TU WIEN Informatics

Incentive Mechanisms in Fog Computing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Anja Suhih, BSc
Matrikelnummer 01529286

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Dr.-Ing. Stefan Schulte

Wien, 22. März 2022

Anja Suhih	Stefan Schulte

Informatics

Incentive Mechanisms in Fog Computing

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering & Internet Computing

by

Anja Suhih, BSc
Registration Number 01529286

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr.-Ing. Stefan Schulte

Vienna, 22nd March, 2022

Anja Suhih	Stefan Schulte

Erklärung zur Verfassung der Arbeit

Anja Suhih, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 22. März 2022

Anja Suhih

Acknowledgements

I would like to use this opportunity to thank my advisor, Professor Stefan Schulte, for a very pleasant cooperation and delightful communication throughout this last phase of my studies. His dedication and technical support, accompanied by speedy, constructive feedback have played a significant role in the process of writing my Master's thesis.

Further thanks are directed to my colleagues and friends, for the selfless note sharing, sometimes difficult, but always interesting discussions, and for always keeping me motivated – you managed to make even those all-nighters fun.

Last, but not least, I want to thank my family for the unconditional support and encouragement they provided, not only during the course of of my studies, but for the last 25 years.

Kurzfassung

Das Internet der Dinge (engl. Internet of Things, IoT) ist eines der am schnellsten wachsenden Paradigmen in der Informationstechnologie. Es umfasst eine Vielzahl physischer Geräte, die mit Sensor-, Kommunikations-, Netzwerk- und/oder Verarbeitungstechnologien ausgestattet sind, mit dem Ziel, sich mit anderen Geräten und Systemen über das Internet zu verbinden, um Daten auszutauschen. Obwohl diese Geräte für die alltäglichen Aufgaben des durchschnittlichen Benutzers recht praktisch sind, sind sie aufgrund der begrenzten Speicher- und Verarbeitungskapazitäten nicht leistungsfähig genug, um komplexe Aufgaben alleine auszuführen. Die fehlenden Ressourcen findet das IoT in einer symbiotischen Beziehung mit der Cloud. Die Cloud bietet praktisch unbegrenzte Speicher- und Verarbeitungsleistung und hat in Zusammenarbeit mit dem IoT bemerkenswerte Erfolge gezeigt. Allerdings nehmen die IoT-Anwendungsfälle und -Domänen ständig zu, so dass das Cloud-IoT-Paradigma allmählich an seine Grenzen stösst. Ein Bereich, in dem sich die Cloud als unzureichend erwiesen hat, sind Systeme, bei denen die Echtzeit-Datenverarbeitung eine entscheidende Rolle spielt. Die Cloud-Rechenzentren, die eine zentralisierte Datenverarbeitung bereitstellen, sind oft weit vom Endbenutzer entfernt, was zu einer hohen Zugriffslatenz führt. Hier ist die Fog eine sinnvolle Alternative. Fog Computing ist eine Erweiterung des Cloud Computings und bietet die erforderlichen Ressourcen physisch näher an den IoT-Geräten, am Rande des Netzwerks. Sie bietet zeitnahe Dienste für zeitkritische Probleme, Standortbewusstsein, geringe Latenz, ermöglicht die geografische Verteilung und Interaktionen in Echtzeit.

Die Fog bietet daher Lösungen für die Datenverarbeitung im IoT. Dies ist jedoch nur dann der Fall, wenn genügend Fog-Knoten bereitgestellt werden und zusammenarbeiten, um den besten Service zu liefern. Da die Fog noch ein sehr junges Paradigma ist, sind viele Aspekte noch nicht vollständig erforscht. Ein solches Konzept ist ein effektiver Anreizmechanismus, der Netzwerkknoten dazu anregt, ihre Ressourcen für den Rest des Netzwerks zur Verfügung zu stellen. Genau das ist das Hauptanliegen dieser Arbeit – wir schlagen einen nicht-negativen kreditbasierten belohnungsorientiert Anreizmechanismus vor, das als Ziel hat, das Knoten-Engagement im Fog-Netzwerk zu erhöhen.

