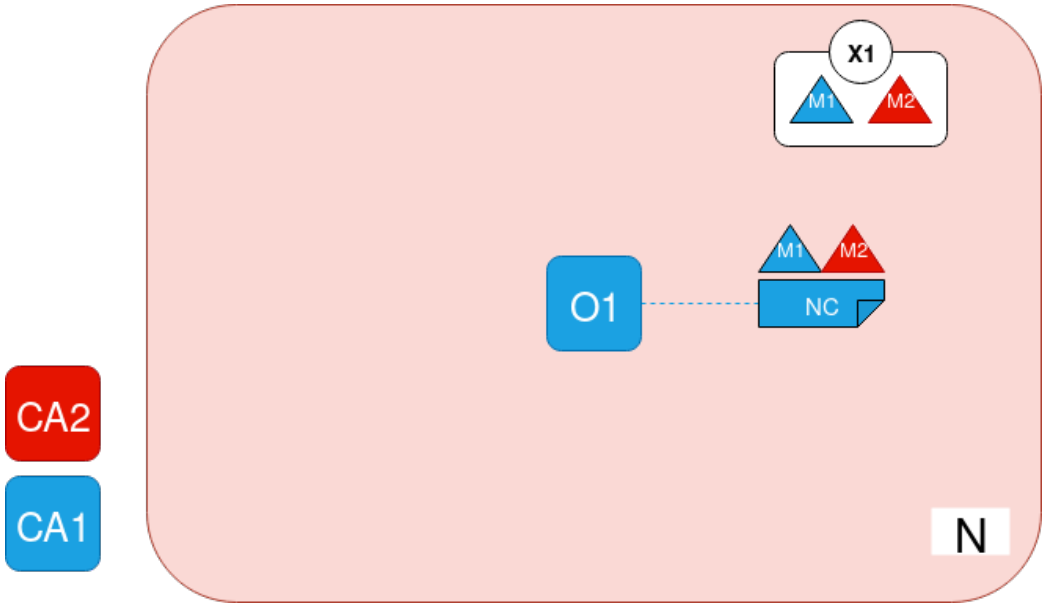
Now an administrator from organization M1 modifies the network configuration to give administrative powers also to the organization M2.



Now both organizations have equal rights over the network. Even though orderer node O1 is running on M1's infrastructure, the organizations share equal rights over it. Usually orderer service consists of multiple orderer nodes from different organizations. In this example we will just have one orderer node to avoid making a mess on the diagram. We can see that also CA2 has been added so that the users from organization M2 can be identified.

Creating a consortium

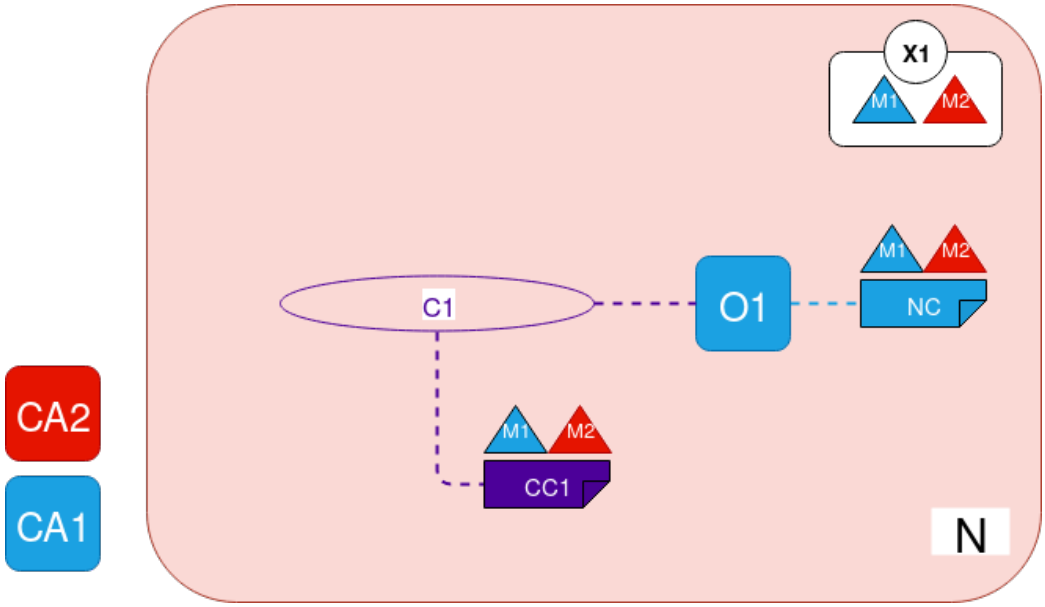
A consortium defines the set of organizations in the network who want to conduct with each other.



In the diagram we can see an addition of a new consortium, X1, made up of the two organizations. Consortium definition is saved in the network configuration NC. In general in the network we would have many organizations and any organization who has administrative rights over the network could define consortia. A consortium can contain any number of organizations from the network.

Creating a channel

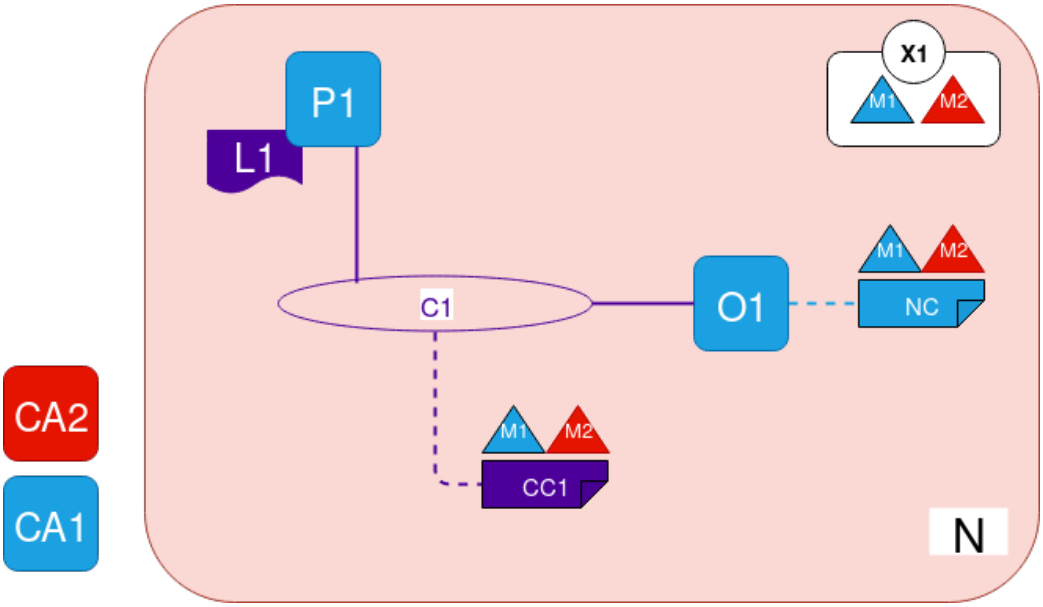
For the members of a consortium to be able to communicate to each other they need to create a channel.



The channel represent a private communication mechanism for the member of the consortium which created it. Any organization which is not part of the consortium can't see or participate in the channel. Each channel has it's own private ledger. We can see that channel is controlled by a different set of policies which are stored in the channel configuration CC1. These policies determines the rights of the organizations over the channel. Channels are a powerful tool because they allow organizations to both share the infrastructure and keep it private at the same time. We can see that the channel C1 is connected to the orderer service, but to nothing else. For now, the channel represents the potential for conducting business between the organizations in the consortium. In the next sections we will add actors that will realize this potential.

Peers

Now each organizations can add peers, which are nodes that host the ledger.

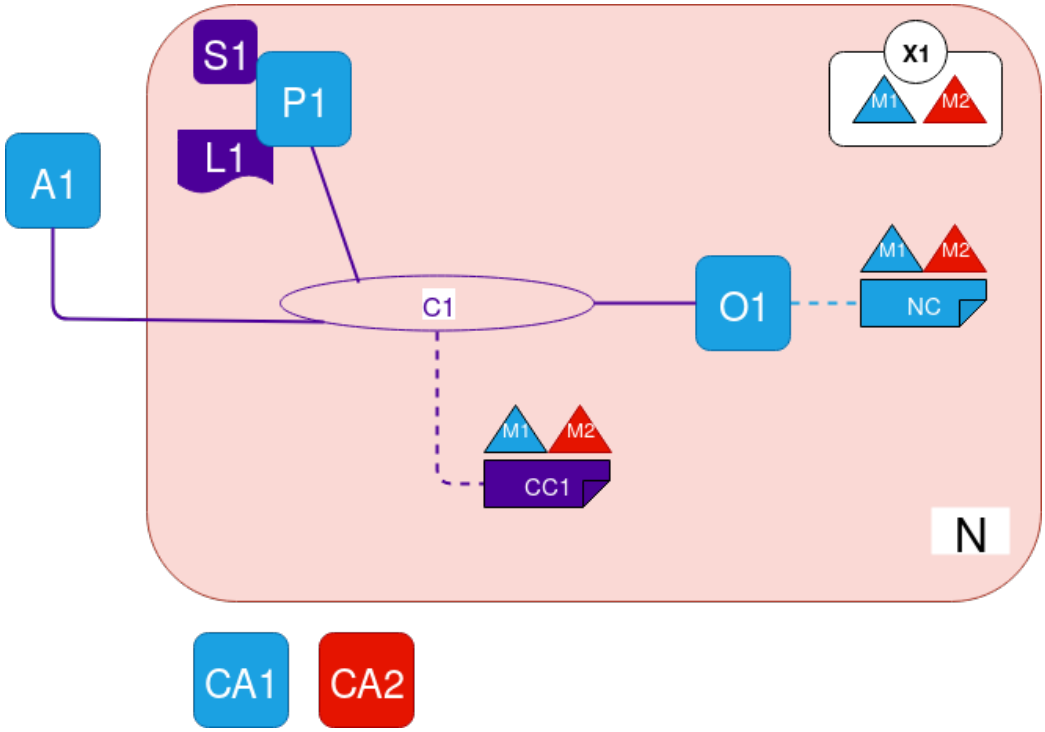


In the diagram we see that we have added a peer P1 belonging to the organization M1, and it's hosting the ledger L1(the ledger of the channel C1). Each peer connected to a channel hosts a physical copy of the ledger, but the ledger is logically hosted on the channel.

Once a peer is started it can join a channel through an orderer. When the orderer receives the join request, it uses the channel configuration to determine P1's permissions on the channel. For example, if the peer can write information on the ledger.

Clients and Smart Contracts

So far we have built a network that can host the blockchain, but we still don't have a way to interact with it. For this we need smart contracts and client applications.



In the diagram we see a smart contract S1 hosted on the peer P1. Smart contracts, or chaincodes in hyperledger fabric terminology, are written in a programming language(Go, Java, NodeJS) and represent the only way to access/modify the ledger. They essentially represent the business logic.

For a chaincode to be invokable(runnable) it has to be installed and instantiated. When a chaincode is installed on a peer, it means that the peer is able to run the chaincode, and it can see the implementation logic of the chaincode. When a chaincode is instantiated on a channel, it means that all members of the channel will know about the existence of the chaincode, that is any member of the network will be able to see the chaincode interface but not the implementation logic. Only the peers where the chaincode is installed will be able to see the implementation logic. When a chaincode is instantiated, the endorsement policy is provided and saved in the channel configuration. Endorsement policy specifies which organizations must approve the transaction before it's accepted and saved to the ledger.

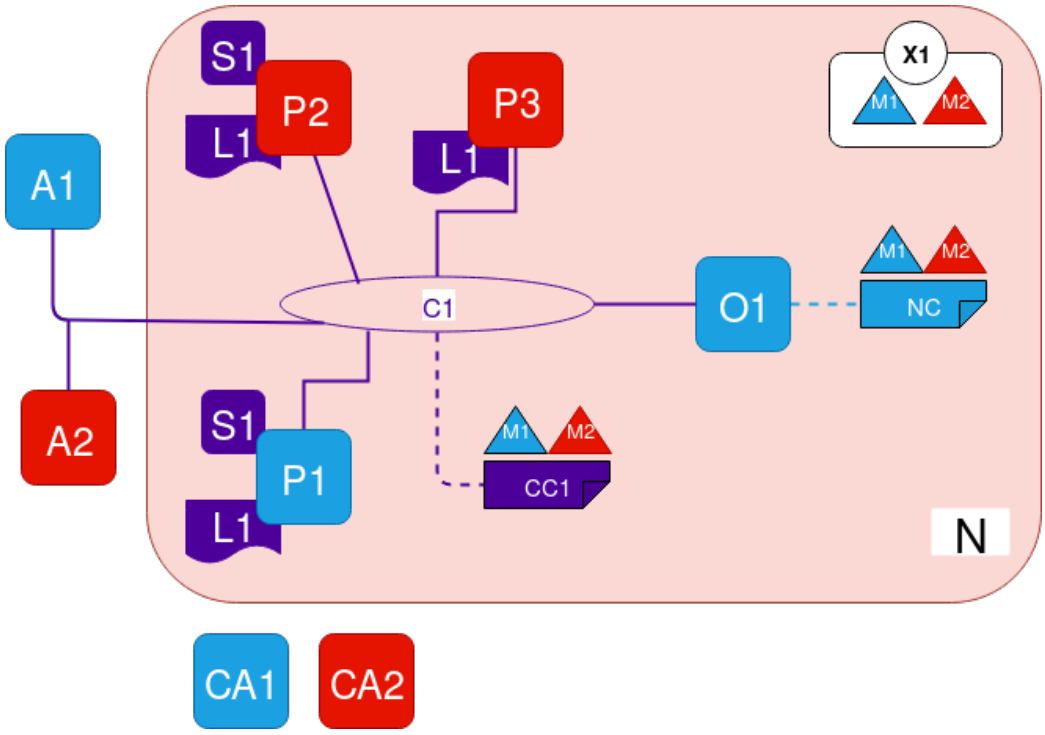
We can see outside the network the addition of a client application A1. This client belongs to the organization M1, therefore it has a certificate issued by CA1 and all the transactions it wants to do must be signed. To interact the client has to join the channel C1 since it is the only mechanism of communication in the network.

The invocation of a chaincode usually goes as:

1. Client sends a transaction proposal to peers belonging to organizations specified by the chaincode endorsement policy
2. The transaction proposal is used as input for the chaincode, then the chaincode is run to generate a transaction response, and the peer where it is run signs it to create a endorsed transaction response, which is returned by the peer to the client application.
3. Then these endorsed transaction responses are packaged together with the transaction proposal to form a fully endorsed transaction, which is distributed to the whole network.

Complete network

Since the consortium is made from two organizations we can assume that also the organization M2 will interact with the network.



In the diagram we see that the organization M2 has added two peers: P2 and P3. P2 has the chaincode S1 installed on it while P3 doesn't. P3 still knows the existence of the chaincode and it's interface. Note that when chaincode is installed on P2 there is no need to instantiate it, since it's already instantiated on the channel by P1. This shows that we can view

smart contracts the same way as the ledger, physically hosted on peers but logically on the channel. M2 has also added it's own client application A2 to be able to interact with the blockchain network.

Transactions

Execute-Order-Validate

One of unique capabilities of hyperledger fabric is the execute-order-validate architecture for transactions. This architecture is the key behind fabrics flexibility, scalability and performance. The architecture specifies that the transaction flow is divided in three steps:

1. *execute* a transaction and check its correctness, thereby endorsing it.
2. *order* transactions via a consensus protocol
3. *validate* transactions against the relevant endorsement policy(based on the chaincode executed) before committing them to the ledger

In fabric, an endorsement policy specifies which peer nodes, and how many of them, need to vouch for the correct execution of a smart contract. This means that only a subset of peers needs to execute the chaincode invoked by the transaction. This allow for parallel execution and scalability of the system. The first step also eliminates non-determinism meaning that unlike other blockchain platforms chaincodes can be written in standard programming languages.

Peer Types

Based on the execute-order-validate architecture we can divide peers in two types:

1. Endorsing Peers: Every peer that has a smart contract installed *can* be an endorsing peer. But to *be* an endorsing peer, the smart contract on the peer must be invoked by a client application to generate a signed transaction response.
2. Committing Peers: Every peer node in a channel is committing peer. It receives a block of generated transactions, which are validated before being committed to peer's copy of the ledger as an append operation.

Orderer

Other than being the management point for the network, the ordering service has another important function: distribution of transactions. It collects endorsed transactions from client applications, orders them into transaction blocks, which are then distributed to every peer node in the channel. At each committing peer, transactions are recorded, whether valid or invalid, and their local copy of the ledger is updated.

Transaction Flow

Transaction flow always starts in the same way. A client, with an appropriate and valid certificate, connects to a peer. After that, the client is able to invoke chaincodes on the peer. The invocation happens when the client sends the invocation proposal containing the name of the chaincode and input parameters. From here the transaction flow depends on the type of transactions. Transactions can be divided in two types, based on if they update the ledger or if they simply read from the ledger.

Query Transactions

These are transactions that only read data from the ledger. Since each peer contains an identical copy of the ledger, the chaincode invoked by the transaction is only executed on the peer where it is invoked. There is no need for any consensus protocol since the ledger is not modified. After the execution of the chaincode, the peer returns the proposal response, that contains the result(s) of the query, to the client application.

Update Transactions

The transactions that update the ledger can't be executed only on one peer. Consensus protocol must be respected, and transactions must be ordered and packaged into blocks before being committed to the blockchain.
The update transactions start in the same way except now when the peer responds to the client, the proposal response will contain a proposed update, *i.e.* the update that peer would apply if other peers have already agreed to it.
After receiving the proposal response containing the proposed update, client's job is to deliver the proposed update as a transaction to all peers in the network. To accomplish this, the client will send the proposed update to the orderer. The orderer will take the proposed update, package it in the transaction format, order and collect transactions into blocks. When an orderer has

a block ready, he sends it to all the peers in the network, who then validate each transaction in the block and update their respectful ledgers. Since this collection and ordering may take some time(few seconds) the application is notified asynchronously when the ledger is updated.

4.3 Mapping the physical actors to blockchain network roles

4.3.1 Clients

From the blockchain architecture we see that clients are the ones who invoke transactions. We define two types of clients:

1. Clients that invoke transactions to upload new reports into the blockchain
2. Clients that invoke transactions to read CAM information stored in the blockchain

Base stations represent the first type. After receiving the reports they use an installed client application to send the reports by invoking the corresponding transaction in a peer. Every base station acts as a client and therefore has a valid certificate that allows both querying and modifying the blockchain data.
The second type of clients can be represented by authorized parties interested in querying the information stored on the blockchain. Some example of such parties are: Insurance companies, Police department, Traffic analysts. As with any client application to invoke transaction they need to posses a valid certificate. For these parties the privileges corresponding to their certificates should allow only querying the data, *i.e.* not modifying.

4.3.2 Peers

Committing Peers

Committing Peer is every peer that maintains a copy of the distributed ledger. Therefore every Base Stations will act as a committing peer.

Endorsing Peers

Endorsing peers are peers who execute the invoked transactions. Not all base stations need to be endorsing peers. Higher the number of endorsing peers, higher the possible level of parallelism.

4.3.3 Orderers

We can consider the ordering service as a cluster of ordering nodes. The type of nodes depends on the ordering algorithm used: Kafka or Solo. In the context of this thesis we will consider ordering nodes as an already provided cluster.

4.3.4 Many roles of base stations

We see that a base station can play up to three roles: Clients, Committing Peers, and Endorsing Peers. It is important to state that even though all the roles are implemented in the same physical location they are logically completely separated. For example if a client is interacting with the peer in the same base stations, it still needs a valid certificate for the interaction.

4.4 Load Balancing

In most cases whenever the client inside a base station needs to upload the report, it would invoke the transaction on the peer in the same base station. This makes sense since this is where the latency is lowest(same physical location)
By the specification vehicles send CAM Reports to the physically closest base stations. In a dense traffic area this could mean most of the reports being received by a single base station resulting in a processing delay. In such cases the client can choose to invoke the transaction on a peer in a different base station(belonging to the same channel). This introduces some latency but provides load balancing on the clients' base station. A protocol for finding the least loaded base station would need to be implemented. The ordering service could be modified as the central point of this protocol so that client's can get the least loaded base station by communicating with the ordering service.

4.5 Geographical Sharding

In hyperledger fabric each peer must maintain a full copy of the ledger. There is no possibility of data sharding. This means that if this architecture would be installed throughout a country, each peer would have to maintain the complete ledger for all reports generated in the whole country. This is clearly infeasible.

