Distributed Ledger Technologies for Managing Heterogeneous Computing and Sensing Systems at the Edge

Smart Systems
Turku Intelligent Embbedded and Robotic Systems (TIERS) Lab
Master's Degree Programme in Information and Communication Technology
Department of Computing, Faculty of Technology
Master of Science in Technology Thesis

Author:
Daniel Andrés Montero Hernández

Supervisors:
MSc. (Tech) Jorge Peña Queralta
Assoc. Prof. Tomi Westerlund

July 2022

UNIVERSITY OF TURKU
Department of Future Technologies

DANIEL ANDRES MONTERO HERNANDEZ: Distributed Ledger Technologies for Managing Heterogenous Computing and Sensing Systems at the Edge

Master of Science Thesis, 61 p.
Turku Intelligent Embedded and Robotic Systems (TIERS) Lab
July 2022

The increased popularity of Internet of Things (IoT) devices, ranging from simple sensors to powerful embedded computers, has created the need for solutions capable of processing and storing information near those assets. Edge Computing (EC) has become a staple architecture when designing solutions for IoT, as it optimizes the workload and capacity of systems dependent of the Cloud, by placing the required computing power near to where the information is being produced and consumed. An issue with these solutions, is that reaching consensus regarding the state of the network becomes more challenging as they scale in size. Distributed Ledger Technology (DLT) can be described as a network of distributed databases that incorporate cryptography and algorithms to reach consensus among the participants. DLT has gained traction over the past years, particularly due to the popularity of Blockchain, the most well-known type of DLT implementation. In addition to the capability of reaching consensus, another key concept that brings EC and DLT together, is the reliability and trust that the latter offers through transparent and traceable transactions. In this thesis, we present the design and development of a proof-of-concept system that uses DLT Smart Contracts (SC) as the core for efficiently selecting Edge Nodes for offloading services. We present the experiments conducted to demonstrate the efficacy of the system and our conclusions regarding the usage of Hyperledger Fabric for managing systems at the edge.

Keywords: DLT, Edge Computing, IoT, Hyperledger Fabric, Smart Contracts, Sensing Systems

Contents

List of Figures

List of Tables

List Of Acronyms

DLT	Distributed Ledger Technologies
EC	Edge Computing
HFN	Hyperledger Fabric Network
IoT	Internet of Things
SC	Smart Contract

1 Introduction

As a means of bringing together some of the latest key technical developments, we at TIERS have set out to establish a collaborative and long-term autonomous framework for distributed robotic systems under the RoboMesh Project. The objective of this document is to establish a design and part of the development of this framework. For the purpose of testing the efficacy of such a system, we will be combining and integrating Edge Computing and Distributed Ledger Technologies as the backbone for an efficient and seamless computational offloading service.

Edge Computing (EC) is a paradigm of distributed computing and a well-established topological concept, and its primary objective is to optimize workload and capability of systems, by placing information processing capabilities closer to where things and people produce and consume said information [1].

The crescent popularization of EC and the Internet of Things (IoT) has been a topic of research in the past years [2]. This exponential growth of IoT has been leveraged by the continued interconnection of devices, computing resources and robots. An IoT network is made up of devices with network capabilities, which monitor data or bring intelligence to multiple domains [3]. The connection of IoT devices which we will explore in this thesis, comprised mostly of data collecting sensors, can be defined as a Sensing System.

Distributed Ledger Technology (DLT) has gained traction over the past years as well, particularly due to the popularity of Blockchain, the most well-known type of DLT implementation. The technology itself is no longer a novelty. The whitepaper that described the

cryptography peer-to-peer monetary exchange was published over a decade ago, and the main characteristics have remained unchanged; but it has slowly made its way into applications and situations where financial or value transactions are not the only objectives [4]. DLT implementations have high potential in applications that require the participation of multiple actors, which may have different objectives in distinct points of time, in which data transactions need to be recorded, shared, and synchronized across the participants [5].

1.1 Significance and Motivation

There has been a massive introduction of devices that can communicate with each other, sometimes without the direct intervention of people. It is expected that by 2025 there would be around 150 billion active device connections. The accelerating growth of IoT devices in the market has resulted in macroeconomic developments due to the increased investments in the sector [2], an important incentive for the research and development of TIERS in this area. This project is also motivated by the objective of the RoboMesh project that the TIERS research group participates in: implementing a trust-emphasized framework for the integration of distributed computational offloading orchestration with multi-modal sensor fusion algorithms and collaborative decision-making. As part of the aforementioned system, we have designed and developed an extensible DLT-based system that allows the selection of edge resources for offloading sensor related tasks to of distinct characteristics in a network.

While distributed resource management plays a crucial role in the efficacy of the system, and the implementation of algorithms for sensor analysis in the capabilities of the system, the focus of this thesis is the design and development of Hyperledger Fabric Smart Contracts to select the best Edge resource available to perform a specific task.

In summary, our main objective is the design and development of a system that will collect resource data from edge nodes, measure network latency between the nodes and

other computers acting as sensors, and handle the data storage and edge node selection for offloading through Smart Contracts. An implementation of task offloading orchestration will be implemented for testing the capabilities of the system, and we will present a graphical interface that will support the user of the system. The code will be made publicly available to our GitHub repository[1]. This thesis will conclude with our observations regarding the viability of the usage of Hyperledger Fabric Smart Contracts for orchestrating task offloading in the Edge.

1.2 Related works

Both Edge Computing and Distributed Ledger Technologies are currently relevant and novel topics in both the industry and academia. The management of Edge systems is a challenging topic and research regarding the use of DLT as an assisting technology is growing in popularity. Security, resiliency and permissioned multi-actor participation; characteristics of DLT [6], can help overcome the challenge of consensus in the collaborative networks. The system must be able to reach an agreement regarding the state of the network and the availability of the resources. Consensus is one of the biggest challenges that we are faced with when planning and designing systems that can be large in scale, and must also communicate with potentially heterogeneous computing resources while attempting to bring results in a low-latency basis. For the remainder of this section, we explore and review the different architecture proposals and prototypes of DLT frameworks for Edge Computing and Sensing systems.

1.2.1 DLT and Edge Computing

As the advancement of computing technologies allows for extension of systems, Edge Computing and other similar implementations, such as Fog Computing, become more

[1] https://github.com/TIERS/fabric-edge-node-selector

prevalent as ways to expand the use of the Cloud. Similar to the Cloud, Edge Computing has the objective of assisting the user by providing computation power, data storage and application services in a manner that maintains lower latency and improves the perceived quality of service. While there exist many benefits to the technology, the literature points out the challenges that such implementations have regarding the security and privacy of the information that is processed, due to the constant migration of services across edge nodes [7].

Literature that explores the application of DLT-backed systems is becoming more prevalent as the technology becomes more frequent due to the numerous applications that make use of it, ranging from smart grids to localized communication of IoT devices. One of the key concepts that is used to tie blockchain technology to the management of Edge Nodes is the guarantee of data integrity and validity, however, it is also mentioned that the cryptographic workload required can be computer-intensive depending on the consensus algorithm that is used [8].

Salimi *et al.* implemented a proof-of-concept DLT framework and the authors conclude that the integration of robotics applications with Hyperledger Fabric can have minimal impact on the utilization of computational resources [9].

Peña *et al.* discussed an architecture that enhances the autonomous operations of connected robots and vehicles with blockchain as the enabling technology to provide services and manage resources in a transparent and secure way [10].

1.3 Contributions

The contribution of this document is a proof-of-concept platform for offloading computational tasks to Edge resources using Distributed Ledger Technologies as the core, and can be summarized as:

- Hyperledger Fabric Smart Contracts that can store the data, helps determine the tar-

gets for latency measurement and selects an edge node based on the task parameters

- a Go application that interfaces with Hyperledger Fabric
- a Go application that can measure the host's current resource availability, and measures the host's latency to specific targets
- configuration file for Postman-like REST-Clients to interact with the system

1.4 Structure

The aim of this document is to detail the design and implementation of a Hyperledger Fabric system using Smart Contracts, the implementation of an application to interface with the Hyperledger Fabric Network via REST API, and an application to collect the resource availability in homogeneous edge computing resources, measure network latency and execute Docker images.

- Chapter 2 introduces the distributed computation concepts relevant to the work and details the software, hardware and network components.
- In chapter 3, we describe in detail the proposed Smart Contracts, algorithms, communication methods and design decisions for the system.
- In chapter 4, the implementation of the system is described along the focus of the tests and the data obtained.
- In chapter 5, we report the results obtained from the experiments.
- Finally, chapter 6 explains the conclusions obtained from the implementation of this work and outlines future work directions.

2 Background

The main part of the project is the development of the Edge Management and Offloading System by exploiting the characteristics of different distributed computing paradigms, hence the technologies will be explained in detail in this section. Furthermore, the software and hardware used in the deployment of the system will be presented and distinguished.

2.1 Distributed Computing Systems

Through the years, multiple definitions for Distributed Computing have appeared in literature and rarely do these definitions agree with one another. The differences between these definitions usually lies in the scope of the distributed computing system that is being described by the authors. For the purpose of this work, we have settled on a combination of the definitions used by van Steen [11] and Coulouris [12]:

"A distributed system is a collection of autonomous networked computing elements, that appears to its users as a single coherent system, that communicate and coordinate their actions by passing messages."

From this definition we can extract the three main characteristics of such systems:

- Existence of multiple entities, or nodes, with their own local memory that behave independently. Nodes can be described as independent hardware devices, or as independent software processes. And unlike in other computer paradigms, there is

no shared memory pool between the nodes used to orchestrate their efforts [13], instead:

- Communication between the nodes is handled through message passing. The messaging system may vary between implementations, with some of the common ones being the use of protocols present in operating systems, such as TCP/IP, or by using Middleware Message Brokers [14].

- End-clients or users interact with the system as if it were a single entity. Since achieving a single-system view for a system made up of multiple entities, distributed systems are often just made to appear to be a single coherent system to the user by behaving according to the expectations [11].

2.1.1 Cloud Computing

The Cloud, as it is often referred to, is the delivery of a plethora of different services through the network access. These services offered are typically served by distributing functionalities through a collection of multiple devices, sometimes in multiple locations. Cloud computing relies on sharing networked resources to cohesively, seamlessly and conveniently present functionalities to users with access to the network [15].

Cloud Computing is a popular and relevant topic nowadays due to the proliferation of the IT service delivery business model. As of today, the reliance on the Cloud has grown incredibly, due to many of the services we consume over the internet being deployed on these cloud platforms. The main reason cited for this phenomena is the price advantage that it presents to individuals and organizations, as the usual pay-per-use model is cheaper than investing in private infrastructure, considering not only the hardware, but also the set-up and its maintenance [16], [17].