Um dies zu erreichen, implementieren wir einen Algorithmus, der die teilnehmenden Knoten belohnt und sie für die Aufgabenausführung in Credits bezahlt. Je mehr Aufgaben ein Knoten ausführt, desto mehr Credits verdient er. Die erworbenen Credits spielen bei der Job-Übermittlung eine wichtige Rolle, wobei die Jobs der Knoten mit einem höheren Credit-Saldo priorisiert werden. An die Designphase schließt sich ein ausführlicher Umsetzungsprozess an, der mit einer detaillierten Evaluation abgeschlossen wird. Darin zeigen wir, dass die Knotenbeteiligung in Szenarien mit hoher Arbeitslast bis zu 100% erreichen kann. Dies bedeutet, dass alle Knoten zu beitragenden Mitgliedern des Systems werden.

Abstract

The Internet of Things (IoT) is one of the fastest-growing paradigms in information technology. It encompasses a wide variety of physical devices ("things"), embedded with sensory, communication, networking, and/or processing technologies, with the goal of connecting with other devices and systems over the Internet, to exchange data. These devices, although quite convenient for average-user daily tasks, are usually characterized by limited storage and processing abilities, and are not powerful enough to perform complex tasks on their own. The resources it lacks, IoT found in a symbiotic relationship with the Cloud. The Cloud offers virtually unlimited storage and processing power and has shown remarkable success in cooperation with the IoT. However, with continuously increasing use cases and application domains, the Cloud-IoT realm is starting to display some limitations. One such domain, where Cloud has proven insufficient, are systems where real-time data processing plays a crucial role. Cloud data centers, providing centralized data processing, are often located a long way from the end-user, causing high access latency. This is where the Fog becomes effective. Fog computing is an extension of Cloud computing, offering the necessary resources physically closer to the IoT devices, at the edge of the network. It provides timely services to time-sensitive issues, location awareness, low latency, enables geographical distribution and real-time interactions.

The Fog offers solutions to computing problems in the IoT. However, this is only the case when enough Fog nodes are deployed and work together to deliver the best service. Given that the Fog is still a very young paradigm, many aspects have not yet been fully explored. One such concept is an effective incentive mechanism that stimulates network nodes to make their resources available for the rest of the network. Precisely that is the main concern of this thesis – we propose a non-negative credit-based reward incentive mechanism, aiming to increase node engagement in the Fog network.

This is accomplished by implementing an algorithm to award the participating nodes, paying them for task execution in credits. The more tasks the node executes, the more credits it will earn. The acquired credits play an important role during job submission, prioritizing the jobs of the nodes with a higher credit balance. The design phase is followed by an elaborate implementation process and concluded with a detailed evaluation. In it, we show that in high workload scenarios, node involvement can reach up to 100%, meaning that all nodes become contributing members of the system.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The *Internet of Things (IoT)* has gained a lot of popularity in the last decade, with recent predictions forecasting that the total IoT market will reach more than one trillion U.S. dollars in 2030 [72]. The IoT can be defined as the pervasion of business and private spaces with "*a variety of things or objects ... [which] interact with each other and cooperate with their neighbors to reach common goals*" [11], forming "*an interconnected world-wide network based on sensory, communication, networking, and information processing technologies*" [45]. IoT devices are capable of collecting information from their surroundings by using different types of sensors. This data can then be shared with other IoT devices, forwarded to data stakeholders, or stored in the cloud. The usage of IoT technologies has been proposed for many different application areas, including smart healthcare, smart cities, and smart agriculture, to name just some examples [38]. Hassan et al. [35] present a taxonomy of IoT application fields, based on the work of recent studies. They declare health care, the environmental, smart cities, commercial, industrial, and infrastructural fields as main areas of IoT, each with additional subdomains.

The IoT enables devices to operate without human interaction, and quite often without human knowledge. According to Statista [73], the number of IoT-connected devices worldwide will nearly triple from 8.74 billion in 2020 to more than 25 billion IoT devices in 2030. Another report predicts that, "*by 2023, IoT devices will account for 50 percent of all networked devices (nearly a third will be wireless)*" [3].