To solve this issue we can (mis)use the concept of channels. To recall, in fabric channels are used for private communication in a blockchain network. Each channel has it's own ledger and is only accessible to the members of that channel.
Based on distribution of base stations we can divide them into geographical areas of some size. Each geographical area will maintain it's own channel and ledger. The base stations on the border between areas will belong to all neighboring channels. PICTURE Problem could be no connectivity, in this case the reports should be sent to the base station closest to the generated CAM inside the report.

4.6 Application Architecture

Looking at the specification of the use case we see that the amount of data to generate and analyze is enormous. Therefore the design of the architecture should be done with great care and taking into account many factors to avoid unnecessary data and to minimize data duplication. Some factors to take into account is how the blockchain is structured and the append-only property of the blockchain, transaction format, the required querying possibilities, design of assets, etc.

4.6.1 Requirements

The developed application should allow storage of CAMs, verification of CAMs, and be as flexible as possible with it's querying capabilities. Some query examples would be:

- Retrieve a CAM and check if the CAM is valid
- Retrieve all CAMs of a vehicle in a certain time interval
- Retrieve all vehicles which have went over a speed limit in a certain time interval
- and many more

4.6.2 Input for the application

As we have seen the stations send reports to our application for storage. The report structure is again reported here for clarity purposes:

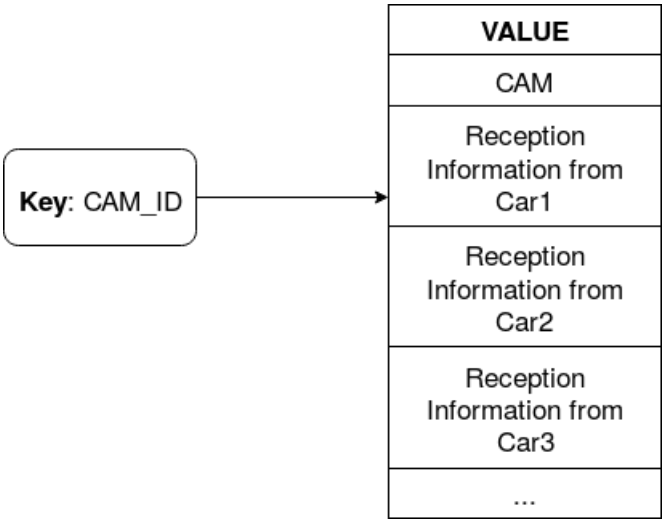
ReportId	A string that uniquely identifies the report derived from the generated CAM. It is obtained from the concatenation of the vehicle ID and the timestamp of the generated CAM.
GeneratedCam	The CAM message generated by the reports' owner.
GeneratedCamHash	
ListReceivedCams	A list containing reception information for each received CAM since the last report.
PreviousReportId	The id of the previous report generated by this report's owner.
PreviousReportHash	The hash of the previous report generated by this report's owner.

In the **ListReceivedCams** each member consists of:

ReceivedCamId	The ID of the received CAM
ReceivedCamHash	The hash of the received CAM
ReceptionInfo	Physical characteristics of the report owner in the time instant when the CAM was received

4.6.3 Initial design

We see that our application should store the data found in the reports, essentially the CAMs and the reception information needed for the verification of CAMs. From the description of hyperledger we see that data interaction happens through a key value store(KVS) database. For now let us concentrate on the requirement that we want to be able to retrieve a CAM efficiently and check if it's valid. Since we cannot know which reports contain reception information for a CAM, *i.e.* we don't know which cars received the CAM, we should store a CAM and it's reception information together. We can start by storing each CAM with the information needed for it's verification under the same key. Since CAMs don't have IDs we can define it as the ID of a vehicle concatenated with timestamp of the CAM. So an example of our key value pair in the database will be:



Now if the application needs to retrieve the CAM, it simply accesses it using the CAM_ID and selects the CAM field. If it needs to verify it, it has all the information in the reception information fields. The verification is done using a position verification algorithm.

4.6.4 Problems with the initial design

The initial design allows us to satisfy our requirement, but is very memory inefficient. To understand why we should examine the ledger in more detail. As previously explained, the blockchain is a collection of cryptographically connected blocks. Each block contains the data and metadata to guarantee immutability, but the actual content of a block is simply a set of transactions. Transactions record modifications performed to the defined assets. They record it using the format of a Read Write set(RW-set).

Read Write Set

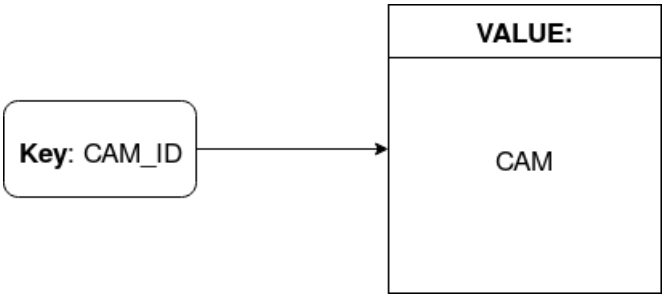
Before explaining the semantics of the RW-set, it should me mentioned that each asset next to a key has also a version. Each modification of an asset increments the version. This is used to check that during the transaction endorsement process no other transaction has already modified the value.

Each transaction contains a RW-set. The read set is made of a list of unique keys and their committed version that the transactions reads. The write set contains a list of unique keys(these can overlap with the keys present in the read set) and their new values that the transaction writes. A delete marker is set(instead of the new value) if the update performed is to delete the key.

An example of a read-write set: ADD ILLUSTRATION
In the validation phase, a transaction is considered valid if the version of each key present in the read set of the transaction matches the version in the world state.

Memory utilized by the initial design

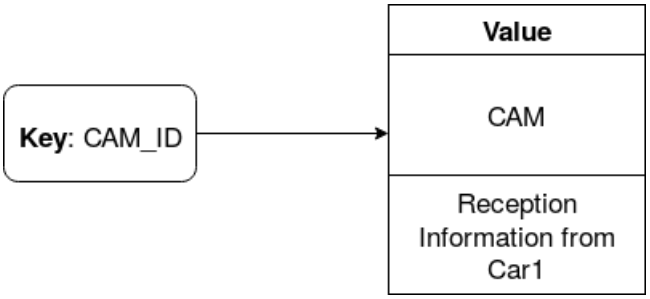
Due to the nature of transactions and read-write sets, append structures are very inefficient. To understand let's run through an example for one specific CAM.
To start we receive a report containing the CAM as the generated CAM and information about other received CAMs. In our example we are only interested in the generated CAM. So after our application is done processing the report, our key value pair would be:



And in the blockchain the RW-set of the corresponding transaction would be:

$$\textit{ReadSet}\text{: } \{\}$$
$$\textit{WriteSet}\text{: } \{ \text{ Key} = \text{CAM\_ID, Value} = \text{CAM; } \}$$

Now we receive another report and in the processing of it we realize that the report has received our CAM so we decide to save the related reception information. So now our asset looks like:

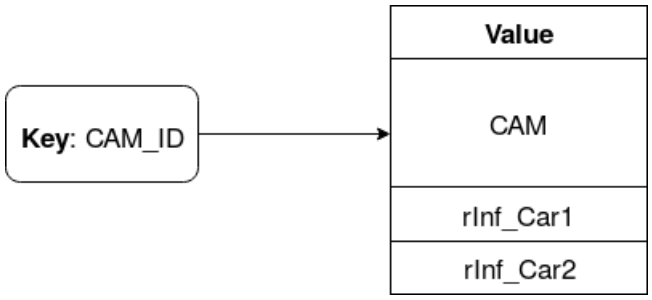


And the new transaction will have the RW-set:

ReadSet: {Key = CAM_ID, Version = 1}
WriteSet: { Key = CAM_ID, Value = (CAM, rInfCar1); }

Since we are adding the reception information we need to first read the value, then add rInfCar1 to the value, and after write the whole value back.

Another report containing rInf about our CAM arrives. We perform everything as in the previous step. So our asset becomes:



And the new transaction has the RW-set:

ReadSet: {Key = CAM_ID, Version = 2}
WriteSet: { Key = CAM_ID, Value = (CAM, rInfCar1, rInfCar2); }

Again we have to read the value, append our new information, and then rewrite the value in the blockchain.

Now for all the cars that received our CAM, reports arrive containing the reception information we need for CAM verification. And all of them follow the pattern described in the last two steps of the example. For the following calculation lets define N_r as the number of reports that arrived whose information we need to save for our CAM.

Let's see how much memory has our key value pair occupied on the ledger. First in the world state we have the CAM and N_r reception informations from different reports, therefore :

$$MemoryWorldState = sizeof(CAM) + N_r * sizeof(recInf)$$

This size is necessary since we need the value stored in world state database for efficient querying.
Then in the blockchain we have the value written for each transaction. Since there was the initial transaction(T0) for writing the CAM and N_r transactions($T_1, ..., T_{N_r}$) for each report containing the CAM. Therefore:

$$MemoryBlockchain = sizeof(T_0) + \sum_{i=1}^{N_r} sizeof(T_i)$$

Since the values are much larger than the keys, in the computations we will concentrate on the values found in transaction write-sets. Since the complete value has to be overwritten each time, we have:

$$\begin{aligned} sizeof(T_0) &= sizeof(CAM) \\ sizeof(T_i) &= sizeof(CAM) + \sum_{j=1}^{i} sizeof(recInf) \end{aligned}$$

This size is due to the fact that each transaction is recorded in the blockchain with it's read-write set and we can't go back and deletes part of it because the blockchain is immutable. So our initial structure consisting of one CAM and Nr recInf will occupy:

$$MemoryWorldState = sizeof(CAM) + N_r * sizeof(recInf)$$

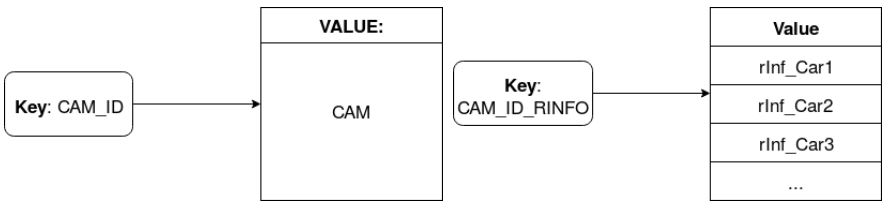
$$MemoryBlockchain = \\ sizeof(CAM) * (N_r + 1) + N_r * (N_r - 1) * sizeof(recInf)/2$$

$$TotalMemory = MemoryWorldState + MemoryBlockchain = \\ (N_r + 2) * sizeof(CAM) + sizeof(recInf) * N_r * (N_r/2 + 1/2)$$

It is worth mentioning that in this subsection when processing a report we have only shown one CAM in the RW-set. In reality there will be as many CAMs processed in one transactions as there are in the report. The goal was to show for just one of them, how inefficient the design is.

4.6.5 Improving the design

We see that whenever we receive new reception information, to save it we have to overwrite the CAM. This can be easily fixed by defining a key that stores only the CAM and the key that stores all the reception information. Therefore our structure would be made from two assets:



This already decreases the CAM part of the size from (Nr + 2) to 2 because it's only written in the initial transaction(on the blockchain) and in the database copy. Since CAM contains more information than rInfos this is

already a substantial increase at the cost of one extra key.
The issue that remains is the quadratic growth of values modeled by append structures. In our case the size of Nr rInfos will grow as $O(N_r^2)$. There are different ways to mitigate this:

1. **Define a new asset for each reception information.** The pros of this model is that it avoid append like structures and allows for maximum parallelism. The cons is that it requires an additional transaction to perform the verification, *i.e* we can't validate CAMs partially whenever a reception information about a CAM is received.

2. **Create an off blockchain database** on each peer where the reception information are stored and when enough of them is collected or some time has passed write them all in one go. This is beyond the scope of the thesis since it involves modifying the hyperledger fabric itself and guaranteeing synchronization between the off blockchain databases.

3. **Implementing an incremental verification algorithm**Since the rInfos are used to verify the truthfulness of a CAM, if the position verification algorithm can be performed incrementally and results aggregated, we could compute a part of it each time a new rInfo arrives and keep only the last rInfo stored as the value.

Examining The Incremental Verification Algorithm

Initially the third option was chosen for the implementation in this thesis since the position verification algorithm can be performed incrementally. But during the testing a large number of invalid transaction was noticed with parallel transactions. When processing the report the chaincode has to check for each reception information if the relative CAM has already been stored in the blockchain. But it is possible(even likely) that there is a pending transaction that still hasn't been committed to the blockchain containing the CAM in question. This means that execution will proceed normally but when the transaction has to be verified there will be a read set version inconsistency and the transaction would be declared valid. This problem is illustrated by the following example: HDAUASIBHDHOAIDKBASDISD

Reception Information Asset

Therefore the chosen implementation is to create a new asset for each reception information.
First a unique key should be defined for each reception information: since

each CAM can be received at most once by each vehicle, a unique key can be obtained by creating a composite key which combines the *CAM_ID* and the*Receiver Vehicle ID*:

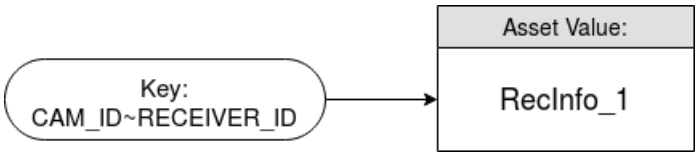


Figure 4.2: Reception Information Asset with a unique composite key consisting of the ID of the CAM to which the information refers and the ID of the vehicle that provided the information

In fabric the composite keys allow providing wildcards for the second key parameter. This means that to obtain all the reception information for a given CAM is simply requesting all the assets given that the first part of the composite key is equal to the CAM_ID.
With this architecture we do not perform any reads in the database while processing a report therefore it doesn't matter if the CAM or the reception information arrive first.

Validity Asset

So far we have defined assets for storing the CAMs and the relative reception information in a memory efficient way.
The last information that needs to be provided is the validity of a CAM.
In order to avoid duplicating the CAM in the blockchain we will define a new type of asset which will store the current validity value for a given CAM. Therefore for each CAM we will have the following asset:

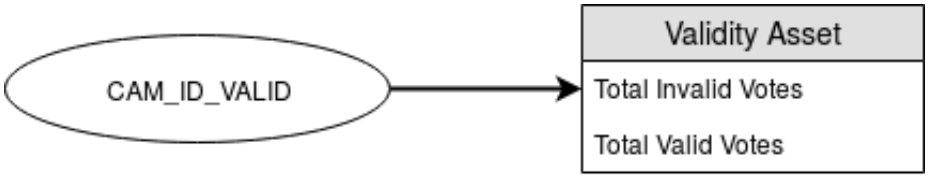
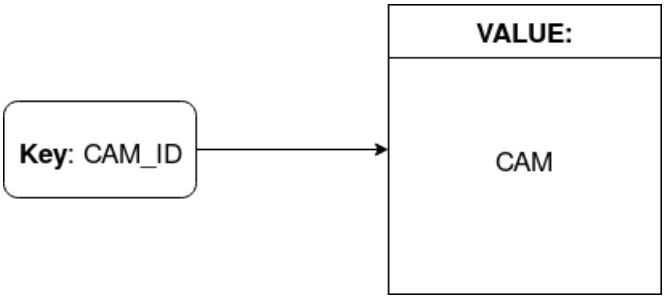


Figure 4.3: Validity Asset representing the results of the position verification algorithm for a given CAM. The key is a concatenation of the CAM_ID and the string "VALID"

4.6.6 Memory utilized by the improved design

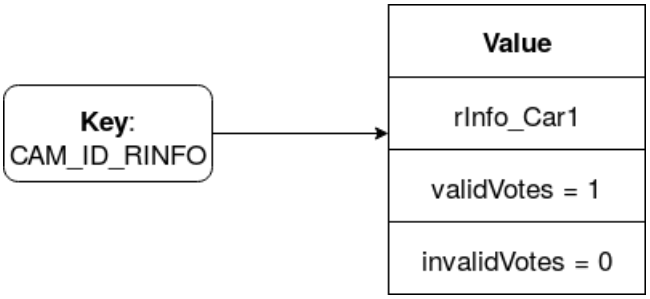
We can see that we have removed all append structures from the asset designs, now let's reiterate the example using the redefined assets and recalculate the complexity.
Again at the start we have the storage of CAM:



And in the blockchain the RW-set of the corresponding transaction would be:

$$\textit{ReadSet}\text{: } \{\}$$
$$\textit{WriteSet}\text{: } \{ \text{Key} = \text{CAM\_ID}, \text{Value} = \text{CAM}; \}$$

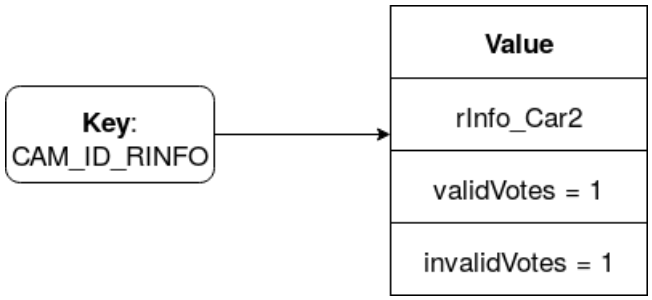
After that we receive a report containing reception information about our CAM. The application processes it and immediately computes the vote of the vehicle that sent the report with regards on the validity of the CAM. Let's assume the vote was valid. The data stored in the world state is:



And in the blockchain the RW-set of the corresponding transaction would be:

$$\textit{ReadSet}\text{: } \{\}$$
$$\textit{WriteSet}\text{: } \{ \text{Key} = \text{CAM\_ID\_RINFO}, \text{Value} = (\text{rInfo\_Car1}, \text{validVotes}{=}1, \text{invalidVotes} = 0); \}$$

As before another report arrives containing new reception information. We perform the same actions as in the last step and let's assume the that this one says that our CAM is invalid, so the asset will become:



And in the blockchain the RW-set of the corresponding transaction would be:

$$\begin{aligned} \textit{ReadSet}&: \{\ \text{Key} = \text{CAM\_ID\_RINFO, Version} = 1\} \\ \textit{WriteSet}&: \{\ \text{Key} = \text{CAM\_ID\_RINFO, Value} = (\text{rInfo\_Car2,} \\ &\qquad \text{validVotes}=1, \text{invalidVotes} = 1);\ \} \end{aligned}$$

Now many more reports will arrive containing rInfo about our CAM, and for each them we will perform the same step as previous, modifying the valid and invalid votes accordingly.
Calculating the memory occupied for a general case of having N_r reports containing recInfo about our CAM. The votes are simply integers.

$$MemoryWorldState = sizeof(CAM) + sizeof(recInfo) + 2 * sizeof(int)$$

Then in the blockchain we have the value written for each transaction. Since there was the initial transaction(T0) for writing the CAM and N_r transactions($T_1, ..., T_{N_r}$) for each report containing the CAM. Therefore:

$$MemoryBlockchain = sizeof(T_0) + \sum_{i=1}^{N_r} sizeof(T_i)$$

With the size of transactions being:

$$\begin{aligned} sizeof(T_0) &= sizeof(CAM) \\ sizeof(T_i) &= sizeof(recInfo) + 2 * sizeof(int) \end{aligned}$$

So the total memory occupied will be:

$$\begin{gathered} TotalMemory = MemoryWorldState + MemoryBlockchain = \\ sizeof(CAM) + sizeof(recInfo) + 2 * sizeof(int) + sizeof(CAM) + \\ \sum_{i=1}^{N_r}(sizeof(recInfo) + 2 * sizeof(int)) = \\ 2 * sizeof(CAM) + (2Nr + 2) * sizeof(int) + (Nr + 1) * sizeof(recInfo) \end{gathered}$$

Which is an enormous improvement comparing to the initial design. The price we pay is that if we wan't to find out which Cars have reported the CAM we would have to query the blockchain. But our goal is mainly oriented to check if CAM is valid or not, which can be done more efficiently with this structure.

4.6.7 When To Validate

As stated before to guarantee transaction validity while allowing for parallelism in the network we should avoid reading any information from the blockchain when processing a new report. This means that it's impossible to program the validation to be performed automatically during the CAM Reports storage.
Since there is also a time limit on smart contract execution it is impossible to have a periodic process in the blockchain that validates the cams.
The option that remains is that the base station which uploads the CAMs also requests the validation of the CAMs. Therefore the validation will be performed in the following steps:

1. A stations collects reports from vehicles passing by.
2. Station invokes the smart contract to upload the reports to the blockchain. Station keeps the IDs of all reports(same as the generated CAM in the report) that were uploaded in this batch.
3. Station waits some time for the reports to be processed and stored in the blockchain.
4. Station invokes the smart contract to validate all the cams that were sent in this batch.