2.1.2 The Edge & Edge Computing

Edge Computing is a distributed computing topology and design paradigm, and is commonly used to design the architecture of applications. This particular network topology supports the introduction of computing applications and services as close as possible to the source of the data that will be computed. This source of data depends on the particular use case of the system, but can generally be referred to as the end-user of a system; and accordingly, this section of the network where the data sources and end-users are located is referred to as The Edge. [1].

Typically, Edge Computing serves as an additional layer between end-devices and Cloud services. The main reason why the architecture of a service will include Edge capabilities is to diminish the latency that exists between the two aforementioned layers. However, the benefits of this paradigm depend heavily on the application. The relevant highlights of these benefits are:

- Latency: Since some applications rely on the speed of the responses, processing near the end-device can provide faster results than Cloud services.
- Privacy and Security: Some designs restrict the flow of data between end users and edge node, minimizing the transmission of sensitive data. Furthermore, the connections between the Edge resources and the Cloud can be controlled and monitored.
- Data Bandwidth Efficiency: By routing the data from sensors to an Edge resource where it can be processed, the amount of information that needs to be sent to the Cloud services can be dramatically reduced, sending only the relevant data or results.

Some other benefits include limited autonomy, disconnected operation and local interactivity [18], [19].

2.1.3 Distributed Ledger Technologies

Distributed Ledger Technology (DLT), as defined by Wright and De Filippi [20], is a "distributed, shared, encrypted databases that serve as an irreversible and incorruptible repository of information". From a technical standpoint, the implementation of a DLT system can be described as replicated databases that arrange chronological bundles of transactions in a way that the integrity of each proposed transaction can be checked. The systems give the users the ability to store and access information in the shared database using cryptographic validation system. Unlike standard centralized accounting ledgers, DLTs are maintained by a distributed network of participants [21].

The way that DLT systems achieve the previous definition is by implementing a technology stack that is mainly comprised of four components: distributed ledger, consensus algorithms, cryptography and smart contracts [22].

2.1.3.1 Distributed Ledgers & Blockchain

Distributed ledgers are consensually shared and synchronized databases. In contrast to a centralized ledger, distributed ledgers are less prone to cyber-attacks and fraud, since they don't suffer from having a single point of failure [23]. In a Blockchain the data stored in the ledger is updated in real-time via the consensus of the different nodes present in the network [24]. Generally, as information is introduced by the users, the transactions are collected in groups, which are referred to as blocks. Once the system determines that the block has enough transactions, the block is closed and linked to the previous block, forming a chain of data, unlike a regular database table. The end result of this configuration is an irreversible timeline of data transactions, which is why it is generally asserted that once data is part of the chain, it cannot be removed or edited. [24], [25].

However, some DLT implementations differ from Blockchain. The ledger can be adapted to function as a "Directed Acyclic Graph". In this context, acyclic means that not all network peers are required to be in sync at all times and are built around the

assumption that over time, using built-in communication methods, all information will be shared and validated by other nodes [26]. Regardless of the data synchronization method, we are interested in exploring the permission configurations of these distributed ledgers, those that are permissionless and those that are permissioned. In a permissionless implementation, anyone is free to join the network and participate in the activities, as is the case for Bitcoin; whereas in permissioned ones the operator verifies and selects the entry of participants. Other implementations are also considered private, but for effects of this document, these do not count as fully distributed ledgers [27].

These different permission implementations exist due to the growth of enterprise use cases, some of the performance requirements and underlying characteristics need another type of network implementation. Permissioned DLT platforms are known for the capability of delivering enterprise-grade functionalities. A permissionless implementation allows for anonymous clients to participate, and consensus is typically reached via Proof of Work or Proof of Stake. Whereas the permissioned counterparts, as the Hyperledger Fabric website [28] explains, "operate a blockchain amongst a set of known, identified and often vetted participants operating under a governance model that yields a certain degree of trust", in other words, allows for secure interactions between entities that have a common goal, but do not necessarily fully trust each other. The identification of participants permits the use of consensus protocols that are less resource intensive.

2.1.3.2 Consensus Algorithms

On logical grounds, distributed systems can't have a central authority present to validate and verify each transaction, yet every proposed transaction in a DLT must be secured and verified. Consensus mechanisms are also the mechanisms that generate new blocks in a blockchain [29]. Said verification and validation must be put in place to tolerate Byzantine Faults, a known condition and challenge of distributed computing systems. Byzantine Fault refers to components being prone to failure, and the existance of imperfect informa-

tion regarding whether a component has failed [30], [31].

Consensus algorithms commonly present in DLT implementations are:

- Proof of Work (PoW):

  Requires members of the network to solve an arbitrary mathematical equation that generates hashes, long strings of numbers. The hash serves as evidence of the computational power expenditure, and if changes are made to the transaction data used, the hash will result in an unrecognizable and non-replicable hash. The member of the network to first solve the problem creates the next block and receives a reward. [31], [32].

- Proof of Stake (PoS):

  The members of the network, instead of investing in hardware to quickly solve equations, invest in the digital currency of the blockchain and use their investment as collateral, or stake. Nodes with higher stakes are selected to validate new blocks of data. Consensus is encouraged by removing the collateral from the nodes that verify bad or fraudulent data, and are awarded in proportion to the stake they took if the majority of validators agrees on the result [31], [33].

- Practical Byzantine Fault Tolerant (PBFT):

  PBFT [34] is an algorithm that uses a system based on roles, communication and consistency. In a network, nodes with different roles are expected to exist and they verify cryptographic information generated by each other to vote on a consistent answer. Other than the communication between the client responsible for the proposal of the transaction, and the mechanism that maintains all distributed ledgers in sync, it depends on a three-phase consensus protocol:

  1. Pre-Prepare: primary node verifies and records a transaction proposal, and assigns an order number. The information is broadcasted to secondary nodes

as a Pre-Prepare message. Pre-Prepare messages are verified by listening secondary node and begin broadcasting a Prepare message to other secondary nodes.

2. Prepare: secondary nodes are actively receiving prepare messages from other secondary nodes, verifying them and ensuring consistency. After the verification, secondary nodes broadcast a Commit message.

3. Commit: Once enough commits are received by a node, verified and checked for consistency, the node confirms that the transaction has been voted by enough nodes.

- Paxos & RAFT:

  Similar algorithms to PBFT exist, like Paxos and RAFT. Different DLT frameworks have implemented algorithms to make use of the different advantages that they may have [31], [35]. A characteristic of these two consensus algorithms that is worth mentioning is that they are not Byzantine Fault Tolerant (BFT) unlike PBFT, instead, they are described as being Crash Fault Tolerant (CFT) [36]. Hyperledger Fabric, the DLT framework that we will be using for this project, is a permissioned DLT, meaning that nodes are verified members, and thus, focuses on resolving crashes rather than Byzantine faults [37]. In other words, the primary nodes are always expected to act correctly.

2.1.3.3 Smart Contracts

In the context of DLT applications, Smart Contracts are programmed functionalities that automate the exchange of data inside of a blockchain in a transparent manner. These mechanisms do not require a middleman to verify the validity of the exchange, since the mechanism is agreed to by all parties in the network, and all the movements and transactions are recorded into the distributed ledger. The storage of the transactions eliminates

mistrust between parties and the absence of a verifying middleman raises the cost and time efficiency [38]. Said smart contracts are stored as code in the DLT network, and are executed by members of the network, or they can be triggered automatically when the conditions match the policies [39].

2.1.3.4 Hyperledger Fabric

Fabric is an open-source, modular and general-purpose DLT framework that offers identity management and access control features. As presented in the Hyperledger Foundation website [40]:

Hyperledger Fabric is an enterprise-grade permissioned distributed ledger framework for developing solutions and applications.

Hyperledger Fabric Properties

Hyperledger Fabric, as explained by it's own websites [28], [36], [41], [42], presents the following properties:

- Permissioned DLT

  Fabric implements a permissioned DLT. The framework is intended for use within a single organization or a group of aligned organizations, allowing for members to interact with each other on the established network.

- Data Objects

  In Fabric data objects are referred to as Assets. These data objects can be modeled to represent anything that the organizations in a network agree on. Assets are represented in both binary and JSON form.

- Smart Contracts & Chaincode

  In Fabric, smart contracts are written in Chaincode (CC). CC is the software that defines the assets and all the instructions regarding the transformation of said assets;

also known as the business logic layer. These aforementioned functions are initiated by a transaction proposal from a client, and result in a set of key-value writes which can be submitted in the network for ledger state propagation.

- Distributed Ledger

  The way that Hyperledger Fabric manages its distributed ledger is through the implementation of a Blockchain and a World State Database.

  The Blockchain component is a sequenced and immutable record of all the state transitions that have occurred in the network. State transitions refers to the result of every transaction proposal that has been submitted. The data that is created or modified by the transactions are data objects referred to as assets by Hyperledger Fabric.

  The World State Database is used to record and access the current state of the data objects. Since the latest versions of the assets are available, the database engine allows the execution of queries without having to decrypt the latest block in the chain. Hyperledger Fabric supports two different storage options, LevelDB and CouchDB

  LevelDB is the default storage option, and offers simple key-value and, while very fast, does not support indexes or the execution of complex queries. CouchDB, the alternative database, is a document-oriented NoSQL engine. It allows for data indexing and JSON queries, offering more flexibility.

- Consensus

  Since version 1.4.1 of Hyperledger Fabric, the development of the deterministic consensus algorithm started to focus on using RAFT, the CFT consensus algorithm explored in the chapter 2.1.3.2 Consensus Algorithms. In figure 2.1 we summarize how consensus is reached during the transaction flow of Fabric.

- Network Components
  - Certificate Authority (CA): Entities that store, sign and issue digital certificates in order to ensure the ownership of a public key. Each organization should have and manage its own CA.
  - Peer: In Fabric, the nodes that maintain the ledger and execute the smart contract functionalities are known as Peers. These entities are owned and maintained by each organization in the network, and each participating organization should have one or more peer nodes in the network. In the 2.1.3.2 PBFT explanation, these would be the secondary nodes. Peers keep a copy of the ledgers that their respective organization have access to.
  - Ordering Service: Made up of nodes that ensure consistency of data in all of the other peers. They order the transactions in the blocks and perform the validation and commit mechanisms. In the 2.1.3.2 PBFT explanation, these would be the primary nodes.
  - Channels: A channel is the representation of a private blockchain. Channels are configured via Policies to allow organizations from joining via authentication, and also dictate which members can read from or write to the ledger. Each channel creates a private ledger, allowing for data isolation and confidentiality.
  - Dynamic Membership: Fabric supports adding and removing organizations, peers and ordering service nodes without disrupting operability in the network.
- Transaction Flow

  The way that the Fabric framework interacts with data objects is through Transactions. Transactions are the invocations of SC, the business logic packaged as CC in

Fabric. Fabric uses RAFT in order to reach consensus in the network. The workflow includes the following steps [43], [44]:

1. Creation of transaction: The flow begins when the client submits a transaction through the application. The transaction is sent to peers from each organization for endorsement.