The above-mentioned IoT devices are characterized by limited storage and processing capacities, which raises concerns regarding their reliability, performance, security, and privacy [55]. This is where *Cloud computing* comes into play as a major enabler of the IoT. Cloud computing offers virtually unlimited storage [71] and processing capabilities, and, at least partially, solves important IoT issues by expanding the available storage and by increasing the processing capabilities [17]. The National Institute of Standard and Technologies (NIST), in its definition, outlines the main aspects of Cloud computing as follows: "*Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction*" [52].

Accordingly, the Cloud, permitting an on-demand lease of nearly unlimited resources and storage, as well as uncostly processing capabilities, assists the IoT to compensate its technological constraints. At the same time, by extending its scope and dealing with real-world things in a more distributed manner, while delivering new services in a large number of real-life scenarios, the Cloud also benefits from the IoT. In many cases, the Cloud serves as an intermediary layer between IoT devices and (end-user) applications, obfuscating all of the complexities and functionalities required to incorporate the latter.

As already mentioned, Cloud computing helps to avoid some IoT limitations, but there are others, like mobility support, geo-distribution, location awareness, and low latency for which Cloud computing is not well-suited. Likewise, there are still some concerns about data security and user privacy in the Cloud [26]. For instance, there are several drawbacks in the relationship between highly geo-distributed IoT devices, and a centralized Cloud used for processing data coming from these devices. High link delays (latency), low data transfer speed, neglecting of computational and storage resources of IoT devices, security and privacy are often named the most critical ones [37].

The number of IoT applications that are delay-sensitive and rely on real-time data processing is increasing in many application scenarios [18]. That is why, the (drastic) reduction of latency of IoT applications is very significant, enabling real-time communication resulting in improved decision making. An example would be a smart transportation system, where each car generates massive amounts of data, not all of which needs to be sent to the Cloud. In this situation, it is of crucial importance that the IoT devices (such as cars) process real-time data and make correct decisions. Waiting for the data to be transferred to the Cloud, and then waiting for the results from the Cloud to implement the decision is out of the question in situations like these [74]. In fact, time-sensitive data should be processed in an edge computing architecture at the point of origin, or sent to an intermediary server located in close geographical proximity to the data source. This way, the car will make an informed decision on the fly to avoid potentially dangerous circumstances.

Seeing that, as previously noted, IoT devices do often not have the capacity to do (heavyweight) computations themselves, another layer between the Cloud and the IoT devices is needed. This is where *Fog computing* enters the picture. The Fog offers solutions to the discussed problems by extending services and resources offered by the Cloud to the edge of the network, and therefore closer to the end devices [54]. An overview of a typical Fog scenario can be found in Figure 1.1. The OpenFog Consortium defines Fog computing as "*a horizontal, system-level architecture that distributes computing, storage, control and networking functions closer to the users along a cloud-to-thing continuum*" [58].