The amount of time the station has to wait can be determined from the experimental data which will be explored later in the thesis.
Furthermore, considering that the use case of thesis doesn't require validation as soon as possible it makes practical sense to perform validation of uploaded CAMs when the station is idle.

Multiple Validations

Since validation is performed per request, there exists a possibility that additional reception information arrives after the validation is performed.
To accommodate for this, whenever a validity asset is queried additional information is returned specifying if new reception information has arrived

since the last validation performed on the CAM. In this way the client can decide if it is satisfied with the current validity results, or it wants to invoke a new transaction that adds the results from the new reception information.

Deleting Reception Information

Whenever a validation is performed the position verification results for all reception information is stored in the validity asset. Since all assets are immutably written in the blockchain it is reasonable to delete all the reception information for which the position verification results were calculated from the state database. Note that all the reception information that was deleted can still be retrieved by traversing the blockchain.

4.6.8 Providing flexible querying capabilities

So far our improved design allows us to query and verify CAMs while storing them in an efficient manner. But it doesn't allow us much querying flexibility, *i.e.* we can only query based on the CAM_ID or CAM_ID_RINFO. To see how we can add this flexibility to our application, let's choose one more requirement: on top of retrieving CAMs and their validity status **we would like to be able to retrieve all CAMs of a certain vehicle in a certain time interval.**

4.6.9 Choice of the database

As explained before, we would like to avoid append-like structures due to their quadratic growth rate of memory needed. To remember, hyperledger fabric supports two world state database implementations: LevelDB and CouchDB.

LevelDB

In LevelDB, data can only be queried using asset keys. This implies that if we want to retrieve all CAMs of a vehicle in a certain time interval we have to define a key foe every possible time range of interest. But since we are defining a range, the value of the key will be a list of CAMs which is an append structure. The overhead can be reduced by controlling the granularity of a range. It still means size growth of

$$O(N^2)$$

, but the N can be limited for each interval: worst case is

$$1/CAMgenerationFrequencyMax * lengthOfInterval$$

. The overhead is still very large, but relatively controlled. Another caveat is that if we implement such structures, on top of the overhead, we have only solved the current requirement. For implementing another requirement, especially ones based on ranges, such as querying vehicles based on speed in an interval, we would have to create new append structures. This follows for any type of new requirement because we have to bring the querying data from the CAM(our value) inside our key.

CouchDB

CouchDB supports all querying based on keys as LevelDB, but on top of that it allows to execute rich queries for JSON data. This means that if we store our data as JSON documents, we can query data based on the JSON field values, which for us means that we can query based on some CAM properties instead of only based on keys. Practically it means instead of creating new keys and values in the world state for each requirement needed, we can create efficient lookup structures in the CouchDB, and specify which structure to use when querying.
To allow efficient querying we have to define indexes by specifying which fields we will use for querying. After the index has been defined, we can query the database by specifying our query and which index to use.
In our example we define an index based on the VehicleId and Timestamp. After that we query our database with a query to select all CAMs whose $VehicleId = V_0, Timestamp > T_0, Timestamp < T_1$ and declaring to use index VehicleId_Timestamp.
CouchDB uses MapReduce paradigm for constructing indexes, and indexes are stored in shallow B-tree meaning querying requires in the worst case $O(log(N))$ complexity; in most cases it will be faster since the trees remain shallow even with a large number of keys.
We see that by using CouchDB there is no need to define new Key-Value assets, instead the querying flexibility is integrated in the database configuration. That avoids dealing with append structures and allows adding new requirements by simply defining new indexes.
The only downside of using CouchDB is that unlike LevelDB it runs in a separate container which means that every operation on the database must go through a REST API which implies that the absolute number of transactions per second will be lower that when using LevelDB.

In this thesis the CouchDB will be the database of choice for it's flexibility capabilities and avoidance of huge memory overhead.

4.7 Putting It All Together

The goal of this section is to summarize the ideas explored in this chapter and show how the three main operations: Storing, Validating, and Querying of CAMs and related information is performed.

4.7.1 Asset Types

4.7.2 Storage Operation

The storage of CAMs in the blockchain proceeds in following steps:

1. A station collects reports from passing by vehicles and collects them into a batch.

2. When enough reports is collected or a time limit is reached the station passes the batch of reports to the blockchain client. The blockchain client is located in the same station.

3. The client invokes smart contract for storing reports on the blockchain peer with the input being the batch of reports. Unless a load balancing mechanism is implemented it invokes the transaction on the peer located in the same station. The client also stores the CAM Ids for all generated CAMs in the reports.

4. The peer executes smart contract for storing reports. Providing that the transaction reaches consensus and is valid after some time the results of the transaction are committed on the distributed ledger.

The assets that are stored in the blockchain for each report in the batch are the following:

1. The generated CAM in the report

2. An asset for each reception information in the report

The figure FIGUREREFERENCE represents the storage operation and the assets stored in the blockchain graphically. In the example the batch consists of three reports uploaded by three different cars:

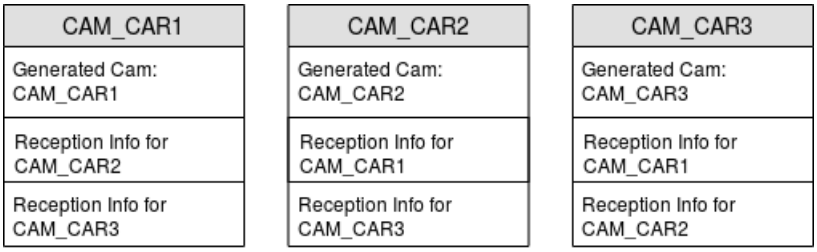


Figure 4.4: Three reports in the batch. Each report contains reception information about the other two.

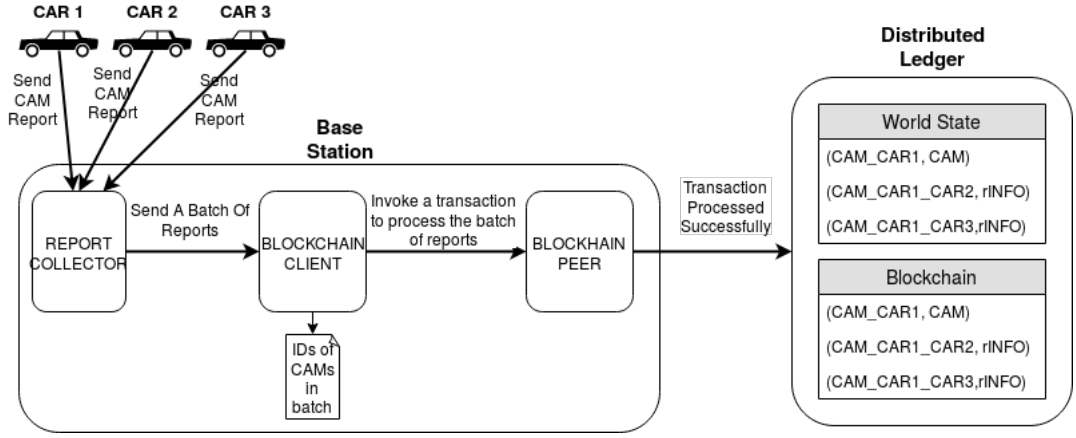


Figure 4.5: Upload Operation: In the ledger we only represent information relating to the CAM generated by the first car. The ledger also stores the second and third CAM and their reception information, but it's not shown here for clearness purposes.

4.7.3 Validate Operation

The validation of CAMs is intrinsically connected with the storage of reports in the blockchain. Whenever a base station uploads a batch of reports for storing, it will wait some time and invoke the validation transaction.
The validation of CAms proceeds in the following steps:

1. After the blockchain client performs the storage operation for the batch of reports it waits until the results of the transaction are propagated throughout the blockchain network.

2. Then the blockchain client reads the CAM IDs from the batch which were stored in a temporary file before the storage operations.

3. The blockchain client invokes the validate operation in the peer located in the same base station. The input to the transaction is a list of CAM IDs to be validated.

Starting from figure 4.4, the base station waits some time and performs the validation. The operation is illustrated in the following picture:

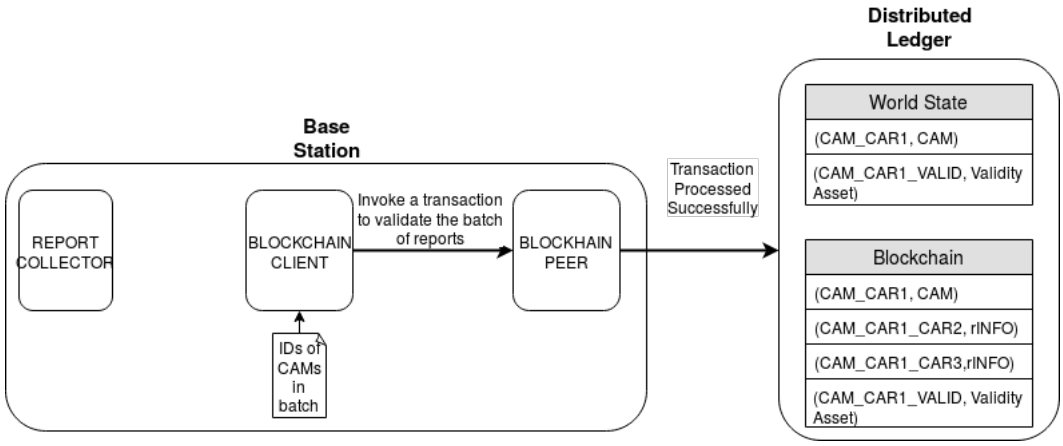


Figure 4.6: Validate Operation: Again we are only representing the data in the ledger for the CAM from CAR 1 for brevity purposes. The corresponding data is present also for the CAR2 and CAR 3.

Chapter 5

Developed Applications

5.1 Vehicular Mobility Simulator

Vehicular Mobility Simulator is a program that simulates traffic in a given area. For certain time length it simulates the following:

- Movement of vehicles
- CAM generation, transmission and reception
- Report creation and transmission from vehicles to base stations
- Transmissions of reports from base stations to the blockchain

The developed simulator is based on the Random Waypoint Model. It is developed in the Go programming language.

5.1.1 Random Waypoint Model

Random Waypoint Model is a random model for the movement of mobile nodes, and how their location, velocity and acceleration change over time[wikipedia].
The following steps describe the movement of each node during the simulation in the RWM:

1. A node is given initial random location
2. The node pauses for a fixed time period
3. Then the node chooses a random destination, speed and acceleration.
4. The node moves with given speed and acceleration to its' destination
5. When the node arrives to the destination, it restarts from the step 2.

Differences from the Random Waypoint Model

In the developed simulator vehicles represent the nodes in RWM. The developed simulator has the following differences with respect to the RWM:

- Acceleration of all vehicles is set to 0
- Speed of all vehicles is equal and is a constant value
- The simulation area is a square area
- Vehicles do not pause before choosing a new destination

5.1.2 Simulator Description

5.1.3 Inputs of the simulator

At the start the developed application reads the simulator parameters from a configuration file written in YAML. The parameters provided by the configuration file are the following:

Area size	The size of the square on which the simulation will be run
Number Of Vehicles	The number of vehicles that generate and receive CAMs
Base Stations	A list of base stations with their respective locations
Simulation Time	Length of the simulation in seconds
CAM generation frequency	The frequency in Hz at which all vehicles generate CAMs
Cheating Probability	Probability that the generated CAM will have fabricated values
Station Generation Frequency	The frequency in Hz at which stations sends a batch of received CAM reports to the blockchain

Table 5.1: Vehicle Simulator Input parameters

5.1.4 Output of the simulator

To reduce the number of transaction, stations upload multiple CAM reports at a time to the blockchain.
The output of the simulator is a trace file where each line of the file corresponds to an event of the station sending a batch of reports to the blockchain. Each line consists of three fields:

1. A batch of reports to be sent by the station
2. ID of the station
3. Timestamp at which the station would send the batch to the blockchain. It is expressed as the number of milliseconds that passed since the start of simulation

5.1.5 Simulator Processing

The vehicle simulator execution happens in three main steps:

1. Simulator Initialization
2. The Simulation Loop
3. Dealing with the leftover output

Simulator Initialization

At the start the program read simulator parameters from the configuration file. The parameters are specified in section 4.1.2. After that a collection of N_v vehicle objects is created. At the start each vehicle is initialized with:

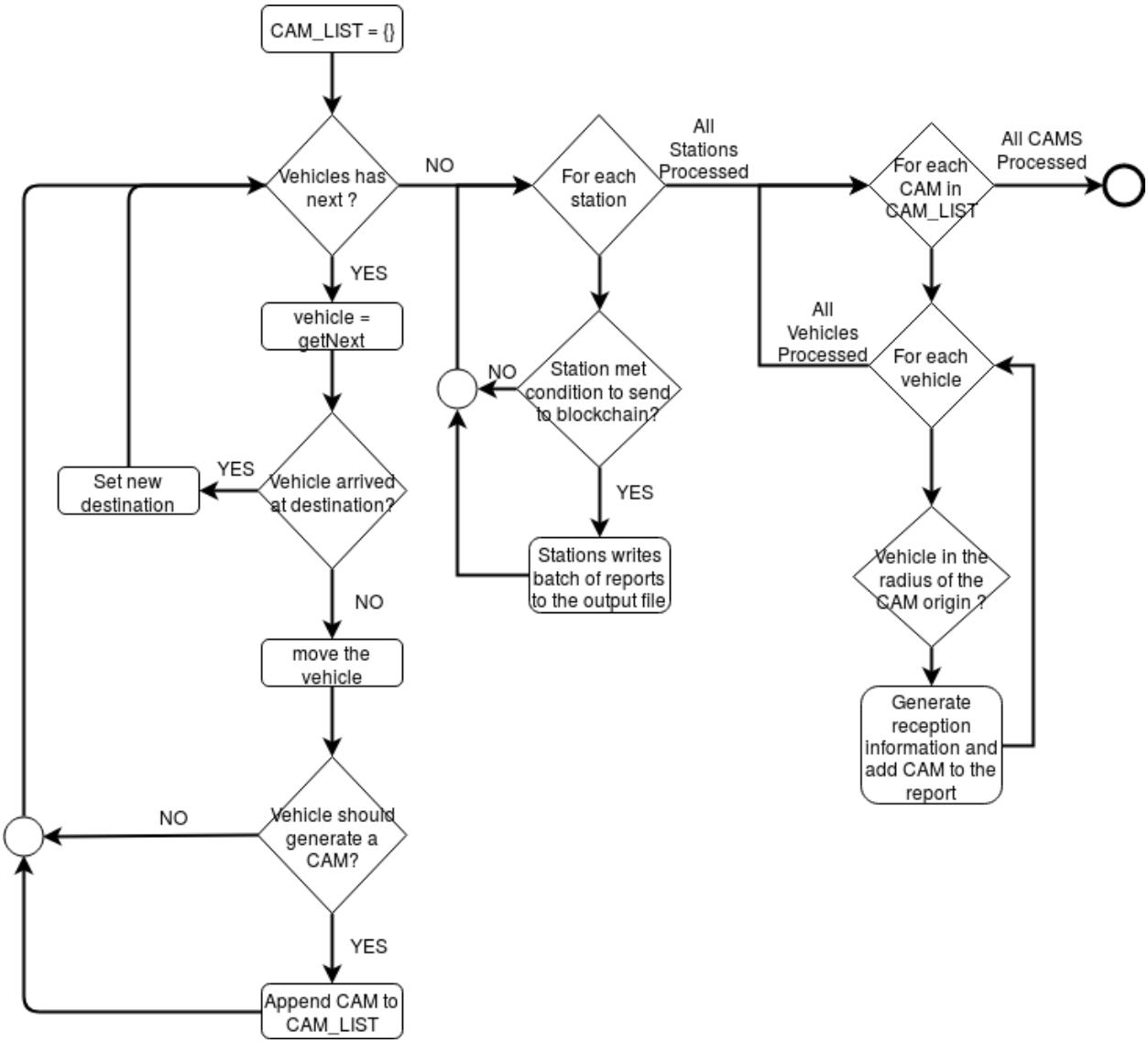
- Random starting location
- Random destination
- Random time of the first CAM generation.

The CAM generation frequency is fixed. It is provided by the config file. As specified in the simulator vehicles do not pause before moving to a new destination. Therefore to avoid having all vehicles generate CAMs and CAM reports at the same time the first CAM generation happens at a random time. This random time is different for each vehicle. It is smaller than $1/CAM_{generationFrequency}$ to avoid excessive delays.
After vehicle collection creation, a collection of stations is initiated. Each station has the location as specified in the config file.

The Simulation Loop

As specified in section 2.4 the timestamp of a CAM specifies the time passed since January 1, 2004 in milliseconds.
The main simulation loop runs for the number of seconds as specified in the

config file with an increment of 1 millisecond. A loop iteration is represented by the following flowchart:



Leftover Output

Stations send a batch of reports to the blockchain periodically with a given frequency, if any reports were received. Depending on the frequency and the simulation length there is a chance that there will some CAM Reports leftover in the station at the end of simulation. Therefore after the simulation loop we write to our output file the last batch of reports for each station.

5.2 Hyperledger Fabric Network Bootstrapper

Hyperledger Fabric Network Bootstrapper is a program written using Go programming language and Bash scripting language. It reads a single configuration file written in YAML as input. After which it configures and starts a hyperledger fabric network. Before explaining the developed tool we will explain how a fabric network is created on a local machine.

5.2.1 Creating a Hyperledger Fabric Network

Running a fabric network is tightly linked with the docker technology. Each node runs in its own separate docker container. This allows us to simulate a physical fabric network on a single machine.

Prerequisites

Before running a Hyperledger Fabric network, the following prerequisites must be installed on the machine:

- The latest version of CURL
- Docker version higher than 17.06.02-CE
- Docker-Compose version higher than 1.14.0
- GoLang version 1.11.x
- Node version 8.9.x
- Python version 2.7

Furthermore it requires setting up some environmental variables[ref.]

Installing Samples, Binaries and Docker Images

After the prerequisites are installed, the next three steps are the following:

1. Cloning the fabric-samples directory
2. Downloading fabric binaries
3. Download fabric images that will be loaded into docker containers

Hyperledger provides a bash script that performs all three steps automatically.
The provided binaries are the following:

- **cryptogen**: tool for generating cryptographic materials
- **configtxgen**: tool for creating the genesis block
- **peer**: set of commands to perform tasks related to a peer
- configtxlator: tool for translation of fabric data structures to JSON
- discover: tool intended for service discovery
- idemixgen: tool to create identity mixer based Membership Service Providers
- orderer: tool to perform tasks related to a orderer
- fabric-ca-client: tool for operating the fabric certificate authority

The binaries in bold text are of interest for the bootstrapper.