2. Endorsement of transaction: Peers verify the client's identity and authority before simulating the outcome of the proposed transaction. The peers return the result of their endorsement as a signature back to the client.

3. Submission to Ordering Service: The client collects the endorsements from the peers. Once the amount of required endorsements has been received, the transaction is sent to the ordering service.

4. Verification and Commitment: The Ordering service verifies endorsements, if it matches, the approved transactions are chronologically ordered and packaged into blocks which are then sent to the peers of the participating organizations. The peers ratify, finalize and commit the transactions to the ledger.

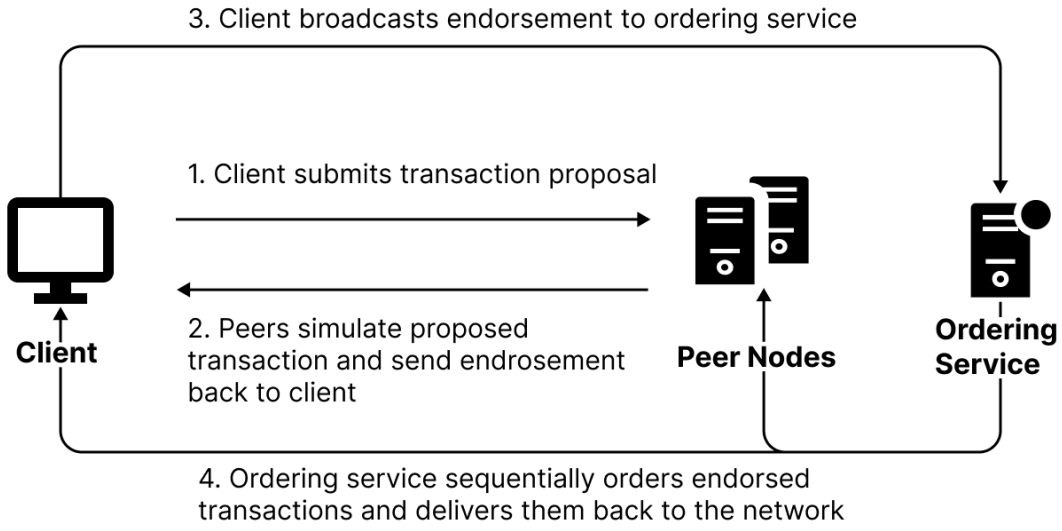

Figure 2.1: Hyperledger Fabric Transaction Flow Overview.

Other noteworthy properties of Hyperledger Fabric is that the execution environment is containerized in Docker, and the framework supports conventional high-level programming languages, like Java, JavaScript and Go, for the development of Smart Contracts and Applications.

2.2 Software Components

In this section, we overview the programming language used in the development of the solution and the OS-level virtualization used.

2.2.1 Go

Go or Golang is a compiled programming language designed at Google. The statically typed language is similar to C, but offers some extra features such as garbage collection, memory safety and concurrency options [45]. Concurrency is one of the main reasons why the language was chosen for the implementation of the project, since during the design phase of the solution it was assessed that the network latency measuring feature would need to run concurrently in order to make the solution as fast as possible.

While the solution could have also been implemented in other languages, like Python, Go has the advantage of being one of the native languages supported by the Fabric SDK, meaning that Smart Contracts from Fabric's Chaincode, the application that interfaces with the Fabric network, and the daemon application that performs the latency checks are all in the same language, making development more streamlined and code maintenance sustainable. Additionally, Go applications need to be compiled and built, and the resulting executable binaries do not have system dependencies.

2.2.2 Docker Containerization

The concept of Containerization, while decades old, became a popular way of packaging and deploying software code after the release of Docker in 2013. This paradigm consists of allowing the core of the OS, through process isolation and virtualization, to create lightweight components that can be executed individually by combining source code, operating system libraries and dependencies. It can be described as a lightweight virtualisation layer. The main benefit of the technology is allowing software to run consistently in heterogeneous architectures. Additionally, the execution of these containerized components is more portable and resource-efficient than using virtual machines (VMs). Developing software components using this paradigm results in components executable on any operating system that supports containerization, making them portable virtually anywhere [46].

Docker is a popular set of open-source platform-as-a-service (PaaS) products based on Containerization that enables managing standardized components. Therefore, as a container platform, we are required to package only the processes and dependencies necessary to execute code. These processes and dependencies are also shared in the host machine thanks to Docker, allowing for multiple containerized applications to be executed with very little overhead [47].

The reason why this particular paradigm is relevant to this project, is that it allows for applications to be packaged and executed in Linux, Windows and even macOS nodes. The only real limitation in terms of hardware is the capability of handling the Docker Engine, and having the required resources for the application to be virtualized. Additionally, the platform works in scenarios where servers with different hardware architectures and components need to be present. Docker makes the implementation of a distributed system in heterogeneous computer possible, and the available Application Programming Interface (API) can be adapted to allow a containerized application to create and run other containers in the host machine, a key functionality in the design and implementation of

this solution.

2.2.3 VPN

Virtual Private Network (VPN) is a technology that expands the availability of private networks across a public network. In order to ensure that the data transmitted between two or more devices in a VPN can not be accessed by other nodes in the public network, the connection is encrypted [48].

As a means for us to connect multiple devices in different private networks to test the design and implementation of the project, we will be using ZeroTier-One, an established solution that offers a free and secure VPN service [49].

3 Solution Design

In this chapter of the project we detail the design of the solution for Managing Heterogeneous Computing and Sensing Systems at the Edge.

The requirements that lead to the design of this solution are as follow:

- ***Collaboration***: Resources are required to interconnect with various other devices and share information between each other.
- ***Time Sensitive Analysis***: The solution must be capable of quickly capturing and analysing the resource availability in the network.
- ***Security, Trust & Reliability***: In order to achieve long-term functionality, the data in the system must be stored in a manner that prevents tampering, without affecting the ability to be shared, the stability or the speed of retrieval.
- ***Extensibility & Scalability***: Due to fast-paced introduction of new devices and technology, the design must allow for flexibility in terms of configuration and capabilities. The addition of devices to the network must be seamless, whether the new devices are meant to be nodes capable of processing data, or more IoT devices for the sensing system.

The platform that we set out to design and implement has the objective of securely storing the information of the devices from a Sensing System and the Edge Computing Resources that will be available for offloading tasks. The data that will be stored is a com-

bination of authentication information, resource status obtained by monitoring hardware and the latency that exists between the devices.

Having analysed and understood the base requirements and objectives of the system, the enabling technology components selected for the solution are:

- ***Hyperledger Fabric***: An enterprise-grade blockchain, stable and extensible framework for permissioned distributed ledgers.
- ***Go***: This programming language compiles components that are fast, easy to deploy and handle concurrency.
- ***Docker***: Feature-rich containerization platform for deploying applications with minimal interference to the Operating System.
- ***ROS***: Open-source framework for development of robotic solutions.
- ***ZeroTier One***: An open-source application that allows the creation and management of virtual private networks, enabling P2P connections between devices regardless of their physical connection.

The RoboMesh Platform for offloading at the edge is what was decided after consideration of the requirements, objectives and enablers available to us. As described by Figure 3.1 RoboMesh Platform Context Diagram, the components are the RoboMesh Platform, the devices that make up the Sensing System and the Edge Resources.

The first component is the RoboMesh Platform, this system is responsible for the management of the data that the devices in the network generate. This system is the combination of two components, a Hyperledger Fabric Network (HFN) with Smart Contracts and a REST Service combined with the Fabric SDK to expose an API that can be consumed by REST Clients or Web Applications.

The Sensing System is what we will refer to throughout the document as the IoT devices that capture data. The devices, or the computer that they are connected to, will deploy a Dockerized Daemon through which they will be monitored.

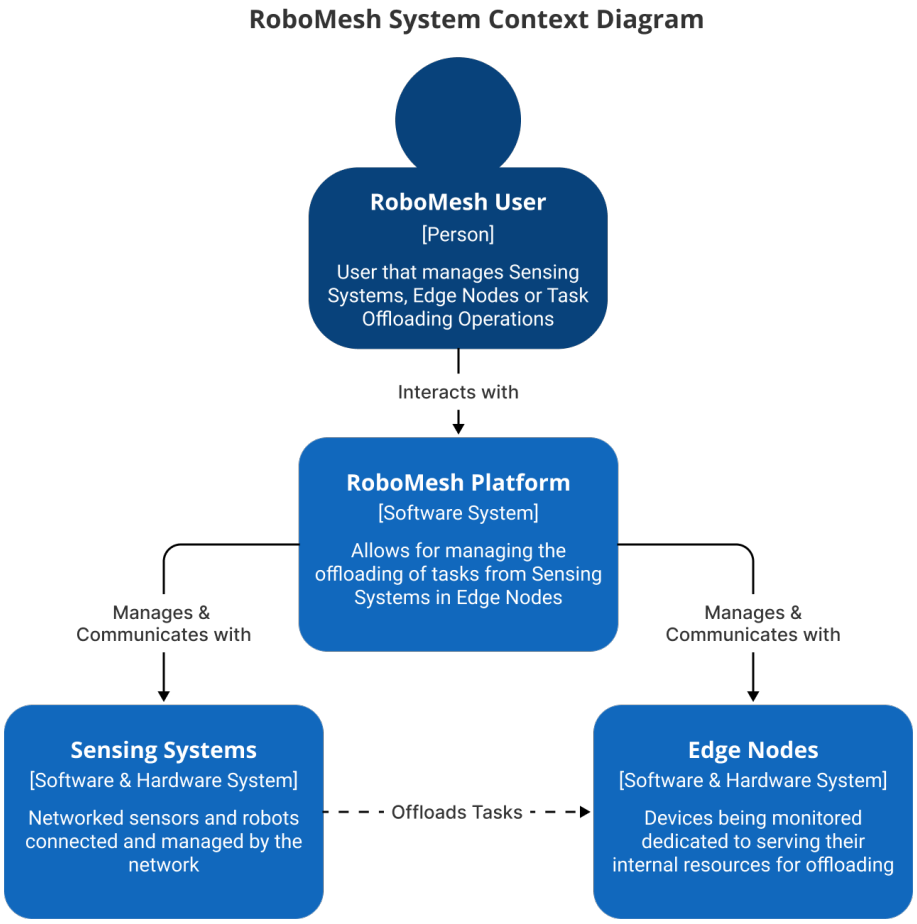

Figure 3.1: RoboMesh Platform Context Diagram

The Edge Resources are the hardware components in the network with the capabilities to perform tasks of higher complexity. Similarly, these computing devices will execute the Daemon in order to have their resources monitored, and gather which IoT devices against which they must measure latency. Additionally, the Daemon will serve as the tool in charge of executing the offloading tasks.