Cryptogen Tool

Fabric is a permissioned system. Meaning that for any operation a valid certificate is needed.
Cryptogen is a tool able to generate cryptographic materal necessary for operating a fabric network. It is intended only for testing purposes. Cryptogen takes as input a YAML file specifying all organizations, their nodes and users. Usually this file is named crypto-config.yaml. The organizations are divided in two types:

- Peer Organizations: organizations providing only peer nodes for the network
- Orderer Organizations: organizations providing only orderer nodes for the network

In reality an organization can provide both node types. They are separated here to provide clearer understanding on how each type affects performance in a test environment. In the input file the template for each organization is the following:

Name	The name of the organization
Domain	The domain of the organization
Node Count	Number of nodes in the organization. The type of nodes depends on the type of organization
User Count	the number of users other than the admin

An example of an input file for a network of one peer organizations with two peers, and one orderer organization with one orderer:

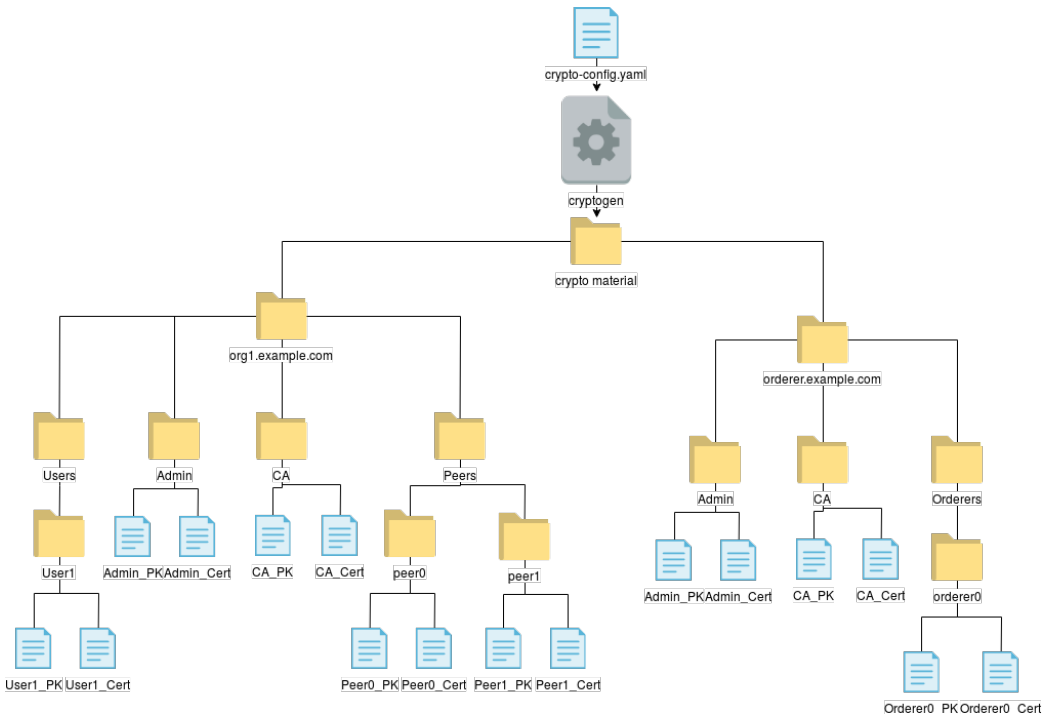
```
---
OrdererOrgs:
- Name: Orderer
  Domain: example.com
  Template:
    Count: 1

PeerOrgs:
- Name: Org1
  Domain: org1.example.com
  Template:
    Count: 2
    Users:
      Count: 1
---
```

The output of the cryptogen tool consists of the following for each organization:

- Private key and signing certificate of the organizations' certificate authority
- Private key and signing certificate for each node of the organization
- Private key and signing certificate of organizations' administrator
- Private key and signing certificate for every user

The last three certificates are issued by the organizations' CA.
Graphical representation of running a cryptogen tool with the input as specified above!ADD LABEL AND REF! is the following:



ADD Description under the picture For the full description of the tool reader is invited to visit [ref]

Configtxgen tool

Configtxgen tool is used for generation of channel configuration artifacts. In the context of the thesis it is used to generate the following artifacts:

- Orderer genesis block
- Channel configuration transaction
- Anchor peer transactions - one for each peer organization. Anchor peers are peers that can be used for cross-organizational communication.

The behaviour of configtxgen is controlled by a YAML file names "configtx.yaml". The configtx.yaml is a complex file containing six main sections. The sections and a high level view of information contained in them is provided below:

1. Organizations:
    - Membership Service Provider (MSP) ID : since fabric transforms identity providers to MSPs, this is the unique ID of the MSP for each organization

- The root certificate directory. In our case generated by the cryptoconfig tool[ref]. This allows to load the root certificate in the orderer genesis block. Now orderers can verify the digital signatures of a member of an organization.
- Policies at the organization level.
- For peer organizations also anchor peers are specified. Anchor peers are peers used for cross-organizational gossip

2. Capabilities: Defines the versions of network components in order to guarantee compatibility
3. Application: Defines the policies at the application level
4. Orderer:
   - Orderer Type: Solo or Kafka
   - Addresses of orderers
   - Addresses of Kafka brokers if Kafka is the ordering service
   - Policies at the orderer level
5. Blocks Configuration:
   - BatchTimeout: The amount of time to wait wait before creating a new block.
   - MaxMessageCount: The maximum number of transactions inside a single block.
   - AbsoluteMaxBytes: The maximum size of a single block
   - PreferredMaxBytes: The preferred block size
6. Channel: Defines policies at the channel level
7. Profiles: Different profiles encodings to use with the configtxgen tool. Each profile represents a consortium of organizations and different profiles are used for different configuration purposes.

Peer Command

Peer command allows administrators to perform specific tasks related to a peer [5].
Peer command has tho following five subcommands [5]:

- peer chaincode

- peer channel
- peer logging
- peer node
- peer version

Each of these subcommands has other subcommands. In the following the commands(including relative subcommands) that are of interest for this thesis are reported. For full description of all possible commands provided by Hyperledger Fabric binaries the reader is invited to visit the Commands Reference page [6] of the Hyperledger Fabric documentation.

Peer Channel Command

Peer Channel Command allows administrators to perform channel related operations on a peer, such as creating a channel, joining a channel, or listing all channel to which the peer belongs to [7].
In the context of the Network Bootstrapper two subcommands were used:

- **peer channel create**: used to create a channel
- **peer channel join**: used to join a peer to a channel

For discussion on how to use this commands appropriately and information on other available commands the user is invited to visit [7]

Peer Chaincode Command

Peer Chaincode Command allows administrators to perform chaincode related operations on a peer, such as installing, instantiating, invoking a chaincode [8]. In the context of the Network Bootstrapper four subcommands were used:

- **peer chaincode install**: Allows to install a chaincode on a given peer
- **peer chaincode instantiate**: Allows to instantiate a chaincode on a given channel. The instantiation should be performed only once.
- **peer chaincode invoke**: Allows to invoke a smart contract on a given peer.
- **peer chaincode invoke**: Allows to query ledger data locally from a given peer. This command doesn't create a transaction.

For discussion on how to use this commands appropriately and information on other available commands the user is invited to visit [8]

Docker Compose

In order to simulate a multi node network on the virtual machine the Docker Compose tool is used. Docker Compose is a tool for defining and running multi-container Docker applications [9].
Docker Compose takes as an input a YAML file specifying containers which to create. For each container we specify an image that will be loaded in the container.
For Hyperledger Fabric the following containers need to be defined:

- A container for each peer in the network
- A container for each orderer in the network
- A container for CLI client to perform administrative task over the network
- If the ordering service is Kafka a container is needed for each kafka broker and for each kafka zookeeper node
- If the CouchDB is used as a state database a container is needed for each instance of the database. Since each peer has its own state database this is equal to the number of peers.

5.2.2 Network Bootstrapper

After understanding all the part required to create and operate a hyperledger fabric network, this subsection explains how the Network Bootstrapper tool operates.
The tool is programmed in the Go Programming Language [10] and Bash Scripting Language [11]. It heavily uses Go Template Package [12] to implement data-driven templates for generating textual output.
The input of the tool is a configuration file written in YAML. The parameters of the configuration file are specified in table 5.2:

Type of Ordering Service	Solo or Kafka.
Number of Orderers	An integer representing the number of orderers in the network. If Kafka is used also the number of brokers and zookeepers is specified.
Peer Organization	An array where each element represents how many peers there are in the organization corresponding to the index of the element.
Batch Timeout	The amount of time to wait wait before creating a new block.
MaxMessageCount	The maximum number of transactions inside a single block.
AbsoluteMaxBytes	The maximum size of a single block.
PreferredMaxBytes	The preferred block size.
Channels	A map where the key is the channel name and the value contains the members of the channel.

Table 5.2: Input Parameters For The Network Bootstrapper

The tool has access to 4 template files. After reading the input, the configuration is used to create 4 output files based on the templates. The output files are specified in table **??**

Chapter 6

Experiments

6.1 Set Up

6.1.1 Testing Environment

All experiments have been performed on a virtual machine with the following specifications:

#CPU Cores	4
#Threads Per Core	1
CPU Operating Frequency	2397.223 MHz
RAM Memory	16 GB
Secondary Memory	16 GB
Operating System	Ubuntu 14.04 64-bit

Table 6.1: Virtual Machine Specifications

6.1.2 Traces Used In The Experiments

All the CAM report traces used in the experiments are generated by the vehicle simulator described in section 5.1.
The input parameters of the vehicle simulator are specified in table 5.1.
All the traces share the same values for a part of the input parameters. The values of the shared parameters are specified in table 6.2.

Seed	Number of Vehicles	Simulation Area Size	CAM Transmission Range	Cam Generation Frequency	Cheating Probability
666	20	1000 m	300 m	1 Hz	10 %

Table 6.2: Shared Simulator Input Parameters for All Traces

In table 6.3 a description of the traces is provided.

Trace ID	Simulation Time	Trace Size(Rounded)	Number Of Reports
Trace 1	11 s	200 kB	220
Trace 2	20 s	400 kB	400
Trace 3	28 s	600 kB	560
Trace 4	44 s	1 MB	880
Trace 5	85 s	2 MB	1700
Trace 6	210 s	5 MB	4200
Trace 7	430 s	10 MB	8600
Trace 8	840 s	20 MB	16800
Trace 9	4250 s	100 MB	85000

Table 6.3: Traces Description

The following vehicle simulator input parameters depend on the experiment that is being executed:

- **Number Of Stations**: Since each stations represents a peer in the blockchain network, the number of stations is equal to the number of peers. The number of peers is specified at the start of each experiment.
- **Minimum Reports Per Transaction**: The default value for all experiments is 20 reports uploaded per transaction. In the experiments where this value is changed it will be specified at the start of the experiment.
- **Minimum CAMs Per Query or Validate**: The default value for all experiments is 20 CAMs queried or validated per transaction. In the experiments where this value is changed it will be specified at the start of each experiment.

6.1.3 Additional Information

As already specified the number of stations in each experiment equals the number of peers. In the experiments the reports generated are divided equally among all stations, *i.e.* each stations receives the same number of reports to upload to the blockchain.
In the experiment whenever a client is performing an operation, the operation will be performed on the peer corresponding to the stations that received the reports. For example if the station #1 wants to upload reports to the blockchain the client will invoke the smart contract on peer #1.
This behavior mimics the design of the architecture since the client located in the station will send it's requests to the peer located in the same station.

6.2 Experiments Part 1

In the first part of experiments the network is benchmarked using the developed benchmarking tool specified in XXX.
These experiments are performed for two reasons:

1. To evaluate the performance of Hyperledger Fabric and developed applications

2. To compare the results with the results obtained in the previous thesis. The previous results can be located in chapter 7 in XXX.

The results of the previous thesis are included in the plots to directly observe the differences in the metrics.

6.2.1 Differences With Respect to The Previous Thesis

Number Of Reports

The size of traces is kept the same as in the previous thesis. Since the encoding of reports differs between the two thesis, the total number of reports(and CAMs) needed to achieve the same trace size is about two times higher in the current thesis.

Endorsing Peers

In all experiments in the previous thesis, except the ones in a multi-processing environment, all the operations are performed on the same peer. This means even though different stations receives different reports, all the stations invoke

the smart contract operations on the same peer.
In this thesis the station that receives reports will upload them to the peer corresponding to that station.

6.2.2 Time to access the state database

Access to a value stored in the blockchain generally takes linear time with respect to the size of the blockchain. To allow faster access, structures to index the blockchain need to be created. In hyperledger fabric this is accomplished by introducing a database that keeps a copy of all values stored in the blockchain. The database acts like a cache, where the last stored value for each key is stored. This allows the speed of access to a value to be constant in time, but introduces an additional copy of the last stored value.

Objective of the experiment

To verify that time to query a set of values does not depend on the size of the blockchain. Additionaly to compare differences in the access speed between two state database solution: CouchDB and LevelDB

Experiment Implementation

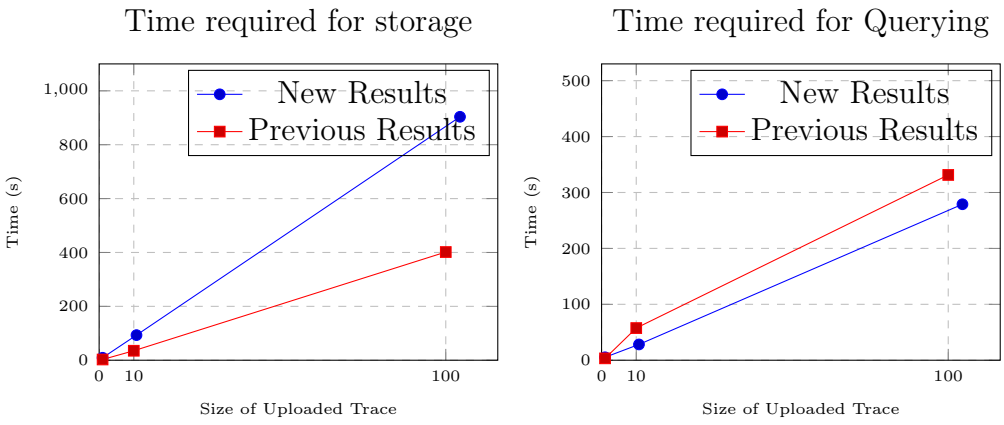
Upload three CAM reports traces of different sizes(1 MB, 10 MB, 100 MB) to the blockchain. For each trace measure the time to retrieve all the uploaded CAMs. The experiment is repeated 5 times for each state database.

Experiment Configuration

Number of Organizations	1
Number of Peers per Organization	2
Total number of Peers	2
CPU Peers	100%
CPU Orderers	100%
Number of Orderers	1
Ordering service	Solo
State Database	LevelDB and CouchDB
Vehicle simulator traces	Traces 4, 8, 10

Table 6.4: Network Configuration

Experiment Results



(a) Storage Time

(b) Query Time

Figure 6.1: State Database Access Results

Analysis and Comparison

Storage Time: As expected, the amount of time needed to store increases linearly with the size of the trace to be stored.
In the previous thesis we see that the results have about two times smaller slope than the new results. This can be explained because in the previous thesis TLS communication was not enabled between the nodes of the network. This means that in new experiments each transaction had to be signed and verified which introduces additional latency.
Querying Time: As specified in the experiment description, access to a key-value pair should be constant in time. This is verified by the experiment because the query time increases linearly with the amount of data(keys) that is queried.
Unlike with the storage time results, querying times are almost the same in both theses. This is because the querying operation simply reads the values from the state database in the peer on which the operation was requested. It doesn't involve any communication between nodes in the network, thus avoids the extra TLS verification steps.

6.2.3 Time to access historical data of the blockchain

As seen in the previous experiment hyperledger fabric provides a database to access the last value for a given key instantaneously. But to retrieve the historical values for a given key, the blockchain needs to be traversed and the values rebuilt.

Objective of the experiment

To verify that retrieving historical values for a given key linearly depends on the size of the blockchain.

Experiment Implementation

Upload and overwrite a 1 MB raw trace 100 times into the blockchain. After each overwrite, retrieve the historical values for a given key.

Experiment Configuration

The experiment configuration is the same as in the experiment 5.2 with only the 1 MB trace (Trace 4) being used.

Experiment Results

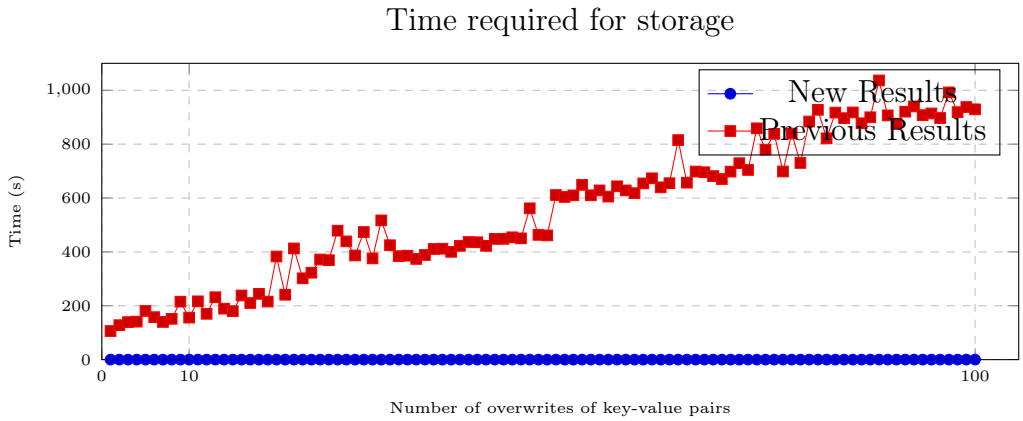


Figure 6.2: Results for Accessing Historical Data

Analysis and Comparison

This is an unexpected result. Since a retrieving history for a key, retrieves all the transactions that modified it, and all the previous values of the key it should have to traverse the blockchain. But surprisingly it seems that the time to retrieve it remains constant irregardless of the number of rewrites. It seems that the peers maintain a cache of historical data that allows constant retrieval. The experiment was retried with larger traces and more overwrites but the results remain the same.
The results in the previous thesis show the expected linear behavior. The experiment was retried by running on the old version of fabric(1.1.0) and expected results were obtained. Therefore it seems that this is behavior has changed with the new version of fabric. Currently no documentation on how to disable this cache behavior is present.

6.3 Storage Overhead Experiments

6.3.1 Storage Overhead of the Blockchain With Respect to Trace Size

The overhead introduced by the blockchain can be divided in two parts:

1. The overhead of the blocks, which consists of:
    - The hash of the current block

- The hash of the previous block in order to build the blockchain
- Block metadata: block number, signature and public key of the block creator, time and date when the block was created, etc.

2. The overhead of transactions in the block. The transaction overhead consists of:
   - Transaction metadata: chaincode name, chaincode version, etc.
   - Signature of the client that invoked the transaction
   - Proposal: the input parameters provided by the invoking client
   - Endorsement: a set of signatures from endorsing peers. In our case from a single peer

Objective of the Experiment

To analyze the amount of overhead when storing data of different sizes in the blockchain.

Experiment Implementation

Start multiple blockchain networks and upload a different amount of data to each one. After the upload, overhead of blockchain structure is analyzed. The overhead is computed as the percentage of increase with respect to the trace size before the upload.
In this experiment we will upload the traces in the raw format. In other words, for each transaction we store the the transaction reports at a randomly generated key. This is done to allow us to calculate a realistic overhead since if the reports were uploaded with processing there would be extra information stored, such as an extra key for each received CAM.

Experiment Configuration

The network configuration is the same as in the table 5.1 with the following differences:

- Only LevelDB is used. This is because the blocks are the same regardless of the state database.
- All of traces specified in [trace table] are used

The configuration of the blocks has an effect on the overhead. In this experiment we will keep the block configuration same for all traces, to fairly evaluate the effect of trace size. In one of the following experiments, different block configuration will be evaluated.

BatchTimeout	2
MaxMessageCount	10
AbsoluteMaxBytes	99 MB
PrefferedMaxBytes	512 kB

Table 6.5: Blocks Configuration

Experiment Results

Analysis and Comparison

6.3.2 Effect of Varying the Number of Transactions per Block

Experiment Objective

As seen in the previous experiment there are two overheads: the overhead of a block and the overhead of a transaction. Objective of this experiment is to calculate the effect of block overheads by manipulating the number of transactions stored in each block.

Experiment Implementation

Creating multiple blockchain network and uploading the same trace in the raw format(without processing) to all of them. Each blockchain will have a different *MaxMessageCount* parameter. To remind the *MaxMessageCount* parameter controls the maximum number of transactions per block.

Experiment Configuration

The network configuration of the experiment is the same as in the table 5.1. with the following differences:

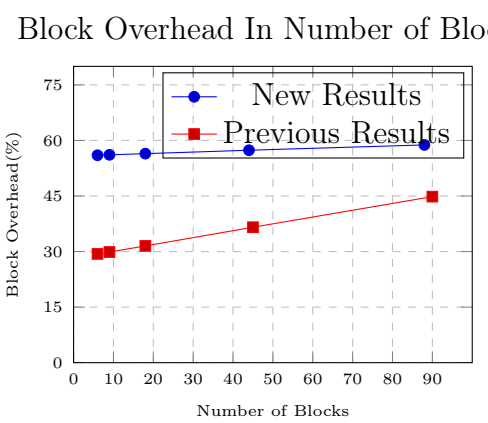
- Since we are measure the number and size of the blocks only LevelDB is used as the state database
- Only trace 4 (1 MB) is used.

The blocks configuration is the following:

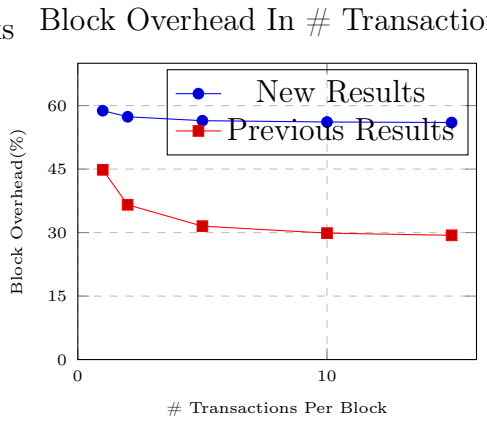
BatchTimeout	2
MaxMessageCount	1, 2, 5, 10, 15
AbsoluteMaxBytes	99 MB
PrefferedMaxBytes	512 kB

Table 6.6: Blocks Configuration

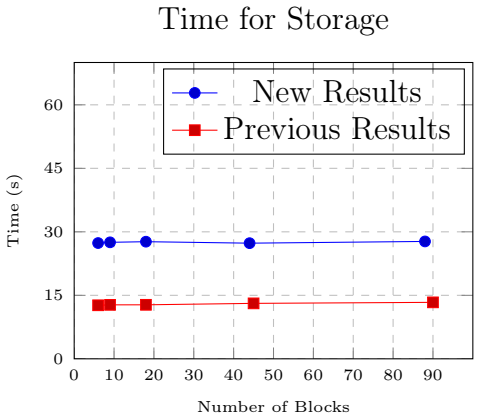
Experiment Results



(a) Block Overhead In # Blocks



(b) Block Overhead In # Transactions



(c) Storage Time in # Blocks

Figure 6.3: Results Of Varying The Number of Transactions per Block

6.4 Scalability Experiments

One of the most important characteristics of hyperledger fabric is that a chaincode invocation has to be executed only on the endorsing peers. This allows for massive scalability and parallelism. In the context of the thesis the endorsing policy requires only a single endorsing peer(station) which means that increasing the number of peers should not affect the performance in a meaningful manner. While increasing the parallelism will improve the performance. In this section we will deal with scalability of the hyperledger fabric.

6.4.1 Raw uploading with many peers in a single organization

Objective of experiments

To analyze performance metrics of storage without any processing with dependence on the number of peers in the hyperledger fabric network.

Experiment Implementation

Starting multiple blockchain networks with different number of peers and uploading a 10MB CAM reports trace to each network, without processing. The following metrics are measured:

- Average CPU consumption
- RAM usage
- HDD usage
- Time required for the storage

The experiment is repeated five times.

Experiment Configuration

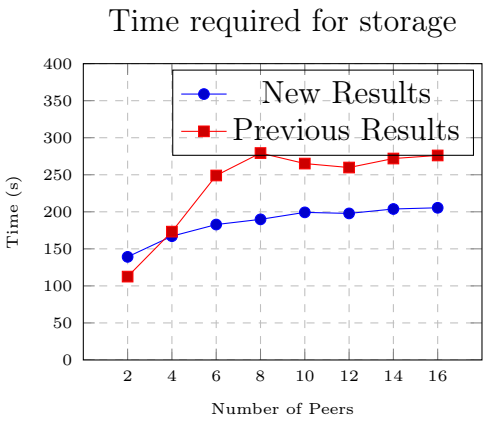
Number of Organizations	1
Number of Peers per Organization	2,4,6,8,10,12,14,16
Total number of Peers	2,4,6,8,10,12,14,16
CPU Peers	5%
CPU Orderers	100%
Number of Orderers	1
Ordering service	Solo
State Database	CouchDB
Vehicle simulator traces	Trace 7(10 MB)
Type of storage	Raw

Table 6.7: Network Configuration

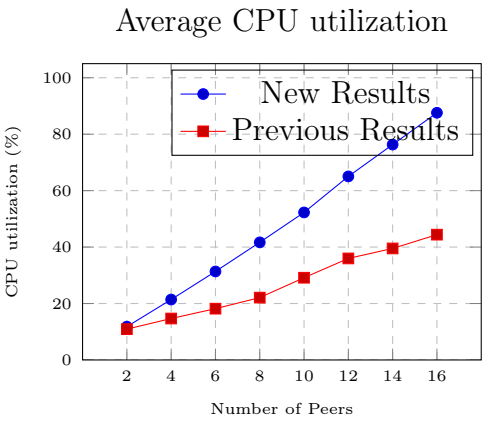
BatchTimeout	2
MaxMessageCount	10
AbsoluteMaxBytes	99 MB
PrefferedMaxBytes	512 kB

Table 6.8: Blocks Configuration

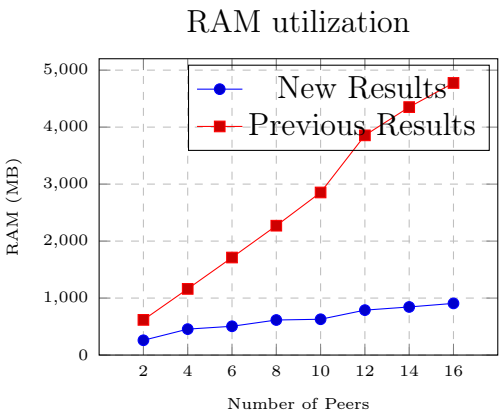
Experiment results



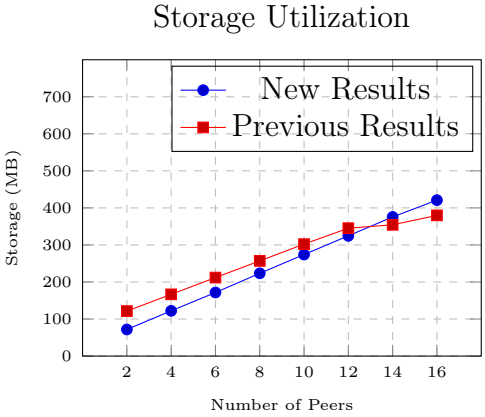
(a) Time Required For Storing



(b) CPU Utilization



(c) RAM Utilization



(d) Secondary Memory Utilization

Figure 6.4: Raw Storage of Reports Results

Analysis and Comparison

Storage Time: The time to store raw reports initially increases linearly with the number of peers. When the number of peers reaches 10 it starts to stabilize around 200 s.
The same is behavior in results of the previous thesis. Although in this thesis the absolute times are a bit lower. This is probably due to chaincode implementation differences.
CPU Utilization: The average CPU utilization increases linearly with number of peers. There are two reasons for this:

1. Whenever the number of peers is increased, a part of the transactions is executed on each peer.

2. Independently of who is the endorsing peer(peer that executed the transaction) all the peers must commit the transaction.

The same behavior is observed in the results of the previous thesis. Although the the slope is about two times higher in new results. The reason is the following, in the other thesis when the number of peers is increased the transaction are all executed on the same peer. Therefore the increase is only due to the commitment of the transactions on other peers. While in this thesis the set of transaction is divided on all peers which causes an additional CPU utilization per peer.
RAM Utilization: The RAM Utilization increases linearly with the number of peers.
The same linear behavior is observed in the results of the previous thesis. There is a striking difference between the actual RAM usage: in this thesis the amount of RAM used is about five times lower. This is an extremely strange result. In the further experiments with multiple organizations, the behavior will equalize. But when there is a single organization there is much less main memory requirements. This result has been analyzed in detail and repeated many times, but still no valid reason is found.
Storage Utilization: There is a linear dependency between number of peers and secondary memory use. This makes sense since each peer stores a copy of the ledger and the state database.
We see the same behavior in fig 7.7 in [Michele's Thesis Reference].

6.4.2 Storage, Querying and Validation with many peers in a single organization

Objective of the experiment

To analyze performance metrics of storage with processing, querying and validation of CAM reports in dependence of the number of peers in the network.

Experiment Implementation

Starting multiple hyperledger fabric networks with various number of peers. For each network the following operations are performed:

- Uploading the trace with processing

- Query all CAMs stored in the network
- Validate all CAMs stored in the network

For each operation we measure the same metrics as specified in experiment 5.3.1.
The experiment is repeated five times.

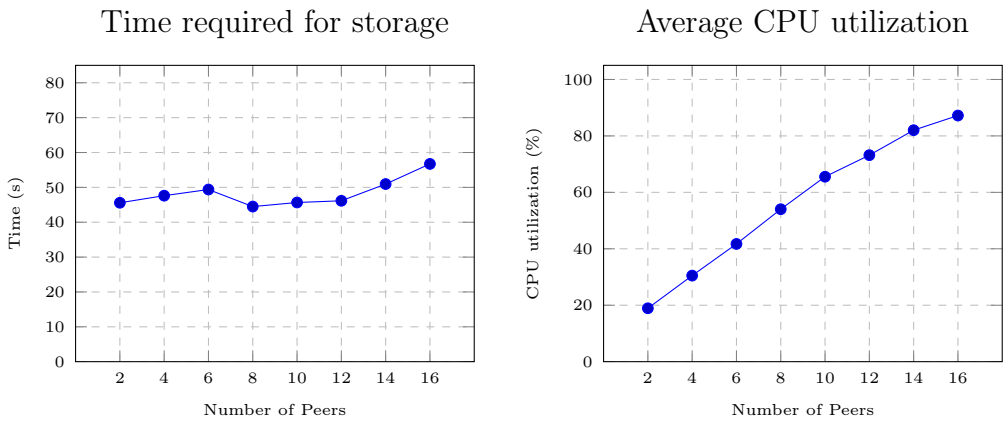
Experiment Configuration

The network configuration is the same as in table 5.1 with the following differences:

- The CPU limit of each peer is increased to 10%
- Trace 4 (1 MB) is used
- Type of storage is with processing

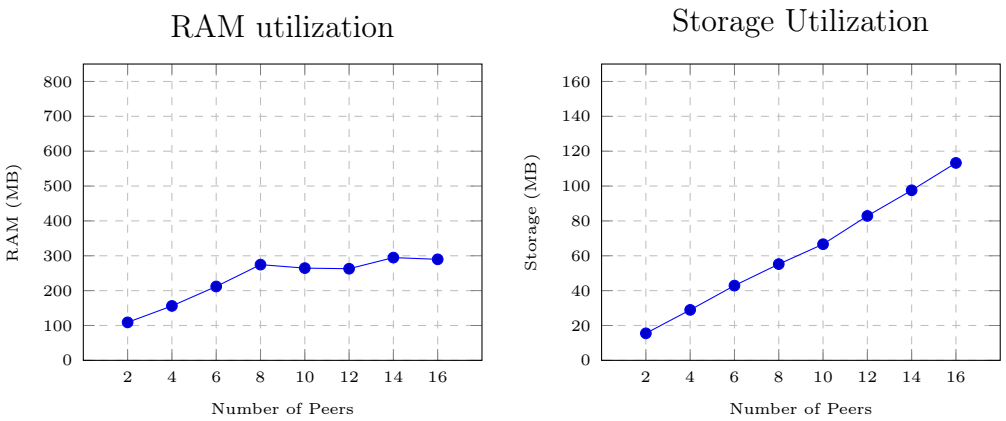
The blocks configuration is the same as in table 5.2.

Storage Results



(a) Upload Time

(b) CPU Utilization



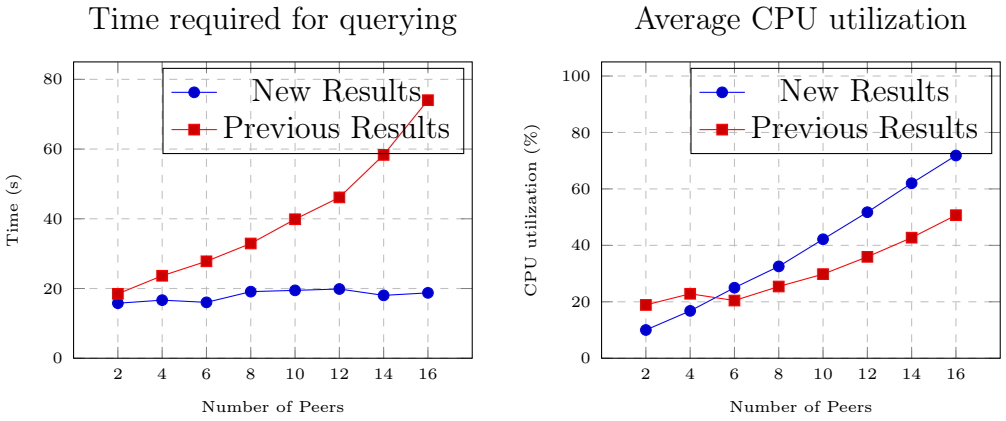
(a) RAM Utilization (b) Secondary Memory Utilization

Figure 6.6: Storage of Reports Results

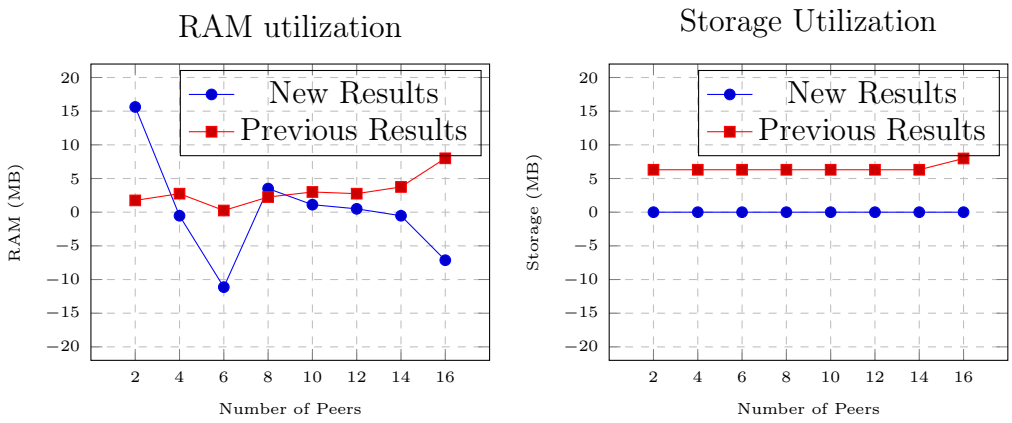
Analysis

The behavior of storing CAM Reports with processing is relatively similar to the storage without processing with the difference that the time requirement and RAM utilization increase with a much smaller slope. This is probably because the trace is 10 times smaller then in the raw storage experiment so the overhead is not as pronounced.

Query Results



(a) Upload Time (b) Average CPU Utilization



(c) RAM Utilization (d) Secondary Memory Utilization

Figure 6.7: Querying of all CAMs results

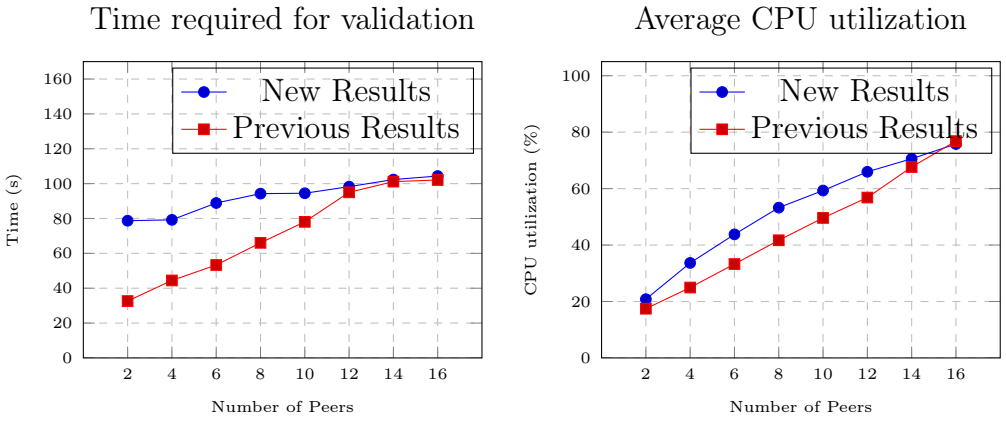
Analysis and Comparison

Querying Time: The time required for querying doesn't increase substantially with the number of peers. This is reasonable since the query operation simply retrieves the value stored in the state database. There is no need for peer to peer, or peer to orderer communication. There is a small increase due to the fact that we are connecting to more peers, so there is a overhead for establishing the connection.
This is different from the results seen in the previous thesis, where the time growth is linear. In that experiment in all cases the data is fully queried from one peer. The linear increase can be explained that since there are multiple peers computing for the CPU time, the context switch reduce the CPU time from the querying peer thus leading to a time increase. This also justifies the results in the current thesis since the querying is done in part of each peer, so the CPU time allocated to each peer is used on the querying operation.

CPU Utilization: The average CPU utilization increases linearly with number of peers. This is a normal result since each peer is allocated a portion of the CPU time and even if no transaction were issued on the peer, it will still perform operations internal to the fabric platform.
In the results of the previous thesis there is also a linear increase but with a smaller slope. This makes sense if we divide the CPU utilization on the idle part and the issued transaction part. In the other thesis all the peers contribute to the "idle" part and only the querying peer contributes to the issued transaction part, whereas in this thesis all peers contribute to both parts it stands to expect a larger slope in the CPU utilization.

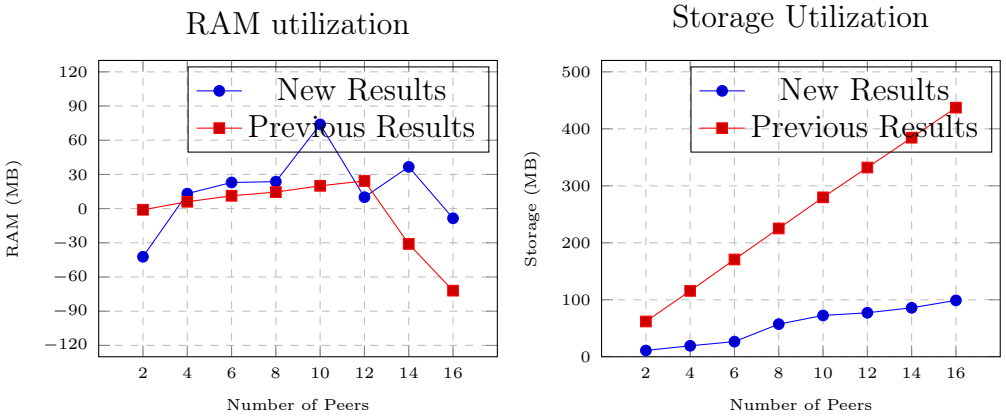
RAM Utilization: The RAM behavior fluctuates around zero assuming small values. This can be explained from the fact that querying operation shouldn't introduce any long term extra information in the main memory. So measuring the difference before and after querying simply depends on the current action of the peer, such as performing hyperledger fabric protocols.
Storage Utilization: There is no increase in storage since querying operation simply returns the values from the state database.
The same results for RAM and Storage Utilization can be seen in the results of the previous thesis.

Validation Results



(a) Upload Time

(b) Average CPU Utilization



(c) RAM Utilization

(d) Secondary Memory Utilization

Figure 6.8: Validation of all CAMs results

Analysis and Comparison

Validation Time: As with querying, the validation operation increases very slowly with the increase in peers. Again the increase is probably due to the overhead of having to connect to multiple peers.
In the results of the previous thesis we see that the time increases at a much faster pace. This is for the same reason as in the querying results: the validation is done fully on one peer and therefore the CPU utilization of other peers does not contribute to the speed of validation. While in this thesis the validation process is done in parts on all peers.
CPU Utilization:The CPU utilization grows linearly with the number of peers. The increase is higher then with the querying operation because all the peers must do the extra work of writing the blocks of committed transaction.
The same behavior is seen in the previous thesis results. **RAM Utilization**: The RAM utilization seems to assume random values with respect with the number of peers. In all of the five experiments no trends were visible. The conclusion is that the validate operation doesn't introduce a significant main memory increase. In other words since the RAM is measure as the difference in main memory before and after an operation, the values completely depend on what action the peers are performing at the time of measurement: committing a block, performing keep alive checks, etc.
In the results of the previous thesis we see a similar but different behavior. There the RAM usage seems to grow linearly in the start and drop due to memory swapping with the high number of peers. The values are typically very small, lower than 20 MB.
Storage Utilization: We have a linear increase in storage due to validation values added to the blockchain and the validation transactions being written to the blockchain.
In the previous results we have the same behavior but with higher amount of data. This is because in the previous thesis the validation results are stored with the same key as the CAM. This means that the CAM will be duplicated in the blockchain.

6.4.3 Raw uploading with many organizations having two peers per organization

Objective of experiments

To analyze performance metrics of storage without any processing with dependence on the number of organizations in the hyperledger fabric network.

The results are compared with the experiment 5.3.1 which contains equivalent number of peers in a single organization.

Experiment Implementation

Starting multiple blockchain networks with different number of organization and each having two peers. Measure the following metrics for the upload of 10 MB of data without processing:

- Average CPU consumption
- RAM usage
- HDD usage
- Time required for the storage

The experiment is repeated five times.