3.1 RoboMesh Platform

In this section of the document we will explore in detail the Smart Contract implementation and how we interface with the system. The Figure 3.2 RoboMesh Platform Container Diagram presents a more in-depth look at the interactions between the components that make up the system. The Distributed Ledger component and the API Application are the core of the solution.

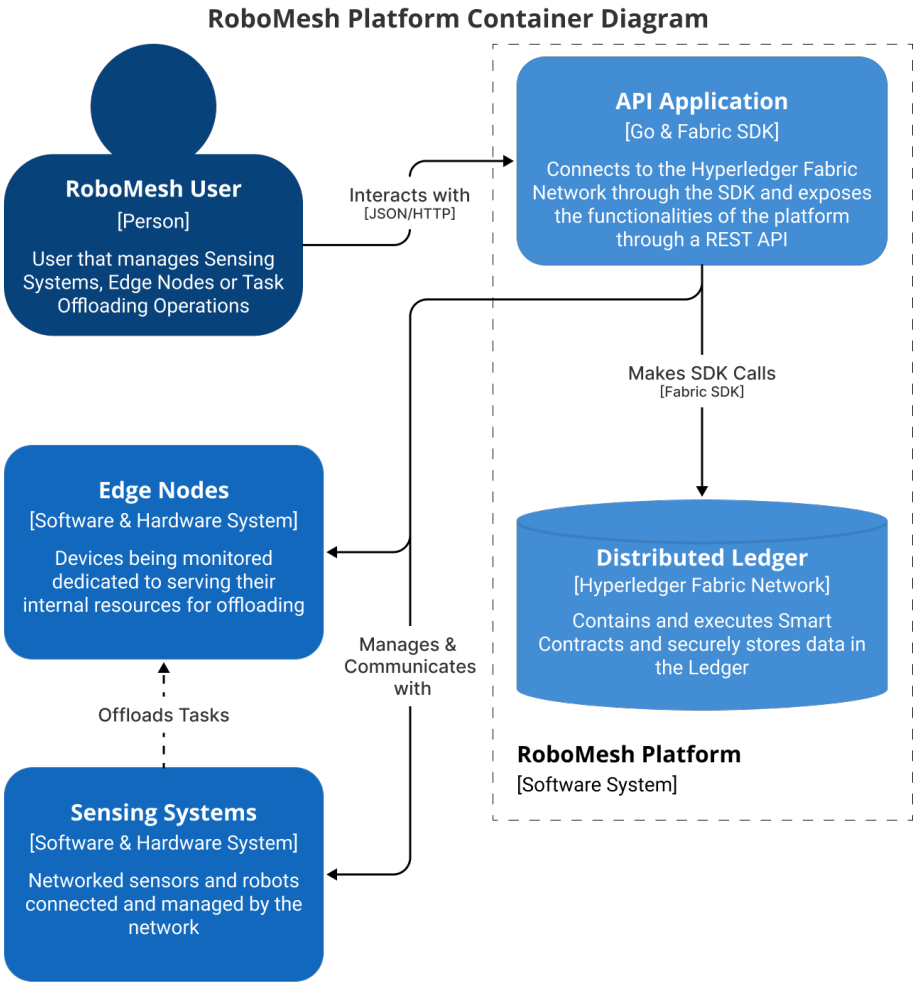

Figure 3.2: RoboMesh Platform Container Diagram

3.1.1 Hyperledger Fabric Network

In the subsubsection 2.1.3.4 Hyperledger Fabric we introduced the internal workings of the network and the components that encompass it. In this subsection we explain how the SPA, the API Application and HFN components of the system interact as described by the Figure 3.3 RoboMesh Platform Container Diagram. In the following sections, the code and data models are presented.

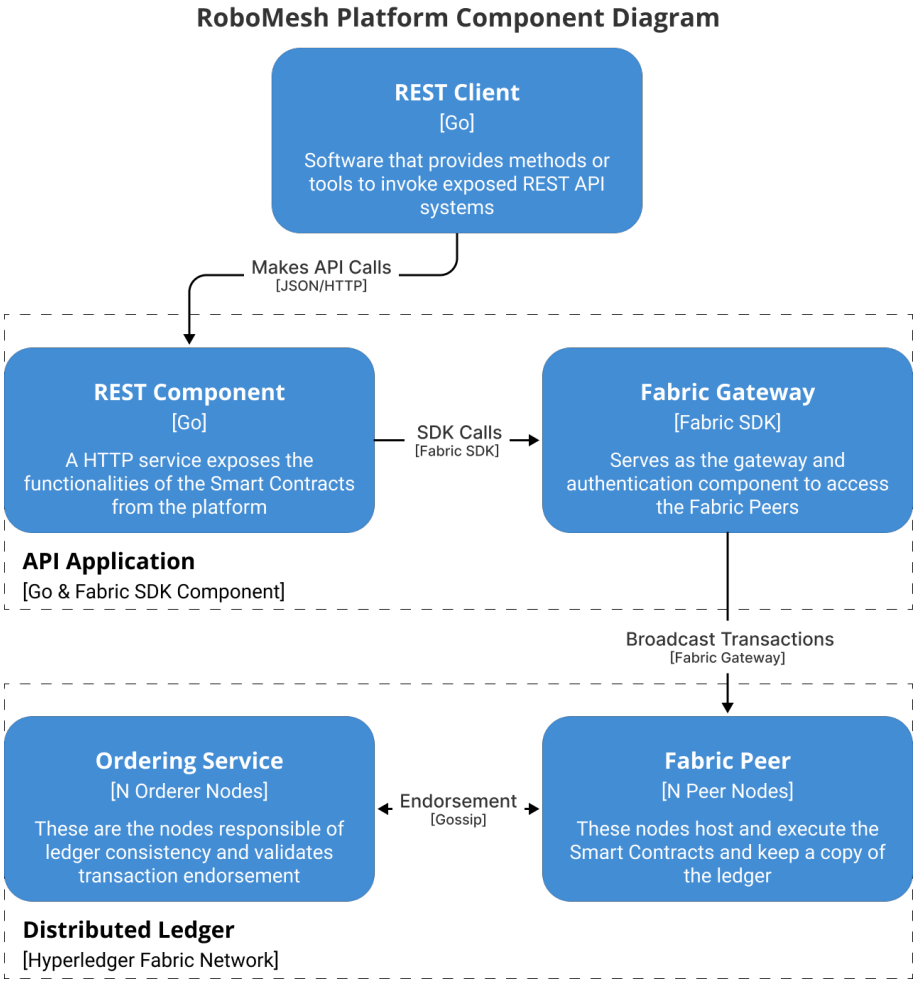

Figure 3.3: RoboMesh Platform Container Diagram

The components that make up the RoboMesh Platform communicate primarily by exchanging JSON objects, although the medium may differ. While the REST component uses HTTP, the components that make up the DLT network use Gossip, a data dissem-

ination protocol which signs messages to ensure authenticity and quickly forwards the contents between the connected peers [50], since there may exist numerous peer and ordering nodes.

In the sections to come, we will describe the object structure of the data that is stored in the ledger, used to communicate with the multiple components and the functions programmed. However, as to not duplicate parts of tables, whenever we mention operations with the intent to create, read, update and delete (CRUD) entries to the ledger, we refer to those detailed in Table 3.1 Common CRUD Operations.

Common CRUD Operations	
Function Name	**Description**
Read	Retrieves and returns the information of the device with the specified ID
Create	Created a device in the ledger with the information provided
Update	Updates the entry in the ledger for a device with the specified ID with the information provided
Delete	Deletes the entry from the world state database
Exists	Checks if a device with the provided ID exists in the world state database

Table 3.1: Common CRUD Operations

Apart from the CRUD operations, we have also designed a commonly used Timestamp Object for the implementation, which is detailed in the Table 3.2 Timestamp Object Structure. This object stores the time of execution of an operation in different formats. While storing the time following the ISO 8601 format is enough for human-readable requirements, storing the time in the Unix Timestamp format allows for quicker indexing abilities, since this format represents the current time elapsed in seconds since the Unix

Epoch on January 1st, 1970 at UTC [51].

Timestamp Object	
Function Name	**Description**
Local Time	ISO 8601 formatted local time of the device
Time in Seconds	Unix Timestamp formatted timestamp of the device in Seconds
Time in Milliseconds	Unix Timestamp formatted timestamp of the device in Milliseconds

Table 3.2: Timestamp Object Structure

3.1.1.1 Smart Contract: Inventory Management

The Inventory Management SC is responsible for securely storing the authentication procedure for accessing all the devices in the network. The stored object also contains some characteristics of devices required to properly filter and sort the required resources.

In the current implementation of the system, the authentication information of the devices is stored in plain text, as detailed in Table 3.4 Property Data Type Structure. This is a security risk and this behaviour must not be replicated in production environments. The decision behind it was for the sake of quickly implementing the proof-of-concept system. The recommended way to ensure that devices can connect to each other through SSH, without risking sensitive information being accessed by unauthorized actors, consists of generating SSH keys for each computer in the network and distributing them to the Edge Nodes accordingly.

Since device information is maintained, the main functions of this SC are the CRUD operations detailed in Table 3.1 Common CRUD Operations. However, in order to quickly filter and access the information of devices, more functionalities were added, as described by Table 3.5 Inventory SC Functions. These filters are useful when deciding what devices

must be targeted by the latency measurement functionality.

Inventory Asset		
Field	**Data Type**	**Description**
ID	String	Object Identifier
Name	String	Human-readable name of the asset
Owner	String	Organization to which the device belongs to
Type	Integer	Identifies the type of the device
State	Integer	Used to determine whether the device is enabled or disabled
Properties	Property	Contains the filterable characteristics of the device, and the authentication information

Table 3.3: Inventory Asset Object Structure

Property Data Type		
Field	**Data Type**	**Description**
GPU	Integer	Used to determine whether the device has a GPU
Hostname	String	The IP through which the device is accessed
HostPort	String	The port enabled for SSH connection
HostUser	String	The username used when connecting through SSH
HostPassword	String	The password used to authenticate

Table 3.4: Property Data Type Structure

Inventory SC Functions	
Function Name	**Description**
List Servers	Retrieves and returns all devices of type 0 stored
List GPU Servers	Retrieves and returns all devices of type 0 with a GPU stored
List Robots	Retrieves and returns all devices of type 1 stored
List Sensors	Retrieves and returns all devices of type 2 stored
List Robots and Sensors	Retrieves and returns all devices of types 1 and 2 stored
List All Assets	Retrieves and returns all of the existing devices

Table 3.5: Inventory SC Functions

3.1.1.2 Smart Contract: Resource Collection

The Resource Collection SC is responsible for securely storing the historic state of the computing devices and producing the necessary analysis of the data. In the current implementation, the SC is not directly responsible for measuring the resource usage of the devices, this responsibility falls on the program described in section 3.2 IoT Daemon.