Experiment Configuration

Number of Organizations	1, 2, 3, 4, 5, 6 ,7, 8
Number of Peers per Organization	2
Total number of Peers	2, 4, 6, 8, 10, 12, 14, 16
CPU Peers	5%
CPU Orderers	100%
Number of Orderers	1
Ordering service	Solo
State Database	CouchDB
Vehicle simulator traces	Trace 7(10 MB)
Type of storage	Raw

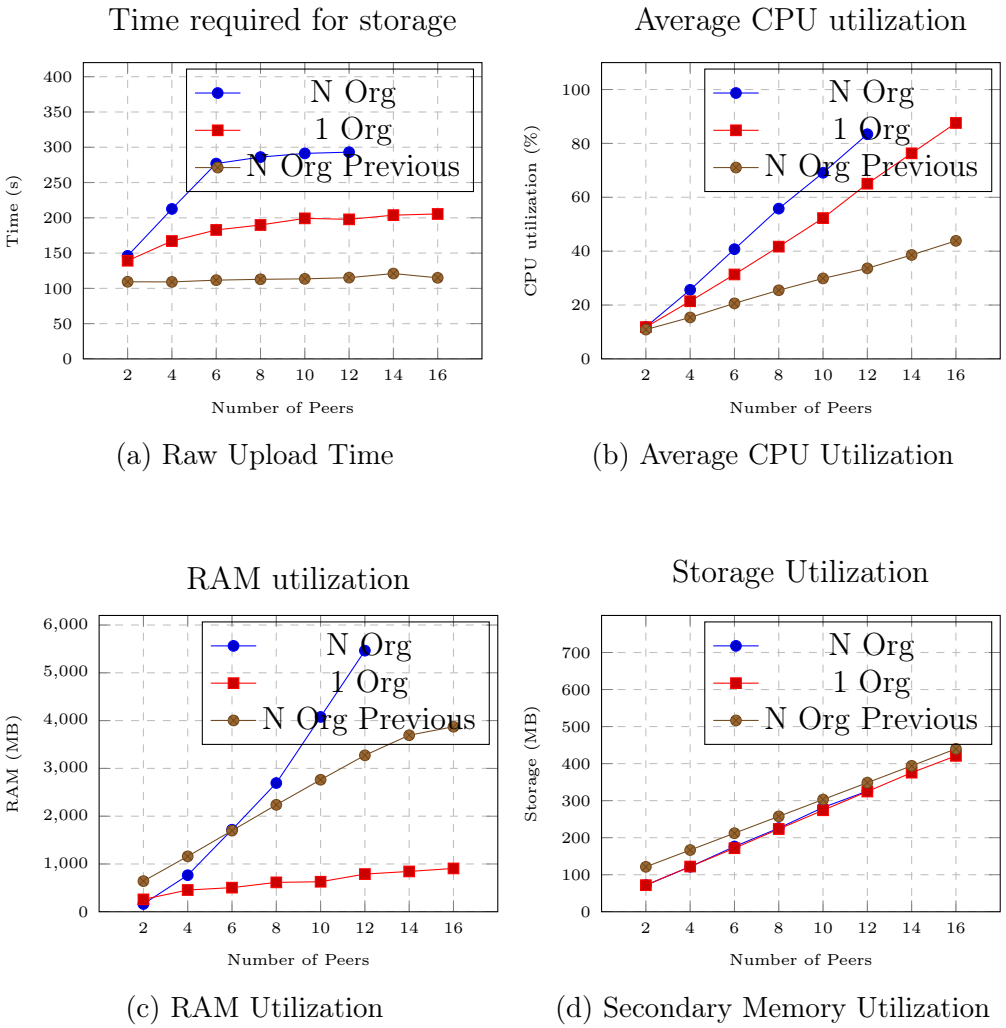
Table 6.9: Network Configuration

BatchTimeout	2
MaxMessageCount	10
AbsoluteMaxBytes	99 MB
PrefferedMaxBytes	512 kB

Table 6.10: Blocks Configuration

Experiment results

The experiments with 7 and 8 organizations could not be performed due to limitations in the testing environment.



(a) Raw Upload Time

(b) Average CPU Utilization

(c) RAM Utilization

(d) Secondary Memory Utilization

Figure 6.9: Raw Storage of Reports Results

Analysis and Comparison

Validation Time: The time behavior follows the same trend as the behavior of an equivalent number of peers in a single organization. While the trend is the same there is an actual increase in time requirements. This is probably because the communication between peers in different organization is performed differently then communication of peers inside an organization. In the results of the previous thesis we see a completely different behavior. In this case the increase in the number of organizations keeps the storage time at a constant. The author also finds this behavior strange.
From a speculative point of view, in the other thesis there could have been an error in other organizations joining the channel. This would lead to only the first organizations having the blockchain and thus keep the time constant. But this possibility was probably explored by the author.
CPU Utilization: The average CPU utilization grows linearly with the number if peers. The slope is slightly larger in the case of multiple organizations.
The same behavior can be seen in the previous results.
RAM Utilization: As stated in the experiment 5.2.1 this is one of the stranger results in the experiments. As soon as the number of organizations is increased the RAM requirements increase exponentially. This implies that there is a lot of extra information that needs to be kept in peer's main memory when dealing with multiple organizations.
In the results of the previous thesis we see almost the same rate of increase with multiple organizations. But in that case the single organization requirements are more or less equivalent to the multiple organizations requirements. Since any real implementation would have multiple organizations, the results of this experiments should be taken as the resource requirements.
Storage Utilization: The increase in storage grows linearly with the number of peers. As expected, the storage consumption is equivalent as in the case of a single organization.

6.4.4 Raw uploading with two organizations and many peers per organization

Objective of experiments

To analyze performance metrics of storage without any processing in the blockchain network consisting of two organizations and many peers per organization. These results can be considered as a middle ground between experiments 5.3.1 and 5.3.3.

Experiment Implementation

Starting multiple blockchain networks with two organizations and varying the number of peers per organization. Measure the following metrics for the upload of 10 MB of data without processing:

- Average CPU consumption
- RAM usage
- HDD usage
- Time required for the storage

The experiment is repeated five times.

Experiment Configuration

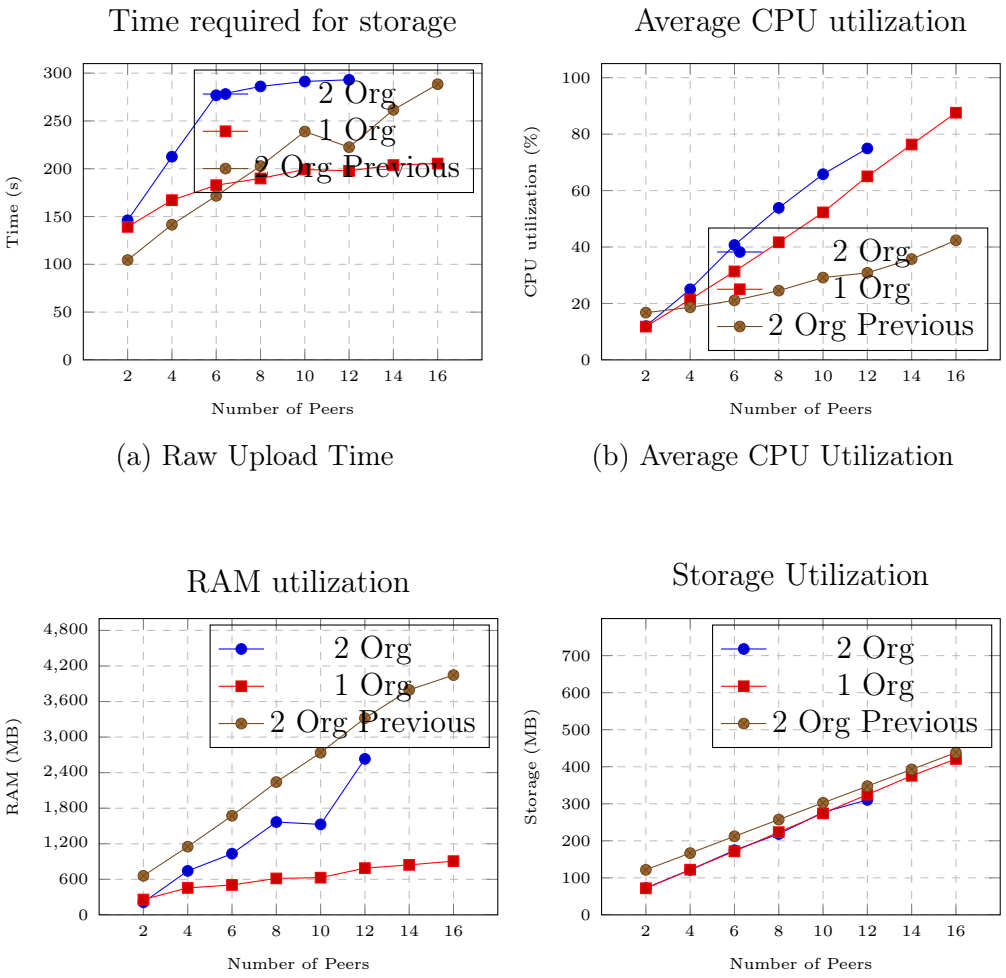
Number of Organizations	2
Number of Peers per Organization	1, 2, 3, 4, 5, 6, 7, 8
Total number of Peers	2, 4, 6, 8, 10, 12, 14, 16
CPU Peers	5%
CPU Orderers	100%
Number of Orderers	1
Ordering service	Solo
State Database	CouchDB
Vehicle simulator traces	Trace 7(10 MB)
Type of storage	Raw

Table 6.11: Network Configuration

BatchTimeout	2
MaxMessageCount	10
AbsoluteMaxBytes	99 MB
PrefferedMaxBytes	512 kB

Table 6.12: Blocks Configuration

Experiment results



(a) Raw Upload Time

(b) Average CPU Utilization

(c) RAM Utilization

(d) Secondary Memory Utilization

Figure 6.10: Raw Storage of Reports Results

Analysis and Comparison

Validation Time: The time behavior follows the same trend as the behavior of an equivalent number of peers in a single organization. The values are a bit larger due to the overhead of dealing with two organizations.
CPU Utilization: The average CPU utilization grows linearly with the number if peers with a slightly larger slope then the case of a single organization.
RAM Utilization: The RAM utilization grows linearly with a much higher

slope then a single organization with an equivalent number of peers. It sits somewhere between the usage with more organizations and a single organization. This is also leads to believe that increasing the number of organizations is more detrimental to the main memory usage then the number of peers.
Storage Utilization: The increase in storage grows linearly with the number of peers. As expected, the storage consumption is equivalent as in the case of a single organization.
All of the similar results can be observed in the previous thesis.

6.4.5 Raw uploading with varying number of orderers

Objective of experiments

To analyze performance metrics of storage without any processing in the blockchain network consisting of different number of kafka orderers. The number of kafka zookeeper nodes and kafka brokers is chosen as the minimum number necessary to guarantee fault tolerance.

Experiment Implementation

Starting multiple blockchain networks with a single organization containing two peers. Each network contains a different number of orderers. Measuring the following metrics for the upload of 10 MB of data without processing:

- Average CPU consumption
- RAM usage
- HDD usage
- Time required for the storage

The experiment is repeated five times.

Experiment Configuration

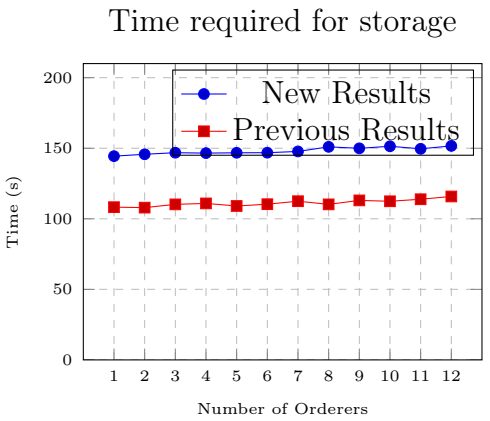
Number of Organizations	1
Number of Peers per Organization	2
Total number of Peers	2
CPU Peers	5%
CPU Orderers	5%
Number of Orderers	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
Number of Zookeeper Nodes	3
Number of Kafka Brokers	4
Ordering service	Kafka
State Database	CouchDB
Vehicle simulator traces	Trace 7(10 MB)
Type of storage	Raw

Table 6.13: Network Configuration

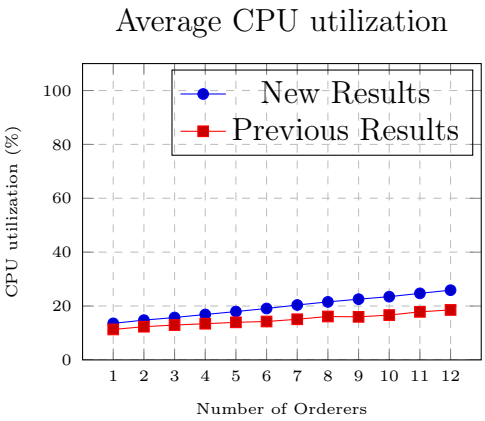
BatchTimeout	2
MaxMessageCount	10
AbsoluteMaxBytes	99 MB
PrefferedMaxBytes	512 kB

Table 6.14: Blocks Configuration

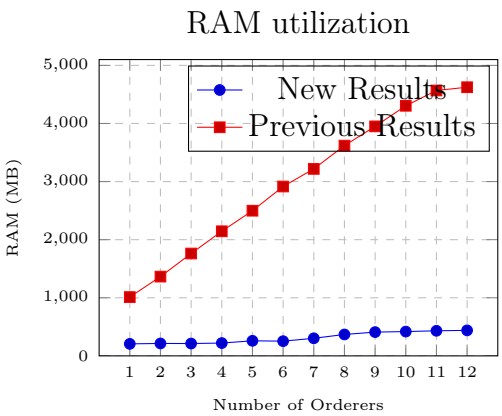
Experiment results



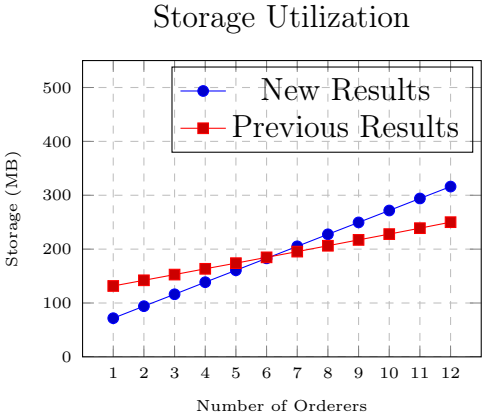
(a) Raw Upload Time



(b) Average CPU Utilization



(c) RAM Utilization



(d) Secondary Memory Utilization

Figure 6.11: Raw Storage of Reports Results

Analysis and Comparison

Storage Time: The storage time remains constant with an increase in the number of orderers. This is reasonable since the transaction processing is performed on the peers.
The same results are observed in the previous thesis.
CPU Utilization: The CPU utilization grows linearly, but with a small slope. This is because most of the heavy processing is in the transaction execution which is performed by the endorsing peers.

The same results are observed in the previous thesis.
RAM Utilization: The RAM utilization grows very slowly with the number of orderers.
Linear behavior is observed in the previous results. But the RAM utilization is much higher in the other thesis. As in the case of the single organization the reason behind this difference is unclear.
Storage Utilization: The increase in storage grows linearly with the number of orderers. This is to be expected, since each orderer stores a copy of the blockchain.
The same behavior is observed in the previous thesis.

6.4.6 Querying and Validation with varying number of orderers

Objective of the experiment

To analyze performance metrics of querying and validation of CAM reports in dependence of the number of orderers in the network.

Experiment Implementation

Starting multiple hyperledger fabric networks with a single organization containing two peers. Each network contains a different number of orderers. We upload CAM reports with processing to all the networks. For each network the following operations are performed:

- Query all CAMs stored in the network
- Validate all CAMs stored in the network

For each operation we measure the same metrics as specified in experiment 5.3.1.
The experiment is repeated five times.

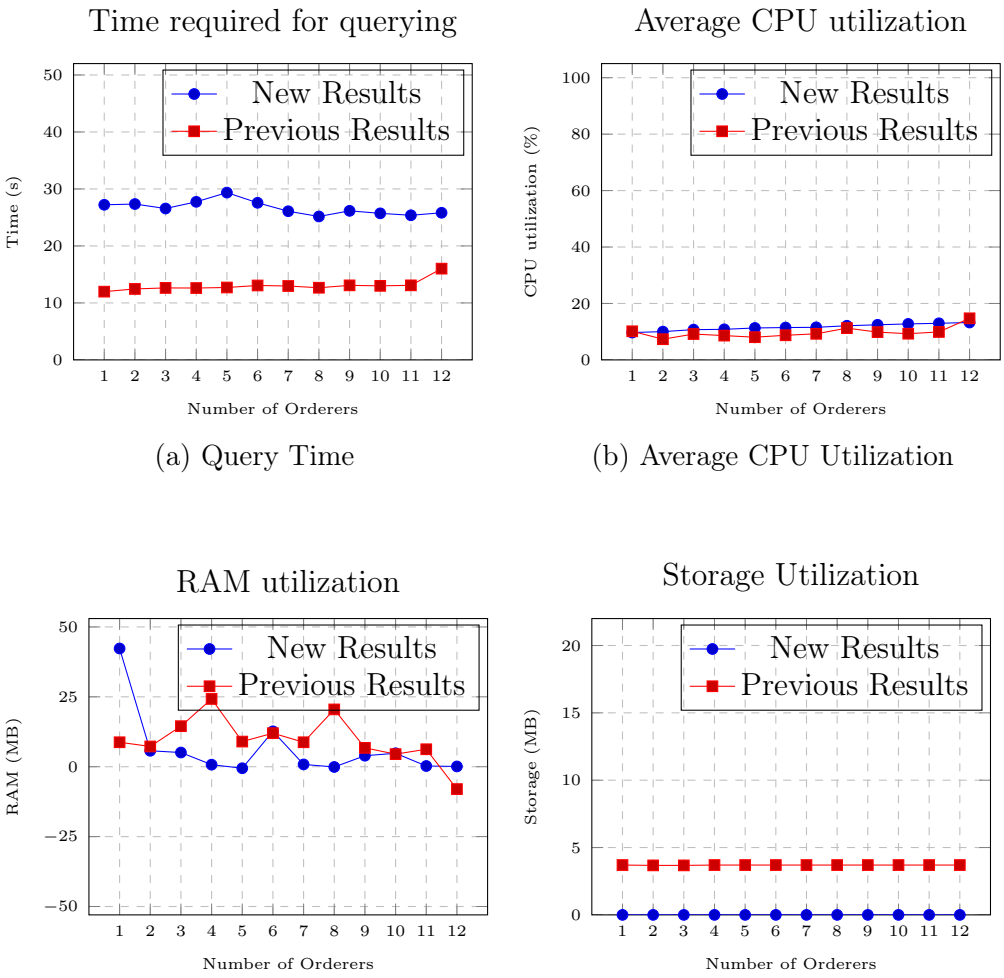
Experiment Configuration

The network configuration is the same as in table 5.7 with the following differences:

- Trace 4 (1 MB) is used
- Type of storage is with processing

The blocks configuration is the same as in table 5.8.

Query Results



(a) Query Time

(b) Average CPU Utilization

(c) RAM Utilization

(d) Secondary Memory Utilization

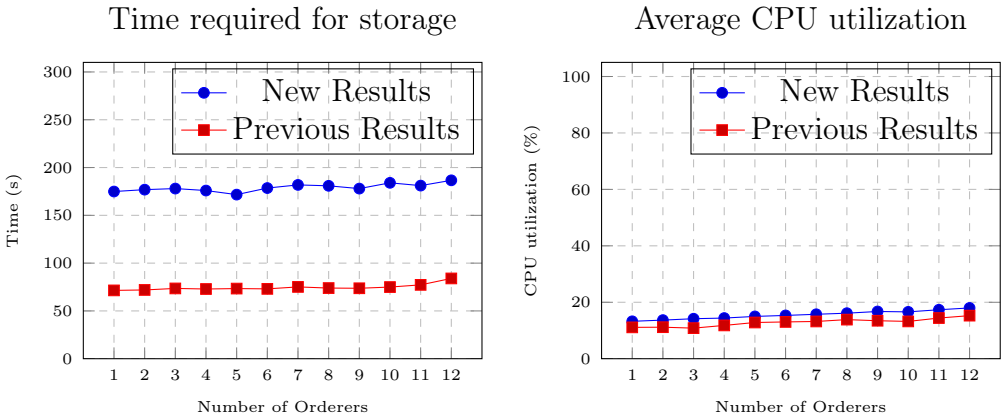
Figure 6.12: Query of all CAMs Results

Analysis

Query Time: Query time remains constant with respect to the number of orderers.
In the previous thesis similar results are observed. The difference the time is about 50% smaller in the other thesis. This is reasonable since even though the traces are same in absolute size, since the size of the report is about two times smaller in this thesis it implies that the number of CAMs to query is two times higher.

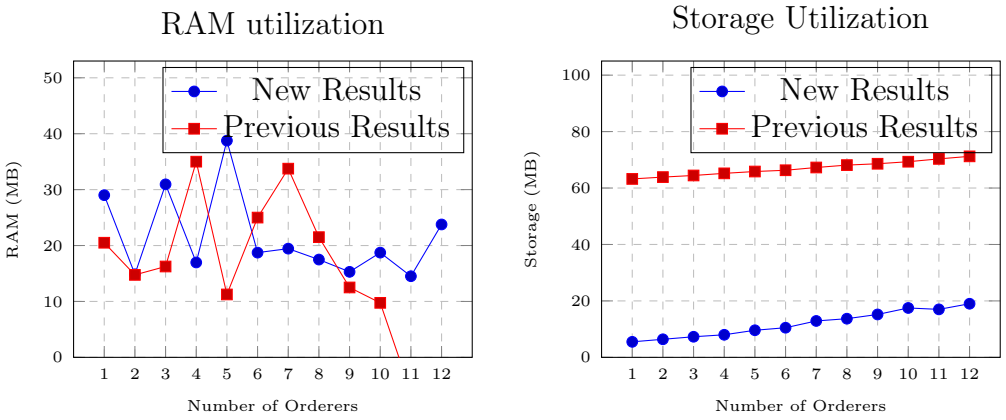
CPU Utilization: CPU Utilization retains more or less a constant value with respect to the number of orderers.
Similar results is observed in the previous thesis.
RAM Utilization: The RAM behavior fluctuates around zero assuming small values. This can be explained from the fact that query operation shouldn't introduce any long term extra information in the main memory. So measuring the difference before and after validation simply depends on the current action of the peer, such as performing hyperledger fabric protocols.
Similar results is observed in the previous thesis.
Storage Utilization: Query operations do not require any extra storage.

Validation Results



(a) Validation Time

(b) Average CPU Utilization



(c) RAM Utilization

(d) Secondary Memory Utilization

Figure 6.13: Validation of all CAMs Results

Analysis

Validation Time: Validation time remains constant with the increase in the number of peers.
A similar behavior is observed in the previous thesis. In that thesis the time is about half of the results in the current thesis. Reason for this is twofold:

1. the amount of CAMs is doubled in the current thesis
2. The validation of this thesis also performs the deletion of reception information so more time is needed

CPU Utilization: CPUS use increases very slowly with the number of orderers. This makes sense since the heavy processing is performed on the peers.
Similar result is observed in the previous thesis.
RAM Utilization: The RAM behavior fluctuates around zero assuming small values. This can be explained from the fact that verification operation shouldn't introduce any long term extra information in the main memory. So measuring the difference before and after validation simply depends on the current action of the peer, such as performing hyperledger fabric protocols.
Similar result is observed in the previous thesis.
Storage Utilization: Storage grows linearly with the number of orderers. This is because each orderer contains a copy of the blockchain and each validate operation increases the blockchain size.
Similar result is observed in the previous thesis. In the other thesis the storage used is much higher. This is because in the previous thesis validation operation duplicates the CAM.

6.4.7 Querying and Validation with respect to the transaction size

Objective of the experiment

To analyze performance metrics of storage with processing, querying and validation of CAM reports in dependence of how many CAMs are stored, queried and validated by each transaction.

Experiment Implementation

Starting multiple hyperledger fabric networks with a single organization containing two peers. In each network operations are performed with transaction of different sizes. The same 1 MB trace is used for all networks. For each network the following operations are performed:

- Upload the trace with processing
- Query all CAMs stored in the network
- Validate all CAMs stored in the network

For querying and validate operations the following metrics are measured:

- Average CPU consumption
- RAM usage
- HDD usage
- Time required for the operation
- Number of new blocks created for the operation
- Total size of the new blocks

The experiment is repeated five times.

Experiment Configuration

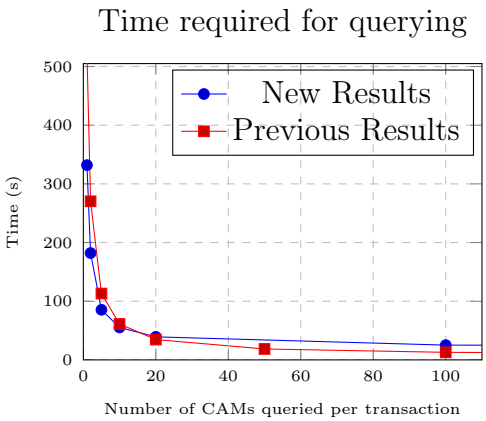
Number of Organizations	1
Number of Peers per Organization	2
Total number of Peers	2
CPU Peers	5%
CPU Orderers	5%
Number of Orderers	1
Ordering service	Solo
State Database	CouchDB
Vehicle simulator traces	Trace 4 (1 MB)
Type of storage	With Processing
CAMs validated or queried per transaction	1, 2, 5, 10, 20, 100, 200

Table 6.15: Network Configuration

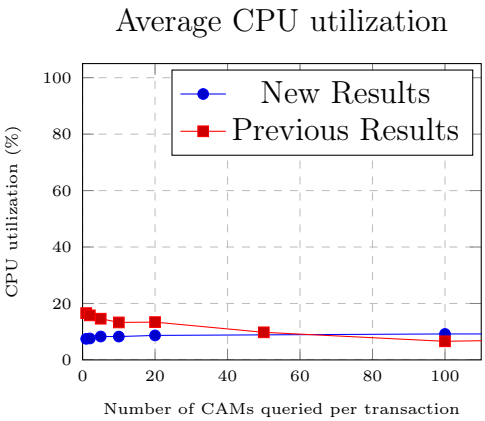
BatchTimeout	2
MaxMessageCount	10
AbsoluteMaxBytes	99 MB
PrefferedMaxBytes	512 kB

Table 6.16: Blocks Configuration

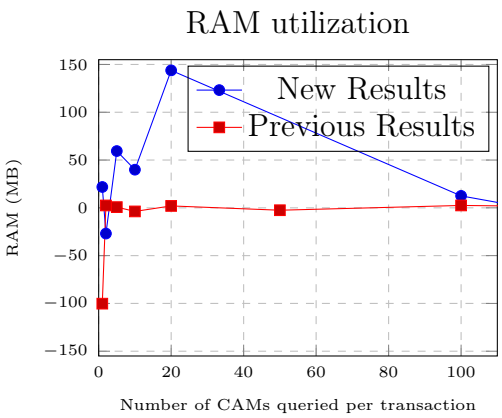
Query Results



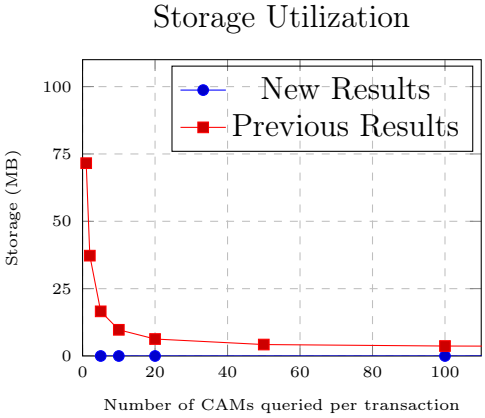
(a) Query Time



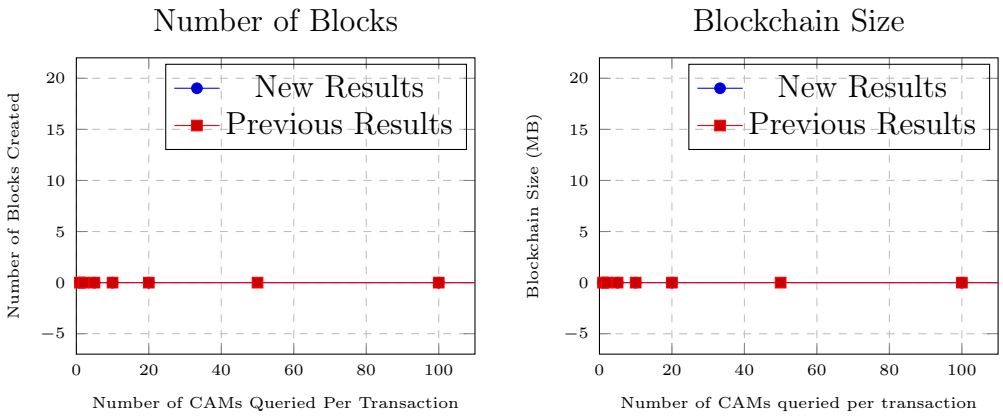
(b) Average CPU Utilization



(c) RAM Utilization



(d) Secondary Memory Utilization



(e) Number of Blocks

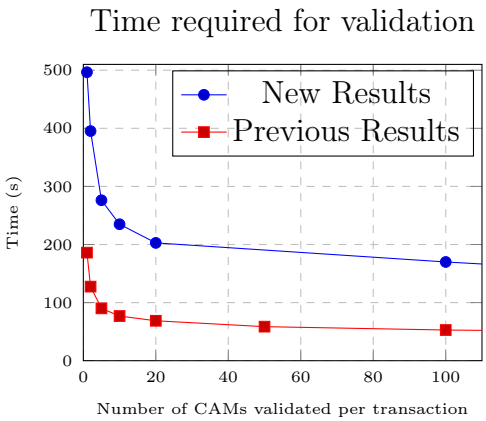
(f) Blockchain Size

Figure 6.14: Querying of all CAMs results

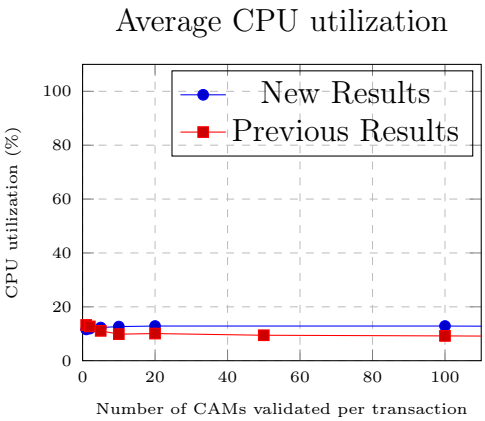
Analysis

Time Results: The time required drops very fast when increasing the number of CAMs quried by a transaction. This is due to the decrease in the number of transactions and corresponding overheads they introduce. The overhead of a transaction is not negligible because it involves cryptographic operations.
Similar results is observed in the previous thesis.
CPU Utilization: The CPU behavior remains constant with an increase in number of CAMs per transaction. The expected behavior would be a decreasing trend since there are less transactions to process. The reasoning can be that the CPU is at its maximum limit in all cases and therefore only reduces the time required.
In the previous thesis a decreasing trend is observed.
RAM Utilization: As the querying operation shouldn't affect the RAM we see that RAM behavior takes small values depending on the operations that were being executed at the instant of measurement.
Similar results are observed in the previous thesis.
Storage Utilization: There is no storage utilization for querying operations.
In the previous thesis there is extra storage for querying operation. This, most likely was a bug in the previous version of fabric.
Number and Size of Blocks: No new blocks were created during the querying operation.
Same result is observed in the previous thesis.

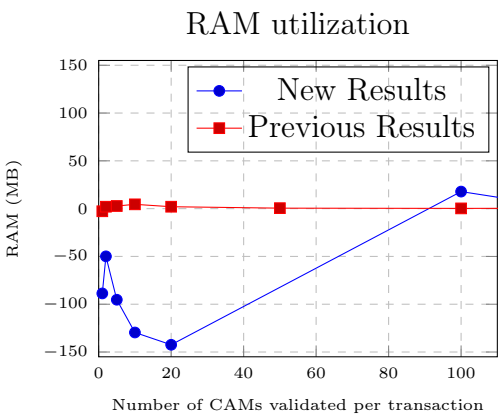
Validation Results



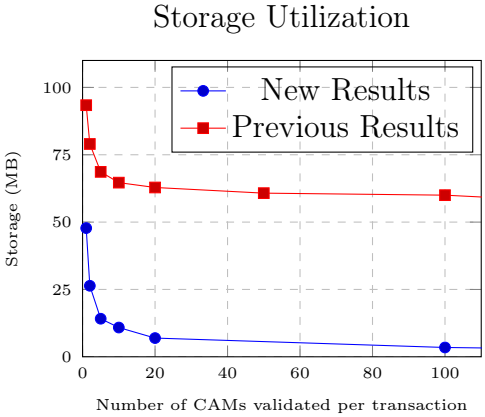
(a) Validation Time



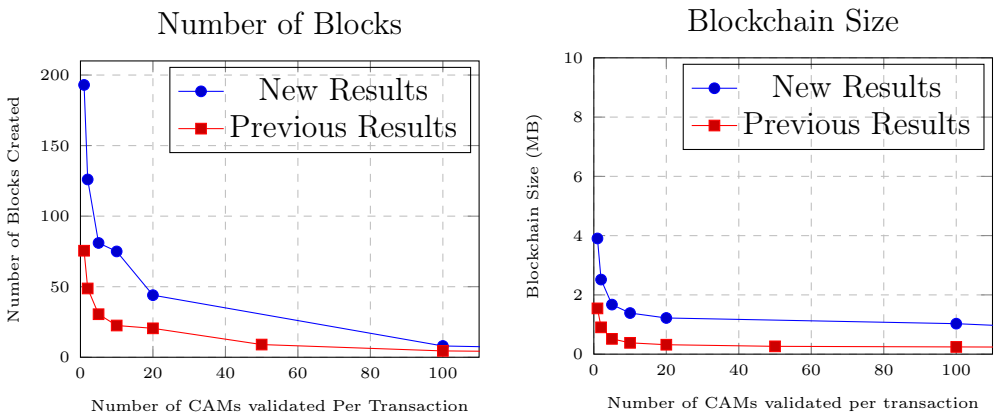
(b) Average CPU Utilization



(c) RAM Utilization



(d) Secondary Memory Utilization



(e) Number of Blocks (f) Blockchain Size

Figure 6.15: Validation of all CAMs results

Analysis

Time Results: The time required drops very fast when increasing the number of CAMs quried by a transaction. This is due to the decrease in the number of transactions and corresponding overheads they introduce. The overhead of a transaction is not negligible because it involves cryptographic operations.
Similar behavior is observed in the previous thesis. Although in their results the time is about half of the time here. Again this is due to the number of CAMs being two times higher in this thesis, and validation deleting reception info.
CPU Utilization: The CPU behavior remains constant with an increase in number of CAMs per transaction. The expected behavior would be a decreasing trend since there are less transactions to process. The reasoning can be that the CPU is at its maximum limit in all cases and therefore only reduces the time required.
Similar result is observed in the previous thesis.
RAM Utilization: As the validation operation shouldn't affect the RAM we see that RAM behavior takes small values depending on the operations that were being executed at the instant of measurement.
Similar behavior is observed in previous thesis.
Storage Utilization: Secondary memory requirements decrease with the increase of the number of CAMs per transaction. This is because each transaction introduces an overhead in the blockchain: the signatures, read write set, etc. Thus having less transaction leads to lower blockchain size. The state database size remains the same regardless of the transaction size.

In the previous thsis the same behavior is observed. In this case the actual amount of storage is larger which makes sense since the validation in the other thesis doesn't delete the reception information from the state database.
Number and Size of Blocks: Each block contains 10 transactions. Thus by increasing the number of CAMs we decrease the number of total transaction and the number of new blocks created.
In the previous thesis the same behavior is observed. As expected the number of blocks is higher in this thesis since the total number of CAMs is about twice higher. The same goes for the blockchain size.

6.4.8 Storage, Querying, and Validation performance with Concurrent Processes

As explained before, the possibility of parallelism is one of the main advantages of hyperledger fabric with respect to other blockchain platforms. In this experiment we will analyze the effect of parallelism by running the three operations with different number of concurrent processes.

Objective of the experiment

To analyze performance metrics of storage with processing, querying and validation of CAM reports in dependence of the number of concurrent processes for each operation.

Experiment Implementation

Starting multiple hyperledger networks with six stations and six peers. Each network will use a different number of processes for all operations. The operations executed will be the following:

- Uploading the trace with processing
- Query all CAMs stored in the network
- Validate all CAMs stored in the network

For each operation we measure the following metrics:

- Average CPU consumption
- RAM usage
- HDD usage

- Time required for the operation
- Number of new blocks created for the operation
- Total size of the new blocks

The experiment is repeated five times.

Experiment Configuration

Number of Organizations	1
Number of Peers per Organization	6
Total number of Peers	6
CPU Peers	10%
CPU Orderers	100%
Number of Orderers	1
Ordering service	Solo
State Database	CouchDB
Vehicle simulator traces	Trace 4 (1 MB)
Type of storage	With Processing
CAMs stored, validated, or queried per transaction	20

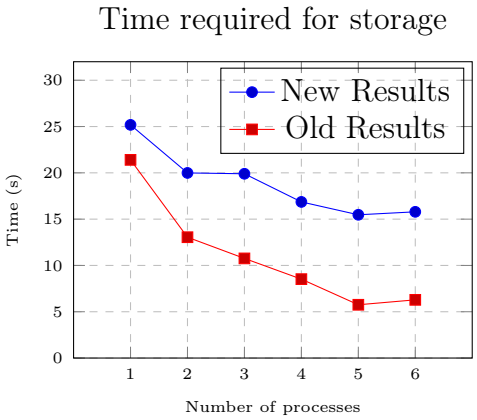
Table 6.17: Network Configuration

BatchTimeout	2
MaxMessageCount	10
AbsoluteMaxBytes	99 MB
PrefferedMaxBytes	512 kB

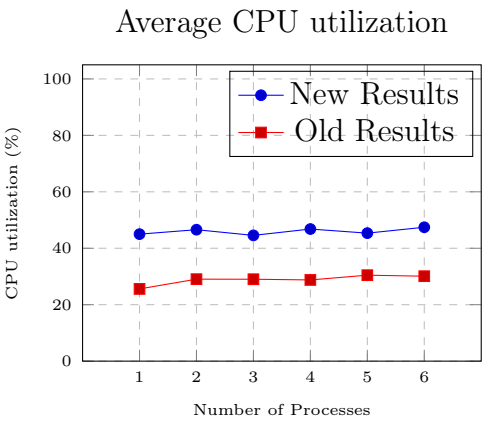
Table 6.18: Blocks Configuration

Experiment results

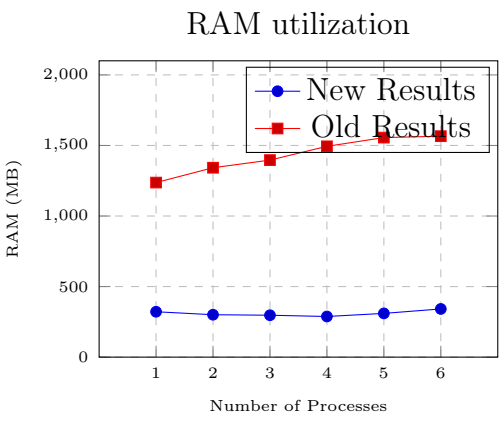
Storage Results



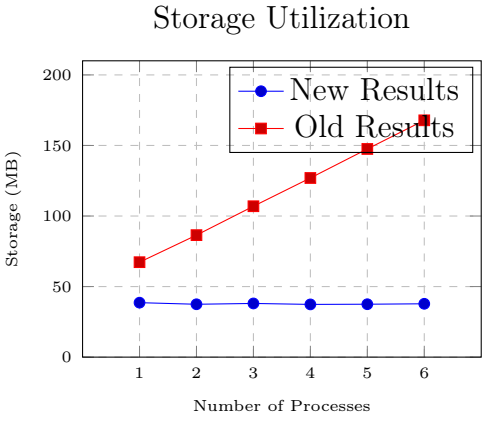
(a) Upload Time



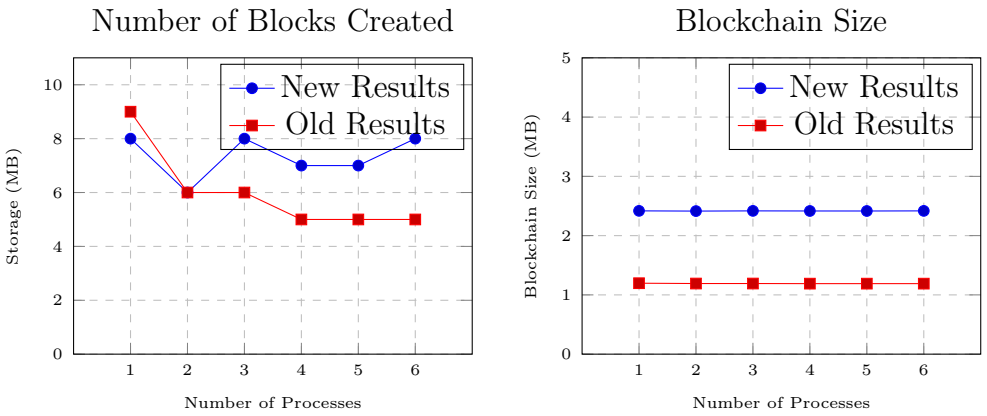
(b) Average CPU Utilization



(c) RAM Utilization



(d) Secondary Memory Utilization



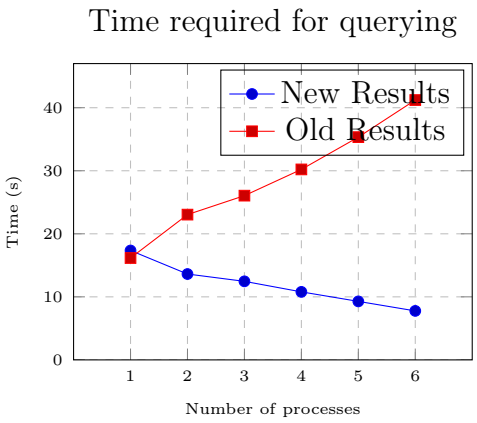
(e) Number Of Blocks (f) Blockchain Size Increase