Since the collection and maintenance of data is necessary the SC implements the CRUD operations detailed in Table 3.1 Common CRUD Operations. Other functionalities offered help accessing the data collected for a particular device and a simple statistical analysis of the data for a time window.

The resource data object, called StoredStat in the code, is described in the Table 3.7 Collected Resource Object Structure, and the analysis object in This object is derived from the ResourceStat object from the Daemon, a full description can be found in subsection 3.2.1 Resource Collection.

The statistical analysis objects are not stored in the database, these are JSON objects that the SC generates by gathering data the moment that the function is executed. The function requires the ID of the device and an amount of minutes. The system then finds

all the collected data of the device in the database, and the time frame is two timestamps: the first timestamp is the moment that the function is executed, and the second timestamp is generated by subtracting the amount of minutes indicated from the first. This means that the system always finds the latest recorded information, between the moment of function execution and some minutes into the past.

Resource SC Functions	
Function Name	**Description**
Get All Resources	Retrieves and returns all stored resource objects
Get All Resources for Device	Retrieves and returns all stored resource objects for a particular device
Get Resource for Device in Time frame	Retrieves and returns the stored resource objects of a device collected in a particular time window
Resource Analysis for Device in Time frame	Retrieves and analyses the resource objects of a device collected in a particular time window, returns the Resource Analysis Object

Table 3.6: Resource SC Functions

Collected Resource		
Field	**Data Type**	**Description**
ID	String	Object Identifier, created by combining the Inventory ID and the timestamp at the time of the collection
Hostname	String	Refers to the Inventory ID, stored on its own for indexing and querying purposes
Timestamp	Timestamp Type	Timestamp Object
Host Information	Host Information Type	Information that the Operative System from the device supplies
CPU Status	CPU Status Type	CPU Information and usage at the time of collection
Memory Status	Memory Status Type	Total capacity of memory and usage percentage at time of collection
Disk Status	List of Disk Status Type	Lists information and capacity of found disk drives in device
Process Status	Process Status Type	Information regarding the number of running processes in the system
Docker Status	List of Docker Status Type	Lists the docker containers found in the system and their information

Table 3.7: Collected Resource Object Structure

Resource Statistical Analysis		
Field	**Data Type**	**Description**
Hostname	String	Refers to the Inventory ID
Duration	Integer	Used to generate the time frame from which the historic data will be selected
Average CPU	Float	Average CPU Usage during the time frame
Average Memory	Float	Average Memory Usage during the time frame
Containers	Integer	Number of running docker containers
Stat Summary	List of Resource Object Summary Type	Summarized version of the Collected Resource Object

Table 3.8: Resource Statistical Analysis Structure

Resource Object Summary		
Field	**Data Type**	**Description**
ID	String	ID from the Collected Resource object from which this data is generated
Timestamp	Timestamp Type	Timestamp Object
CPU Average Usage	Float	Average CPU Usage during time frame
Memory Average Usage	Float	Average Memory Usage during the time frame
Running Containers	Integer	Number of running docker containers

Table 3.9: Resource Object Summary Structure

3.1.1.3 Smart Contract: Latency Collection

The Resource Collection SC is responsible for securely storing the historical latency information between the devices that the Daemon generates. Since this SC also stores data into the ledger, it requires all of the functionalities that were described in Table 3.1 Common CRUD Operations.

The other functions that this SC requires are similar to those in the Resource SC; they help filter out all of the latency measurements done by a certain host, or that include a certain device as their target. Also in this case, a function generates a summarized analysis of the historical data, which are detailed in the Table 3.10 Latency SC Functions.

Latency SC Functions	
Function Name	**Description**
Get All Measurements	Retrieves and returns all stored latency objects
Get All Measurements by Device	Retrieves and returns all measurements done by a device in a specific time frame
Get All Measurements by Target	Retrieves and returns all measurements done to a device in a specific time frame
Latency Analysis by Target	Retrieves and analyses the latency measurements done to a device collected in a particular time window, returns the Latency Analysis Object
Get Latency Targets	Returns a list of devices that the asking server needs to measure latency with

Table 3.10: Latency SC Functions

This SC stores the results of the Latency Measurement Operation described in the subsection 3.2.2 Network Latency Measurement. The object fields are described by the Table 3.11 Latency Measurement Object Structure. The function that analyses the historic

latency information does not store data and is generated on execution, it returns a list of the analysis object, explained by the Table 3.8 Resource Statistical Analysis Structure. In the list there will be an entry of analysis for each server that measured its latency against the selected target. In order for the servers to know which devices they have to measure their latency, the servers can request a list of targets, the object fields are specified in the Table 3.14 Latency Target Type Structure.

Latency Measurement Object		
Field	**Data Type**	**Description**
ID	String	Object Identifier, created by combining the Inventory ID and the timestamp at the time of the collection
Source	String	Refers to the Inventory ID, stored on its own for indexing and querying purposes
Timestamp	Timestamp Type	Timestamp Object
Results	List of Latency Result Type	List of all the latency operations executed by the server

Table 3.11: Latency Measurement Object Structure

Latency Result Type		
Field	**Data Type**	**Description**
Hostname	String	Refers to the Inventory ID of the device that was measured by the server
Latency	Integer	The milliseconds it took to execute the operation

Table 3.12: Latency Result Type Structure

Latency Analysis Object		
Field	**Data Type**	**Description**
Hostname	String	Refers to the Inventory ID of the server that performed the measurement operation
Target	String	Refers to the Inventory ID of the device that was measured by the server
Duration	Integer	Used to generate the time frame from which the historic data will be selected
Average Latency	Float	The average milliseconds that took the executing server to connect to the target device
Latency Summary	List of Integer	List of analysed latency results from the past in milliseconds

Table 3.13: Latency Analysis Object Structure

Latency Target Object		
Field	**Data Type**	**Description**
Hostname	String	Refers to the Inventory ID of the server that requested the information
Targets	List of Inventory Asset	Returns a list of Inventory objects, including the SSH authentication information

Table 3.14: Latency Target Type Structure

3.1.1.4 Smart Contract: Offload Selection Collection

The Offload SC, while also being responsible for the storage of data, is responsible for selecting which server will take part in the offloading of a task, based on the latency and resource status stored by the previously described contracts.

The SC also implements the necessary CRUD operations described in the Table 3.1 Common CRUD Operations in order to securely store and access the Selection Object detailed by the Table 3.15 Selected Server Object Structure.

To summarize the process that selects the correct server, described with pseudo-code in Algorithm 1: Offload Server Selection, the process starts by obtaining the analysis of Latency Measurements previously performed at the targeted device. The system then obtains the list of devices identified as servers, depending on whether the property of the task requires the Edge Node to have a GPU. To ensure that only available devices will be in the selection, the list is filtered to only include those Nodes which have performed the latency measurement in the specified time frame. Using the filtered list of devices, we obtain the analysis of hardware resources of each Node concurrently. Latency and Resource analyses are merged into a list of Selection Objects. The list is then sorted in ascending order by the Latency Average. In the case that two or more devices share the same average, the list is sorted in ascending order by CPU Average, then by Memory Usage Average and finally by the amount of Containers being executed. This sorting method ensures that the server with the best connection or more available resources is selected.

Algorithm 1: Offload Server Selection

Input:

Device to Offload task from: $target$

Task Properties: $taskProperties$

Minutes for analysis time frame: $minutes$

Output:

Selected Server : $selected$;

List of other Servers : $selectedServerList$;

$latencyAnalysis$ = $latencySC$.AnalyseLatencyToTarget($target$, $minutes$);

if $taskProperties$.*GPU == TRUE* **then**

    $serverList$ = $inventorySC$GetServerListGPU();

else

    $serverList$ = $inventorySC$GetServerList();

$filteredServerList$ = removeUnusedServers($serverList$);

$resourceAnalysis$ = [];

`// Analyse each server in list concurrently`

for *server in filteredServerList* **do**

    $resource$ = $resourceSC$.AnalyseResources($server$);

    $resourceAnalysis$.append($resource$);

$selectedServerList$ = combineAnalysis($resourceAnalysis$, $latencyAnalysis$);

$selectedServerList$.sortByValue("latency", "averageCPU");

$http$.Post(applicationUrl, $selectedServerList$[0]);

return $selectedServerList$

Selected Server Object		
Field	**Data Type**	**Description**
ID	String	Object Identifier, created by combining the Inventory ID of the device to be offloaded and the timestamp at time of selection
Timestamp	Timestamp Type	Timestamp Object
Target	String	Inventory ID of the device to be offloaded, stored for indexing purposes
Selected Asset	String	Inventory ID of the Edge Node for offloading
Average Latency	Float	Average latency at time of analysis
CPU Average Usage	Float	Average CPU Usage at time of analysis
Memory Average Usage	Float	Average Memory Usage at time of analysis
Running Containers	Integer	Number of running docker containers

Table 3.15: Selected Server Object Structure

3.1.1.5 Application: Fabric Network Gateway

In order to establish a way to interact with the multiple Smart Contracts hosted in the HFN, we have implemented a Gateway application. This Gateway provided by the Fabric SDK uses the network's signed certificates to authenticate, and can communicate with the Peer Nodes and Ordering Service to execute the transactions and processes of the Smart Contracts. The application has a built-in HTTP server that exposes REST Endpoints in order for the users, and other parts of the network, to communicate and retrieve the data stored in the Ledger.

The first set of Endpoints, detailed in Table 3.16 REST Endpoints of the Application for Daemon is destined to be accessed only by the Daemon Application. The tables in this subsection depict the URL from which the functions can be executed, and which SC or functionality it correlates to.

REST Endpoints for Daemon	
Method and Route	**Description**
POST "/collector"	Used for collecting Resources from Daemon
POST "/measurement"	Used for collecting Latency Measurement from Daemon

Table 3.16: REST Endpoints of the Application for Daemon

The rest of the routed endpoints of the application are all designed to execute the functionalities implemented by the Smart Contracts, as described in the following tables.