Figure 6.16: Storage of all CAM Reports With Processing Results

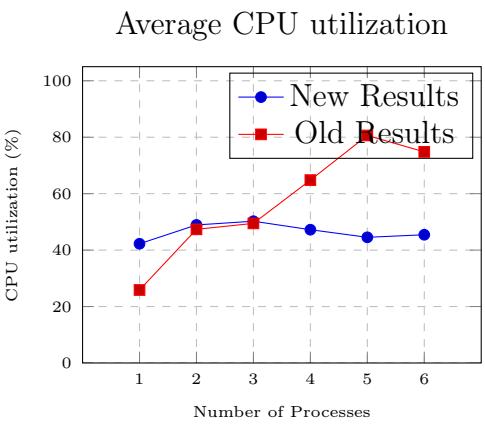
Analysis and Comparison

Time Results: As expected by increasing the number of parallel processes the time drops linearly. This proves that only the endorsing peers(a single peer in out case) need to execute the transaction.
Similar results are observed in previous thesis.
CPU Utilization: CPU Utilization remains constant with the increase of the number of processes. This is because irregardless of the amount of processes the number of transactions to execute remains the same.
Similar results are observed in the previous thesis.
RAM Utilization: RAM utilization remains more or less constant with the increase in number of processes. The small fluctuations are probably due to operations being executed at the time of the measurement.
In the previous we see that there is a small linear increase in RAM usage but generally it doesn't change a lot. The constant behavior is expected since the RAM usage is not measured during the operation but after, and the network is the same in all tests.
Storage Utilization: Storage needed remains the same since the number of CAMs is the same.
In the previous thesis we have a linear increase in storage. This results is strange since we are only increasing the number of processes, not the amount of data stored.
Number and Size of Blocks: Number and size of the created blocks remains the same with respect to the number of processes.

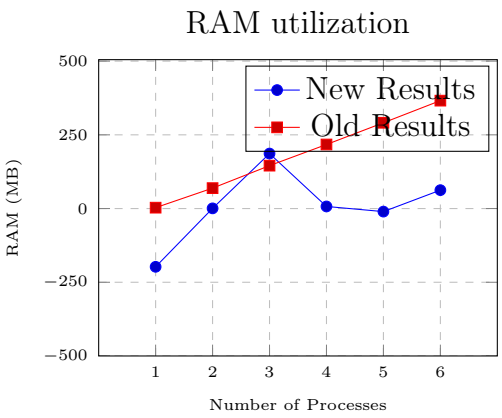
Query Results



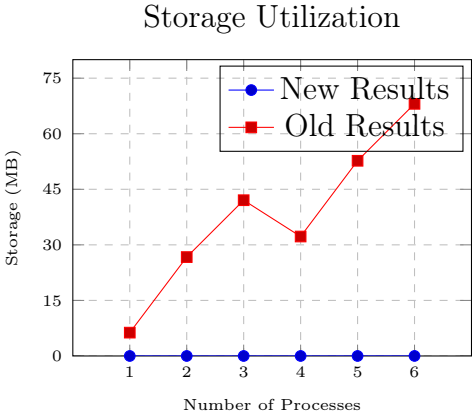
(a) Query Time



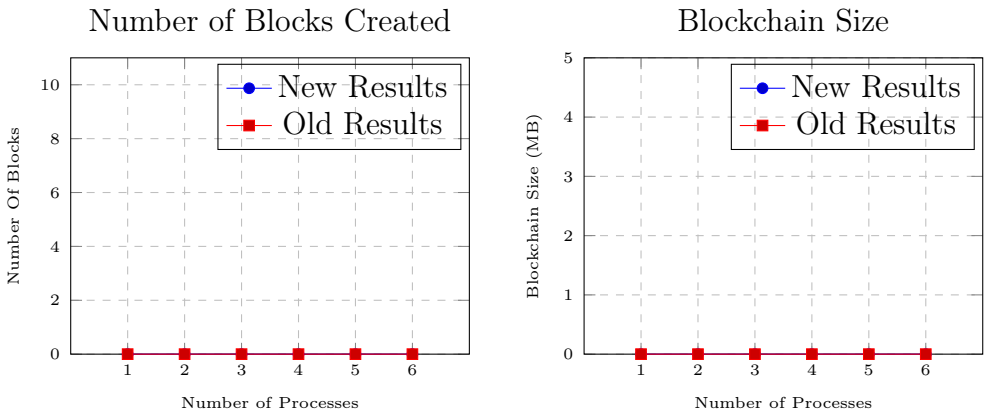
(b) Average CPU Utilization



(c) RAM Utilization



(d) Secondary Memory Utilization



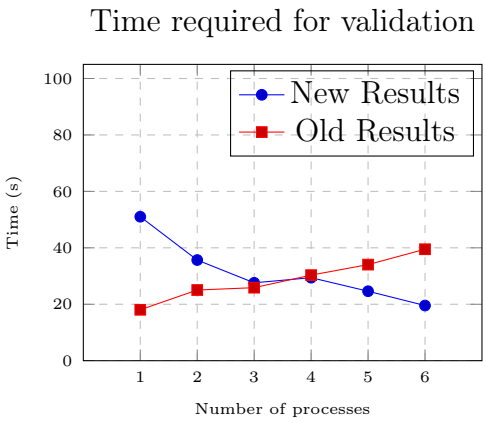
(e) Number of Blocks (f) Blockchain Size

Figure 6.17: Querying of All CAMs Results

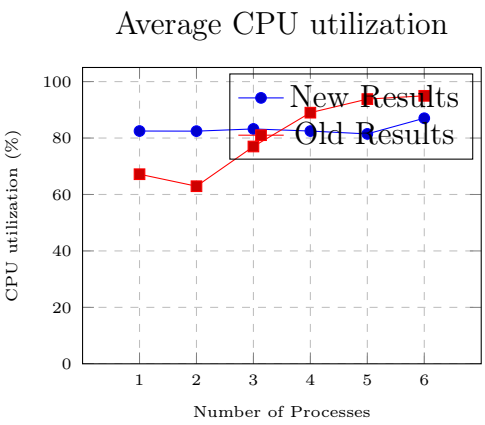
Analysis and Comparison

Time Results: As expected by increasing the number of parallel processes the querying time drops linearly.
In the previous thesis we have a strange result that the time increases with the number of processes. As the author states this is most likely due to the limitations in the number of cores of the testing machine. **CPU Utilization**: CPU Utilization has small fluctuations but generally hovers around the same value. This is because irregardless of the amount of processes the number of transactions to execute remains the same.
In the previous thesis we have that the CPU increases with the number of processes. This behavior could also be explained by the number of cores available in the testing environment. **RAM Utilization**: The querying operation shouldn't affect the RAM. We see that RAM values seem random because they only depend on the action that's performed at the time of measurement.
In the previous thesis we have a linear increase in the RAM usage with the number of processes. This is a strange behavior since the RAM is measured as the difference before and after the action is executed, therefore the number of processes shouldn't affect it. And in general query operations shouldn't introduce any long-term RAM usage. **Storage Utilization**: Querying operation doesn't use any storage.
Number and Size of Blocks: Querying operation doesn't create any new blocks.

Validation Results

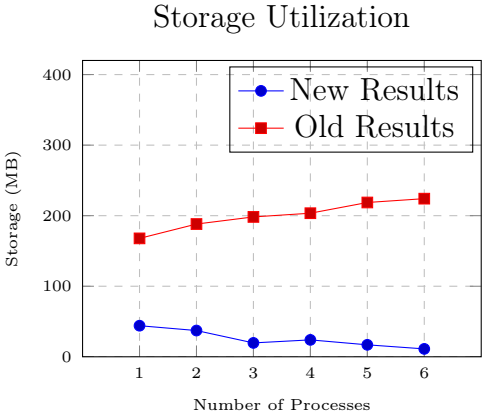


(a) Validation Time

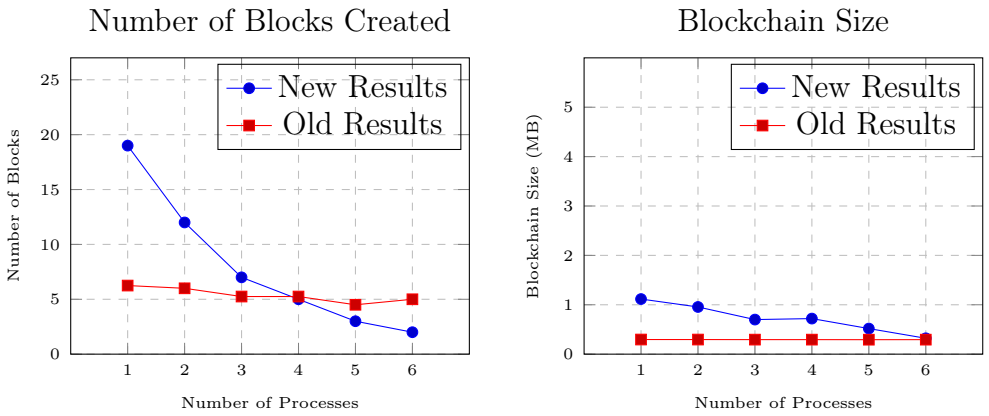


(b) Average CPU Utilization

(c) RAM Utilization



(d) Secondary Memory Utilization



(e) Number of Blocks (f) Blockchain Size

Figure 6.18: Validation of All CAMs Results

Analysis and Comparison

Time Results: As expected by increasing the number of parallel processes the validation time drops linearly. CouchDB requires more time then LevelDB since it's running in it's own container and we need to use the REST API.
CPU Utilization: CPU Utilization has small fluctuations but generally hovers around the same value. This is because irregardless of the amount of processes the number of transactions to execute remains the same.
RAM Utilization: The validation operation shouldn't affect the RAM long term. We see that RAM values seem random because they only depend on the action that's performed at the time of measurement.
Storage Utilization: For LevelDB the storage is independent of the number of processes. For CouchDB the storage requirements drop when increasing the number of processes. This behaviour seems strange but can be understood by looking at the number of blocks created.
Number and Size of Blocks: The number of blocks for LevelDB remains constant but for CouchDB decreases linearly. This can be understood if we remember that a block can hold up to 10 transactions and the max interval between block generation is 2 seconds. Since the validation takes longer with CouchDB, the number of transaction that can be collected in each block is smaller. By increasing the number of processes we allow a larger number of transaction in each block for CouchDB. This result was verified by increasing the batch timeout and seeing the same behavior between the two state databases.
The results from the other thesis are available in [Michele's Thesis Reference]

fig 7.19. The differences for the Time, CPU and RAM can be explained in the same way as for the querying operation. The difference in storage utilization implies that probably the Batch Timeout parameter was set to longer than 2 seconds.

6.5 Throughput and Latency Experiments

In the previous experiments it was not possible to measure throughput and latency. The reason is that a CLI client was used which only gets the notification when the data is processed by the endorsing peer. It does not know when the CAMs are stored in the blockchain network. The previous experiments were performed in order to compare performance with the [Michele's Reference] which had a slightly different architecture and used an older version of fabric.

6.5.1 Fabric Node Client

Fabric supports a Node.js SDK for developing clients. The SDK provides a multitude of options with respect to the bash CLI client.
The most important thing is that after invoking a transaction from the Node client, a notification is sent from the peer to the client when the transaction has been **stored in the blockchain**(therefore after the consensus protocol), not when the peer executes the invoked chaincode. This allows us to measure latency and throughput.

6.5.2 Hyperledger Caliper

Hyperledger caliper is a tool that allows to benchmark the performance and resource consumption of the blockchain. It takes as an input a file describing the hyperledger network configuration and the file describing how the benchmark should be run. The metrics measured are taken from [Metrics Whitepaper Reference]; The parameters measured and their definitions are the following:

Read Latency

$$Read\ Latency = Time\ when\ response\ is\ received - Submit\ time$$

Read Latency is the time between when the read request is submitted and the time when the response is received.

Read Throughput

$$Read\ Throughput = Total\ Read\ Operations/Total\ Time\ in\ Seconds$$

Read Throughput is the number of read operations performed in a certain time interval. It is measured in units of Read Per Seconds(RPS). This can be an informative metric, but is not representative of blockchain performance since most blockchains facilitate secondary data structures to allow for efficient read and query operations.

Transaction Latency

$$Transaction\ Latency = (Confirmation\ time\ @\ Network\ Threshold)\breve{}Submit\ Time$$

Transaction Latency is the amount of time from when a transaction is submitted until it's results are widely available in the network. This includes the propagation time and the settling time due to consensus mechanism. Network Threshold represent the percentage of the nodes in the network at which the transaction results are available. In system that use the POW scheme, the desired threshold is 90%. Whereas in non-probabilistic systems, such as Hyperledger Fabric, the 100% metric is the only meaningful one.

Transaction Throughput

$$Transaction\ Throughput = Total\ committed\ transactions/total\ time\ in\ seconds\ @Entire\ Network$$

Transaction Throughput is the rate at which the valid transaction are committed by the system under testing in a defined time period. This is measure not a single node, but at the entire network level. Invalid transactions do not contribute to the results. The rate is expressed as the number of committed transactions per second(TPS).

Report & CAM Throughput

In the context of the thesis, each transaction performs an operation on multiple CAMs. Therefore we define two derived metric:

$$Report\ Throughput = Transaction\ Throughput * \#Reportsuploadedbyeachtransaction$$

$$CAM\ Throughput = Transaction\ Throughput * \#CAMsvalidatedorqueriedbyeachtransaction$$

6.5.3 State Database Experiments

As said before, CouchDB offers rich querying capabilities which allow multitude use cases in the context of the thesis. The downside is the performance of the blockchain decreases with respect to using LevelDB as the state database. The reason for the performance decrease is that CouchDB runs in a separate container and any CRUD operation must be sent using the CouchDB API over an HTTPS connection.
This batch of experiments aims to determine the amount of performance degradation by using CouchDB as a state database.

Experiments Configuration

Since we are interested in benchmarking the blockchain network, traces are not specified. We simply increase the input rate until we obtain the maximum output rate.

Number of Organizations	2
Number of Peers per Organization	2
Total number of Peers	4
Number of Orderers	1
Ordering service	Solo
State Database	LevelDB and CouchDB
#Reports Stored per Transaction	20
#Reports Queried per Transaction	20
#Reports Validated per Transaction	20

Table 6.19: Network Configuration

BatchTimeout	2
MaxMessageCount	10
AbsoluteMaxBytes	99 MB
PrefferedMaxBytes	512 kB

Table 6.20: Blocks Configuration

Results

Metric	**LevelDB**	**CouchDB**
Max Latency	3.20 s	13.5 s
Min Latency	0.87 s	0.73 s
Avg Latency	1.99 s	16.37 s
Throughput	17.5 tps	5.1 TPS
Report Throughput	350 rps	105 RPS

Table 6.21: Report Uploading Results

Storage Results: Concerning the uploading of CAMs, the LevelDB provides about 4 times better performance with respect to CouchDB. The biggest different is in the latency where CouchDB peforms 8 times worse.

Metric	**LevelDB**	**CouchDB**
Max Latency	2.92 s	32.38 s
Min Latency	0.81 s	4.68 s
Avg Latency	2.04 s	23.48 s
Throughput	12.4 TPS	1.2 TPS
Cam Throughput	250 CPS	25 CPS

Table 6.22: Cam Validation Results

Validation Results: The biggest difference in performance is seen in validation. LevelDB provides about 10 times better performance across all metrics.

Metric	LevelDB	CouchDB
Max Latency	3.74 s	7.96 s
Min Latency	0.68 s	3.30 s
Avg Latency	2.32 s	6.47 s
Throughput	31.4 TPS	14 TPS
Cam Throughput	625 CPS	280 CPS

Table 6.23: Cam Query Results

Querying Results: As expected the querying operation is the fastest since there is no consensus protocol. Still LevelDB provides 2 times better performance then CouchDB.

Results Analysis

Even though CouchDB provides a very flexible and rich querying system, the performance degradation is pretty impactive. The worst performance is in the Validation part, most likely because it requires retrieving keys based on a partial composite key.
The performance could be affected due to running on a virtual machine and most likely having the hard drive access as a bottleneck.
Still CouchDB provide a flexibility that makes the topic of this thesis useful in a large number of scenarios.
Since fabric provides a pluggable world state database, the best option would be to create a database that offers the flexibility of CouchDB but can be run in the same container as the peer.

Chapter 7

Conclusion

Bibliography

[1] T. ETSI, "Intelligent transport systems (its); vehicular communications; basic set of applications; part 2: Specification of cooperative awareness basic service," *Draft ETSI TS*, vol. 20, pp. 448–451, 2011.

[2] T. ETSI, "101 539-1 v1. 1.1 (2013-08) intelligent transport systems (its)," *V2X Applications*.

[3] Hyperledger Contributors, "Introduction – hyperledger-fabricdocs master documentation." https://hyperledger-fabric.readthedocs.io/en/release-1.4/whatis.html, 2019. [Online; accessed 25-March-2019].

[4] Hyperledger Contributors, "Ledger – hyperledger-fabricdocs master documentation." https://hyperledger-fabric.readthedocs.io/en/release-1.4/ledger/ledger.html, 2019. [Online; accessed 25-March-2019].

[5] Hyperledger Contributors, "peer – hyperledger-fabricdocs master documentation." https://hyperledger-fabric.readthedocs.io/en/release-1.4/commands/peercommand.html, 2019. [Online; accessed 26-March-2019].

[6] Hyperledger Contributors, "Commands reference – hyperledger-fabricdocs master documentation." https://hyperledger-fabric.readthedocs.io/en/release-1.4/command_ref.html, 2019. [Online; accessed 26-March-2019].

[7] Hyperledger Contributors, "peer channel – hyperledger-fabricdocs master documentation." https://hyperledger-fabric.readthedocs.io/en/release-1.4/commands/peerchannel.html, 2019. [Online; accessed 26-March-2019].

[8] Hyperledger Contributors, "peer chaincode – hyperledger-fabricdocs master documentation." https://hyperledger-fabric.readthedocs.

io/en/release-1.4/commands/peerchaincode.html, 2019. [Online; accessed 26-March-2019].

[9] Docker Contributors, "Overview of docker compose | docker documentation." https://docs.docker.com/compose/overview/, 2019. [Online; accessed 26-March-2019].

[10] The Go Programming Language Contributors, "The go programming language." https://golang.org/, 2019. [Online; accessed 01-April-2019].

[11] GNU Contributors, "Bash - gnu project - free software foundation." https://www.gnu.org/software/bash/, 2019. [Online; accessed 01-April-2019].

[12] The Go Programming Language Contributors, "template - the go programming language." https://golang.org/pkg/text/template/, 2019. [Online; accessed 01-April-2019].