Inventory REST Endpoints	
Method and Route	**Description**
GET "/inventory"	List All Assets
GET "/inventory/servers"	List Servers
GET "/inventory/servers/gpu"	List GPUServers
GET "/inventory/robots"	List Robots
GET "/inventory/sensors"	List Servers
GET "/inventory/:asset"	Read Asset by ID
PUT "/inventory/:asset"	Update Asset by ID
POST "/inventory/"	Creates Asset based on Payload

Table 3.17: REST Endpoints of the Application for Inventory SC

Resources REST Endpoints	
Method and Route	**Description**
GET "/resources"	List All Resources
GET "/resources/device/:device"	List All Resources for Device
GET "/resources/device/:device/minutes/:minutes"	List Resources for Device in time frame
GET "/resources/analysis/device/:device/minutes/:minutes"	Resource Analysis for Device in time frame
GET "/resources/:id	Read Resource by ID
PUT "/resources/:id	Update Asset by ID
POST "/resources"	Creates Resource based on Payload

Table 3.18: REST Endpoints of the Application for Resources SC

Latency REST Endpoints	
Method and Route	**Description**
GET "/latency"	List All Latency
GET "/latency/target"	Get Latency Targets
GET "/latency/source/:device/minutes/:minutes"	Get All Measurements by Device
GET "/latency/target/:device/minutes/:minutes"	Get All Measurements by Target
GET "/latency/analysis/device/:device/minutes/:minutes"	Latency Analysis for Device in time frame
GET "/latency/:id	Read Latency by ID
PUT "/latency/:id	Update Latency by ID
POST "/latency"	Creates Latency based on Payload

Table 3.19: REST Endpoints of the Application for Latency SC

Selector REST Endpoints	
Method and Route	**Description**
GET "/selector"	List All Selections
GET "/selector/:id"	Read Selection by ID
GET "/selector/target/:device"	Get All Selections for Target
GET "/selector/asset/:device"	Get All Selections of Device
GET "/selector/target/:target/minutes/:minutes"	Select Device for Target in time frame

Table 3.20: REST Endpoints of the Application for Selector SC

3.2 IoT Daemon

In order to collect the necessary data from Edge Nodes, perform the latency measurement operations and execute the offloaded tasks, we designed and implemented a Daemon Service in order for all the required tasks to be executed repeatedly. A Daemon is a computer software that continuously runs in order to perform a specific service [52].

The software implemented in this project uses a CRON Scheduler to perform operating system, HTTP and Docker operations at a specified time rate. CRON is a Linux command that allows tasks to be scheduled for execution at a specific time or periodically [53]. The rate of time at which the operations are executed is configurable by changing the value, expressed in seconds, in the configuration file of the application. By default, the Scheduler executes the programmed tasks every 30 seconds, the tasks are detailed in the Table 3.21 Daemon Scheduled Operations.

Our Daemon is deployed in all of the devices of the network in a Docker Container. It was programmed to handle a limited amount of operations and it exposes three HTTP endpoints to allow for remote communication described in Table 3.22 Daemon HTTP Exposed Endpoints.

Daemon Scheduled Operations	
Function Name	**Description**
Resource Collection	Collects hardware information and status from the host
Perform Latency Measurement	Retrieves a list of devices from Application and measures latency between host and targets

Table 3.21: Daemon Scheduled Operations

Daemon Exposed Endpoints	
Method and Route	**Description**
GET "/heartbeat"	Collects hardware information and returns it without uploading the results to the Application
GET "/latency"	Forces the execution of the scheduled Latency Measurement operation but does not upload results to the Application
POST "/latency"	Performs the latency measurement operation to targets specified in method payload

Table 3.22: Daemon HTTP Exposed Endpoints

3.2.1 Resource Collection

In order to correctly select the Edge Node with enough hardware resources for offloading the tasks, the Daemon is tasked with collecting the hardware status information on the device, this is the data that is used for analysis in the Resources SC of the HFN.

The information collected by the application is detailed in the Table 3.23 Daemon Collected Resource Object Structure and the subsequent tables. It is worth noting that at the time of collection, the Daemon does not assign an ID to the data, instead, the HFN Gateway Application is responsible for determining which host is sending the information and assigning it a unique identifier for storage in the Distributed Ledger.

Daemon Collected Resource		
Field	**Data Type**	**Description**
Timestamp	Timestamp Type	Timestamp Object
Host Information	Host Information Type	Information that the Operative System from the device supplies
CPU Status	CPU Status Type	CPU Information and usage at the time of collection
Memory Status	Memory Status Type	Total capacity of memory and usage percentage at time of collection
Disk Status	List of Disk Status Type	Lists information and capacity of found disk drives in the device
Process Status	Process Status Type	Information regarding the number of running processes in the system
Docker Status	List of Docker Status Type	Lists the docker containers found in the system and their information

Table 3.23: Daemon Collected Resource Object Structure

Host Information Type		
Field	**Data Type**	**Description**
Hostname	String	Human-readable name assigned to the device
Boot Time	Integer	Seconds since the device booted
Platform	String	Operating System Platform
Virtualization System	String	The virtualization platform in which the system relies
Virtualization Role	String	Whether the system is a virtualiztion guest or host
Host ID	String	Unique host ID provided by the OS

Table 3.24: Host Information Type Structure

CPU Status Type		
Field	**Data Type**	**Description**
Model Name	String	Model of the CPU
Vendor ID	String	ID of the CPU Manufacturer
Average Usage	Float	Average usage percentage of the CPU
Core Usage	List of Floats	Usage percentage of each CPU Core

Table 3.25: CPU Status Type Structure

Memory Status Type		
Field	**Data Type**	**Description**
Total	Integer	Total Amount of Memory in Bytes
Available	Integer	Amount of available Memory in Bytes
Used	Float	Percentage of used memory

Table 3.26: Memory Status Type Structure

Disk Information Type		
Field	**Data Type**	**Description**
Device	String	Name of the Disk
Path	String	Path through which the Disk is accessed
Label	String	Human-readable name of Disk
FsType	String	File System Type of Disk
Total	Integer	Total size of Disk in Bytes
Used	Integer	Amount of Bytes of Disk used
Used Percentage	Float	Percentage of Disk used

Table 3.27: Disk Status Type Structure

Process Status Type		
Field	**Data Type**	**Description**
Total	Integer	Amount of processes running
Created	Integer	Amount of processes created since boot
Running	Integer	Amount of currently executing processes
Blocked	Integer	Amount of currently blocked processes

Table 3.28: Process Status Type Structure

Docker Status Type		
Field	**Data Type**	**Description**
Container ID	String	UUID of the Docker Container
Name	String	Human-readable name of Docker Container
Image	String	Image name of the container
Status	String	Status information of the container
State	String	Determines whether the container is currently running

Table 3.29: Docker Status Type Structure

3.2.2 Network Latency Measurement

Performs network latency analysis between the host and the selected targets, instead of relying on Ping, we authenticate into the system and simulate data transfer to more accurately predict the time-to-reply of the systems.

The process is explained by the Algorithm 2: Latency Measurement

In summary, the algorithm obtains a list of devices to which the program needs to connect. In a concurrent fashion, the algorithm starts tracking the execution time as it performs SSH connections to the devices with the obtained authentication data, and prints information to check the validity of each connection. If the connection is valid, the connection is closed and the execution time is stopped. Once all the concurrent tasks are finished, the list of measurements is sent to the Application via an HTTP Post. In the case that a remote connection fails, or the information printed by the device is compromised, the result is stored as a -1. This negative value is ignored when performing the latency analysis between the devices, but it remains in the ledger.

Algorithm 2: Latency Measurement

Output:

List of Latency Results : $measurementResults$;

```
// Get Targets from Application
```

$targetList$ = $http$.Get(applicationUrl);

$measurementResults$ = [];

```
// Perform tasks concurrently
```

for *target in targetList* **do**

- $startTime$ = $time$.Now();
- $sshConnection$ = ssh.Dial(target.Authentication);
- $sshConnection$.executeCommand("echo " + startTime);
- $readTerminal$ = $sshConnection$.readBuffer();
- `// Checks validity of connection and value of string`
- **if** *(readTerminal == startTime) && (sshConnection == TRUE)* **then**
  - $sshConnection$.close();
  - $elapsedTime$ = $time$.Since(startTime).Milliseconds();
- **else**
  - $elapsedTime$ = -1;
- $measurementResults$.append("target": target.ID, "latency": elapsedTime);

```
// Send results to Application
```

;

$http$.Post(applicationUrl, measurementResults);

return $measurementResults$;

4 Implementation and Experiments

In this section, we describe the hardware elements of the devices used to test the platform. We also describe the different experiments made, including the different network topology configurations of the devices and the data that was gathered.

4.1 Hardware Components

In order to test the network with a variety of servers and devices to act as Edge Nodes or IoT devices, we have selected four different computers and one router. The following tables describe the role that each of these devices will have during the experimentation.

HNF Server	
Field	**Description**
Role	HNF Host and Server
OS	Ubuntu 20.04
CPU	Intel i7-10750H (12) @ 5.000GHz
GPU	NVIDIA GeForce RTX 2060
Memory	64GB
Connection Type	Ethernet and WiFi

Table 4.1: HNF Server Hardware Details

Up Board Squared	
Field	**Description**
Role	Server
OS	Ubuntu 18
CPU	Intel Atom E3950 (4) @ 2.000GHz
GPU	N.A.
Memory	8GB
Connection Type	Ethernet

Table 4.2: Up Board Squared Details

Raspberry Pi 4 Model B Rev 1.1	
Field	**Description**
Role	IoT Device
OS	Raspbian GNU/Linux 10
CPU	ARMv7 BCM2711 (4) @ 1.500GHz
GPU	N.A.
Memory	4GB
Connection Type	Ethernet and WiFi

Table 4.3: Raspberry Pi 4 Details

Raspberry Pi 3 Model B Rev 1.2	
Field	**Description**
Role	Server
OS	Raspbian GNU/Linux 11
CPU	ARMv7 BCM2835 (4) @ 1.200GHz
GPU	N.A.
Memory	1GB
Connection Type	Ethernet

Table 4.4: Raspberry Pi 3 Details

DLink DIR-882	
Field	**Description**
Role	Router
Connection Type	Ethernet and WiFi

Table 4.5: DLink DIR-882 Details

4.2 Experimental Results

In order to test the efficacy of the platform in a realistic environment, we have prepared five different network configurations as scenarios. All the experiments will consist of performing an Edge Node Selection, regardless of configuration, and will follow this procedure:

1. Set up all devices, ensure connectivity and execute the Daemon container
2. Start the HFN and purge old data from blockchain and database to avoid faulty analysis
3. Start the Gateway Application
4. Allow the system to collect data and perform latency measurements for 15 minutes. The Daemon is configured to schedule operations every 30 seconds.
5. Execute the Edge Node Selection functionality for tasks that do not require GPU with a 10 minute time frame analysis

Once the selection has been performed, the data is retrieved and evaluated. Additionally to the data created and stored by the Smart Contracts, the data outputted by the HTTP Server of the Gateway Application will let us know the average duration in milliseconds of function execution.

4.2.1 Local Network: Ethernet

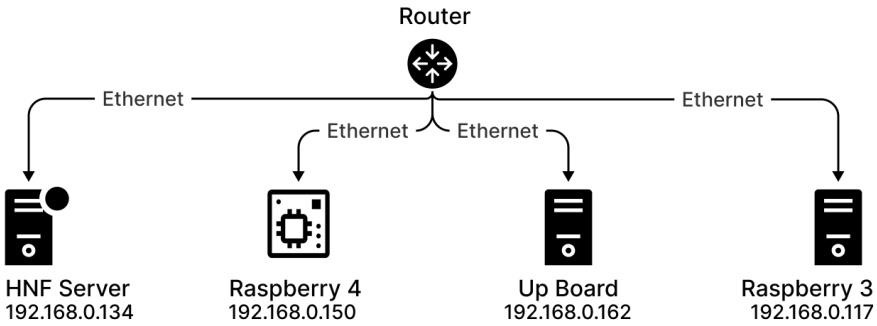

Figure 4.1: Experiment Configuration - Local Network: Ethernet

For this experiment, all computers are connected with an Ethernet cable to the router. All network operations are done using the Local Area Network IP Addresses assigned by the router's DHCP.

Local Network: Ethernet				
Edge Node	**Connection**	**Latency**	**CPU**	**Memory**
HNF Server	Ethernet	274.4 ms	4.97%	29.13%
Up Board	Ethernet	273.08 ms	2.08%	9.39%
Raspberry 3	Ethernet	280.8 ms	4.88%	14.36%

Table 4.6: Local Network Experiment Data - Ethernet

The Asset selected as the offloading node in this configuration was the Up Board. In this configuration the latency difference between nodes is negligible. Although the recorded latency seems high, the *ping* command indicates that the latency between both the selected server and the computer acting as the sensor was 0.659 milliseconds.

During the experiment, the HTTP Server reported that the average duration for a read-only operation was 5.23ms, and the average of a write operation 1.67s.

4.2.2 Local Network: Ethernet and WiFi

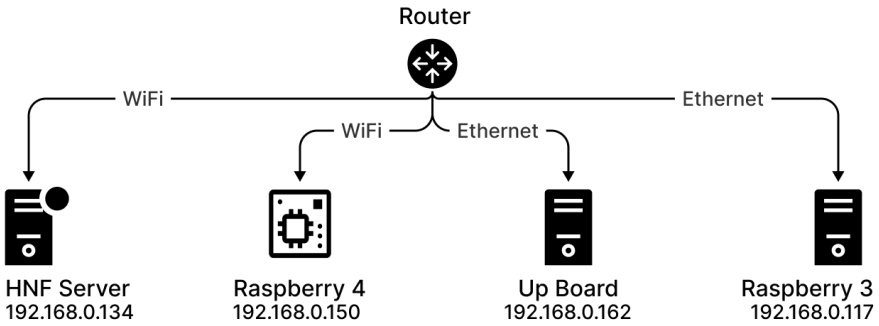

Figure 4.2: Experiment Configuration - Local Network: Ethernet and WiFi

In this experiment, computers with the capability are connected to the router using WiFi. All network operations are done using the Local Area Network IP Addresses assigned by the router's DHCP.

Local Network: Ethernet and WiFi				
Edge Node	**Connection**	**Latency**	**CPU**	**Memory**
HNF Server	WiFi	526.38 ms	5.70%	29.14%
Up Board	Ethernet	286.5 ms	2.05%	9.35%
Raspberry 3	Ethernet	335.3 ms	5.77%	13.88%

Table 4.7: Local Network Experiment Data - Ethernet and WiFi

The selected Edge Node for offloading was the Up Board, averaging 286ms of latency measurement. The *ping* operation from the Up Board to the Raspberry 4 in this configuration averages 2.22ms, a noticeable increase from the first experiment.

During the experiment with this configuration the duration of operations remained close to those from the base experiment, averaging 5.58ms and 1.82s for read-only and write operations respectively.

4.2.3 Local Network and VPN: Ethernet

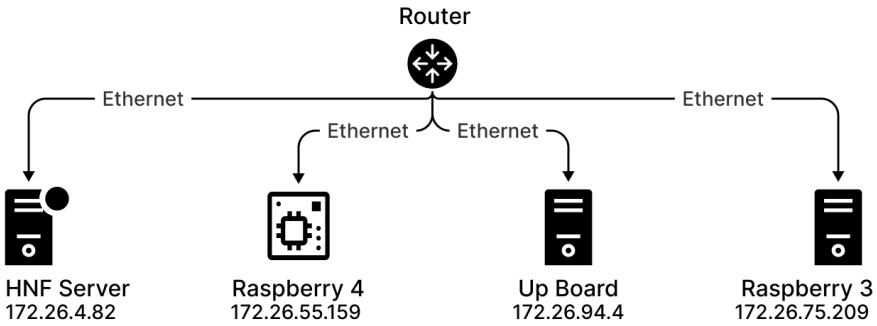

Figure 4.3: Experiment Configuration - Local Network and VPN: Ethernet

For this experiment, all computers are connected with an Ethernet cable to the router. All network operations are done using the IP Addresses assigned by the ZeroTier One VPN service.

Local Network and VPN: Ethernet				
Edge Node	**Connection**	**Latency**	**CPU**	**Memory**
HNF Server	Ethernet	276.18 ms	5.49%	29.62%
Up Board	Ethernet	306.62 ms	4.52%	9.05%
Raspberry 3	Ethernet	357.07 ms	11.82%	13.57%

Table 4.8: Local Network and VPN Experiment Data - Ethernet

In this case the HFN Server was selected as the best Edge Node, and the *ping* commands averages at 1.16ms. While the latency has increased in comparison to the first experiment, it is still acceptable for most operations.

With the addition of the VPN, the duration of the read-only and write operations seems to have increased by a negligible amount, they average 6.2ms and 2.11s respectively.

4.2.4 Local Network and VPN: Ethernet and WiFi

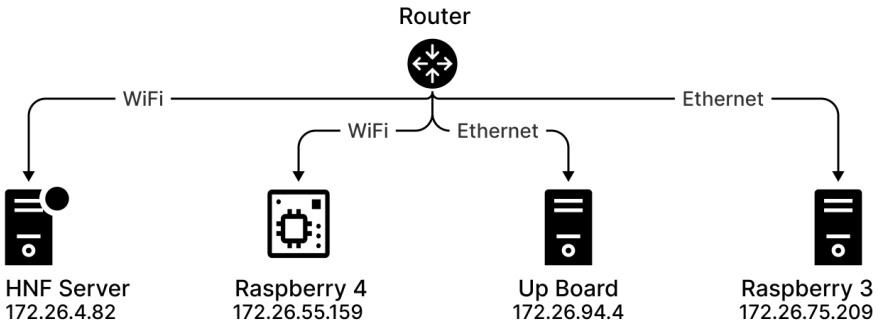

Figure 4.4: Experiment Configuration - Local Network and VPN: Ethernet and WiFi

In this experiment, computers with the capability are connected to the router using WiFi. All network operations are done using the IP Addresses assigned by the ZeroTier One VPN service.

Local Network and VPN: Ethernet and WiFi				
Edge Node	**Connection**	**Latency**	**CPU**	**Memory**
HNF Server	WiFi	331.05 ms	4.49%	29.84%
Up Board	Ethernet	341.86 ms	4.47%	9.03%
Raspberry 3	Ethernet	388.94 ms	12.39%	13.79%

Table 4.9: Local Network VPN Experiment Data - Ethernet and WiFi

The selected Edge Node, in this case, was the HNF Server, with the latency measurement averaging at 331.05ms and the *ping* command at 3.49ms. This experiment has made clear that using both WiFi and a VPN makes an impact, but the averages between the servers are consistent.

The HTTP Server did not seem to have been too affected, with the durations averaging 6.02ms and 2.13s for read-only and write respectively.

4.2.5 Mixed Network: Ethernet and WiFi

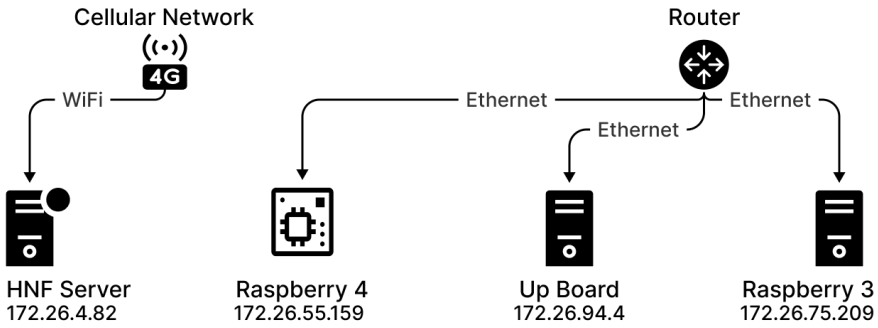

Figure 4.5: Experiment Configuration - Mixed Network: Ethernet and WiFi

In this experiment, computers with WiFi are connected to a 4G cellular band. Other computers remain in the original network. All network operations are done using the IP Addresses assigned by the ZeroTier One VPN service.

Mixed Network: Ethernet and WiFi				
Edge Node	**Connection**	**Latency**	**CPU**	**Memory**
HNF Server	WiFi (Cellular Network)	4735.5 ms	1.98%	30.82%
Up Board	Ethernet	448.33 ms	2.50%	9.32%
Raspberry 3	445.21 ms	2.61%	12.64%	

Table 4.10: Mixed Network Experiment Data - Ethernet and WiFi

This final experiment confirms that even when using different networks, the system is still capable of finding an Edge Node. The HNF Server connected to a 4G Cellular Network has the most latency, and the Up Board was selected as the Edge Node, which is connected to the same network as the Raspberry 4. The *ping* average between the selected node and the target device is 3.52ms. During this experiment, the averages for the read-only and write operations did change, with the durations being 20.12ms and 3.11s respectively.

5 Conclusion

Motivated by the hypothesis of whether a platform that could accurately select an Edge Node for the offloading of tasks with Hyperledger Fabric Smart Contracts was feasible, we have presented the design and implementation of a platform that can analyze the latency and resource availability of nodes in order to select the most fitting server. The experiments confirm that the permissioned network implemented in the HFN allows for reliable, trustable and fast distributed applications. We have also presented the multiple experiments and the data that was gathered from them. From these experiments we were also able to demonstrate the long-range capability of the system, working at manageable speeds even when using VPN services and 4G cellular networks.

When using Smart Contracts as the core for an application, the structure of the object is of utmost importance. During the design process, it must be kept in mind that there are some limitations to the database queries that we can perform, since the world state is recorded in a NoSQL engine, and the intent of indexing must be there from the start. During the experimentation, it became clear that the rate of resource collection and latency measurement plays an important role in the outcome of the selection, and the rate will need to vary depending on the requirements of the platform, and may even need to change from device to device.

5.1 Future works

In future work, we will expand the platform functionalities to determine whether Smart Contracts are also a feasible mechanism to share state information required for executing the offloading of tasks after having selected an Edge Node. The current system can accurately choose nodes based on their current state, and whether the task requires or not a GPU, but the system is not capable of starting the execution of the tasks, it does not have the capabilities to track the execution of said tasks nor does the Edge Node Selection happen autonomously.

Currently, the functionalities of the system do not include a mechanism that ensures that the growth rate of the world state database will remain constant, which makes the long-term autonomy evaluation of the platform a challenge. We started designing methods in which older data can be archived in Cloud systems to avoid the indexing process of the ledger from deteriorating, thus maintaining the speed of the analysis.

Additionally, we would like to study the performance of the long-range capabilities of the system using a 5G Mesh Network, and the efficacy of the Offload Selection for robotics systems that require real-time processing.

References

[1] E. Hamilton, *What is edge computing: The network edge explained*, Dec. 2018. [Online]. Available: https://www.cloudwards.net/what-is-edge-computing/.

[2] W. Shi, G. Pallis, and Z. Xu, *Edge computing [scanning the issue]*, 2019. DOI: 10.1109/JPROC.2019.2928287.

[3] K. Marneweck *et al.*, *Why data-over-sound should be a part of any iot engineer's toolbox*, 2019. [Online]. Available: https://www.arm.com/resources/white-paper/data-over-sound.

[4] A. Panwar and V. Bhatnagar, "Distributed ledger technology (dlt): The beginning of a technological revolution for blockchain", in *2nd International Conference on Data, Engineering and Applications (IDEA)*, 2020, pp. 1–5. DOI: 10.1109/IDEA49133.2020.9170699.

[5] S. K. Krause, N. Harish, and H. L. Gradstein, "Distributed ledger technology (dlt) and blockchain", Dec. 2017. [Online]. Available: https://documents.worldbank.org/en/publication/documents-reports/documentdetail/177911513714062215/distributed-ledger-technology-dlt-and-blockchain.

[6] ITU, *Itu-t technical report: Focus group on application of distributed ledger technology*, Aug. 2019. [Online]. Available: https://www.itu.int/en/ITU-T/focusgroups/dlt/Documents/d12.pdf.

[7] R. Yang, F. R. Yu, P. Si, Z. Yang, and Y. Zhang, "Integrated blockchain and edge computing systems: A survey, some research issues and challenges", *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1508–153++2, 2019. DOI: 10.1109/COMST.2019.2894727.

[8] Z. Xiong, Y. Zhang, D. Niyato, P. Wang, and Z. Han, "When mobile blockchain meets edge computing", *IEEE Communications Magazine*, vol. 56, no. 8, pp. 33–39, 2018. DOI: 10.1109/MCOM.2018.1701095.

[9] S. Salimi, J. P. Queralta, and T. Westerlund, *Towards managing industrial robot fleets with hyperledger fabric blockchain and ros 2*, 2022. DOI: 10.48550/ARXIV.2203.03426. [Online]. Available: https://arxiv.org/abs/2203.03426.

[10] J. P. Queralta, L. Qingqing, Z. Zou, and T. Westerlund, *Enhancing autonomy with blockchain and multi-access edge computing in distributed robotic systems*, 2020. DOI: 10.48550/ARXIV.2007.01156. [Online]. Available: https://arxiv.org/abs/2007.01156.

[11] M. v. Steen and A. S. Tanenbaum, "Introduction", in *Distributed Systems*. Maarten van Steen, 2017.

[12] G. F. Coulouris, "Characterization of distributed systems", in *Distributed systems: Concepts and design*. Pearson Education, 2012.

[13] C. H. Papadimitriou, *Computational complexity*, 1994. [Online]. Available: https://www.pearson.com/us/higher-education/program/Papadimitriou-Computational-Complexity/PGM94583.html.

[14] L. Magnoni, *Modern messaging for distributed sytems*, Sep. 2014. [Online]. Available: https://iopscience.iop.org/article/10.1088/1742-6596/608/1/012038.

[15] P. Mell and T. Grance, *The nist definition of cloud computing*, Sep. 2011. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-145/final.

[16] J. Frankenfield, *How cloud computing works*, Jul. 2022. [Online]. Available: https://www.investopedia.com/terms/c/cloud-computing.asp.

[17] P. Srivastava and R. Khan, *A review paper on cloud computing*, Jun. 2018. [Online]. Available: https://www.researchgate.net/publication/326073288_A_Review_Paper_on_Cloud_Computing.

[18] J. McArthur, A. Chandrasekaran, T. Bittman, and T. Zimmerman, "Predicts 2021: Cloud and edge infrastructure", Sep. 2018. [Online]. Available: https://emtemp.gcom.cloud/ngw/globalassets/en/doc/documents/3889058-the-edge-completes-the-cloud-a-gartner-trend-insight-report.pdf.

[19] V. Gezer, J. Um, and M. Ruskowsvi, *An extensible edge computing: Definition requirements and enablers*, Nov. 2017. [Online]. Available: https://www.researchgate.net/publication/321134141_An_Extensible_Edge_Computing_Architecture_Definition_Requirements_and_Enablers.

[20] A. Wright and P. De Filippi, "Decentralized blockchain technology and the rise of lex cryptographia", Mar. 2015. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2580664.

[21] H. Kakavand, N. Kost De Sevres, and B. Chilton, "The blockchain revolution: An analysis of regulation and technology related to distributed ledger technologies", Oct. 2016. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2849251.

[22] G. Suciu, C. Nădrag, C. Istrate, A. Vulpe, M.-C. Ditu, and O. Subea, "Comparative analysis of distributed ledger technologies", 2018, pp. 370–373. DOI: 10.1109/GWS.2018.8686563.

[23] C. Majaski, "Distributed ledgers", Feb. 2022. [Online]. Available: https://www.investopedia.com/terms/d/distributed-ledgers.asp.

[24] Ledger, "What is blockchain?", May 2022. [Online]. Available: https://www.ledger.com/academy/blockchain/what-is-blockchain.

[25] A. Hayes, "Blockchain explained", Apr. 2022. [Online]. Available: https://www.investopedia.com/terms/b/blockchain.asp.

[26] P. Schueffel, "Alternative distributed ledger technologies blockchain vs. tangle vs. hashgraph - a high-level overview and comparison", Mar. 2018. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3144241.

[27] S. Seth, "Public, private, permissioned blockchains compared", Mar. 2022. [Online]. Available: https://www.investopedia.com/news/public-private-permissioned-blockchains-compared/.

[28] H. Fabric, "Introduction to hyperledger fabric 2.2", Jul. 2020. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.2/whatis.html.

[29] *Consensus algorithms in blockchain*, May 2022. [Online]. Available: https://www.geeksforgeeks.org/consensus-algorithms-in-blockchain/.

[30] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem", *ACM Transactions on Programming Languages and Systems*, pp. 382–401, Jul. 1982.

[31] D. Mingxiao, M. Xiaofeng, Z. Zhe, W. Xiangwei, and C. Qi06, "A review on consensus algorithm of blockchain", pp. 2567–2572, 2017. DOI: 10.1109/SMC.2017.8123011.

[32] J. Frankenfield, *Proof of work (pow)*, May 2022. [Online]. Available: https://www.investopedia.com/terms/p/proof-work.asp.

[33] E. Napoletano, *Proof of stake explained*, May 2022. [Online]. Available: https://www.forbes.com/advisor/investing/cryptocurrency/proof-of-stake.

[34] M. Castro and B. Liskov, "Practical byzantine fault tolerance", *OSDI*, Mar. 1999. [Online]. Available: https://pmg.csail.mit.edu/papers/osdi99.pdf.

[35] T. C. Devs, *An introduction to pbft consensus algorithm*, Oct. 2020. [Online]. Available: https://coredevs.medium.com/an-introduction-to-pbft-consensus-algorithm-11cbd90aaec.

[36] H. Fabric, "The ordering service", Jul. 2020. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.2/orderer/ordering_service.html.

[37] *Raft*. [Online]. Available: https://ebrary.net/206495/computer_science/raft.

[38] M. Hamilton, "Blockchain distributed ledger technology: An introduction and focus on smart contracts", *Journal of Corporate Accounting & Finance*, vol. 31, no. 2, pp. 7–12, 2020. DOI: https://doi.org/10.1002/jcaf.22421. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/jcaf.22421.

[39] *What are smart contracts on blockchain?* [Online]. Available: https://www.ibm.com/topics/smart-contracts.

[40] *Hyperledger fabric*. [Online]. Available: https://wiki.hyperledger.org/display/fabric/Hyperledger+Fabric.

[41] *Hyperledger fabric glossary*, Jun. 2021. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/latest/glossary.html.

[42] *Hyperledger fabric key concepts*, Jun. 2021. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/latest/key_concepts.html.

[43] *Hyperledger fabric transaction flow*, Nov. 2021. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/latest/txflow.html.

[44] *Hyperledger fabric: A simplified overview*. [Online]. Available: https://web.archive.org/web/20220718061019/https://www.oak-tree.tech/blog/hyperledger-overview.

[45] *Go - frequently asked questions (faq)*. [Online]. Available: https://go.dev/doc/faq.

[46] IBM, *Containerization*. [Online]. Available: https://www.ibm.com/cloud/learn/containerization.

[47] ——, *What is docker?* [Online]. Available: https://www.ibm.com/cloud/learn/docker.

[48] *What is a vpn? - virtual private network*, Feb. 2022. [Online]. Available: https://www.cisco.com/c/en/us/products/security/vpn-endpoint-security-clients/what-is-vpn.html.

[49] I. ZeroTier, *Protocol design whitepaper: Zerotier documentation*. [Online]. Available: https://docs.zerotier.com/zerotier/manual/.

[50] *Gossip data dissemination protocol*, Nov. 2021. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/latest/gossip.html.

[51] *Unix timestamp - epoch converter*. [Online]. Available: https://www.unixtimestamp.com/.

[52] T. Contributor, *What is daemon? - definition from whatis.com*, Sep. 2005. [Online]. Available: https://www.techtarget.com/whatis/definition/daemon.

[53] *What is a cron job? - scheduled tasks*, Mar. 2021. [Online]. Available: https://www.hivelocity.net/kb/what-is-cron-job/.