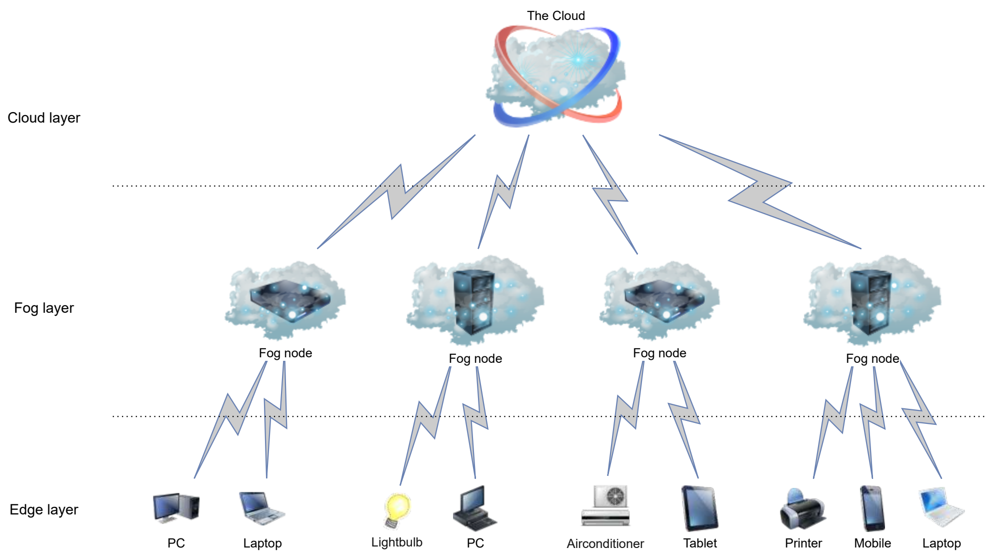

Figure 1.1: Overview of Fog Computing

The Fog extends the Cloud, bringing it closer to the devices that generate the data and act on it, potentially right up to the sensors and actuators of IoT devices. Cisco's white paper [2] gives a pretty broad definition of what a *fog node* is, saying that "*Any device with computing, storage, and network connectivity can be a fog node*". Yi et al. [86] deliver a slightly more detailed definition, describing a Fog node as any facility or infrastructure that is able of providing resources for services at the edge of the network.

Both Fog and Cloud computing are built on compute, storage, and networking resources. The Fog however offers many characteristics that make it a non-trivial extension of the Cloud, such as edge location, location awareness, low latency, geographical distribution, support for mobility, real-time interactions, and device heterogeneity [16].

In addition to the already mentioned benefits, Fog computing may also provide enhanced privacy to the end-user. In Cloud computing, personal data has to travel through the whole network to be collected and stored in a central place, while Fog applications can keep personal data at the edge, transferring only essential (anonymized or encrypted) data to the cloud.

Considering the significant improvements Fog computing provides, especially in certain domains, it should be in our interest that more nodes join Fog computing networks. The nodes in the Fog network can share their idle resources and collaboratively accomplish computing tasks [89], meaning, the more nodes there are in the network, the better the overall performance will be. Unfortunately, the nodes in that network are not aware of this. In real networks, these nodes can be selfish, and unwilling to communicate and cooperate with other nodes [44]. They need to experience that they are gaining something in order to participate, i.e. they need to be properly motivated. One way of motivating the nodes to cooperate is by offering them an appealing *incentive mechanism*. This is exactly what we will deal with in this thesis. Since there has not been many research published on this topic, we will conceptualize and implement an incentive mechanism that would make sure more nodes join a Fog network.

Researching, and analyzing existing, and implementing a pertinent and engaging incentive mechanism in Fog computing is the main goal of this thesis.

1.2 Aim of Work and Methodology

An *incentive mechanism* can be defined as "*a treatment or measure to motivate and encourage people (i.e., to participate in a learning network)*" [19]. However, incentive mechanisms are not constrained only to people or learning networks. This definition can be extended so that it can be applied to computer networks as well. If we are talking about Fog computing, our goal would be to motivate and encourage nodes to join a Fog network. Each node operating inside a network helps to improve the system, and contributes to the success of the whole network. As previously mentioned, the nodes are not always willing to contribute to a certain network. We need to find a way to persuade them.

Precisely that is the objective of this thesis – to find an incentive mechanism whose results would be attractive for the network, motivating nodes to be a contributive part of it.

Today, there is not enough research published that addresses this issue in Fog computing or even in the IoT. Incentive mechanisms in the area of Social Computing [67], Peer-to-Peer (P2P) systems [41, 64], Crowdsensing [83], etc., as well as a more general approach, the impact of incentive mechanisms on project performance [53], have been analyzed, but the research is lacking in the Fog computing domain. As already explained in the previous section, Fog computing brings significant improvements to the constantly-increasing world of the IoT. Thus, the goal of this thesis is to conceptualize and implement an appropriate incentive mechanism for Fog computing, using the already mentioned mechanisms from other fields and adapting them to the Fog.

In order to achieve this, the work in this thesis will be divided into the following phases:

1. **Literature research.** The first step is a thorough research of existing incentive mechanisms. A quick investigation done so far shows that a detailed discussion on incentive mechanisms in Fog computing has not been published yet. This examination will be deepened and extended during the work on this thesis. Assuming no such mechanisms have been proposed in the field of Fog computing, we will have to research and analyze incentive mechanisms from different domains, including, but not limited to, the above mentioned fields, that could potentially be transferred to Fog networks.

2. **Assessment.** Those mechanisms will then be assessed and evaluated. The parameters which will be used in the assessment will be derived from the needs of

Fog networks. During this phase, the attention will be turned to the mechanisms that would be advantageous in Fog computing network, i.e., mechanisms that could be modified and/or extended to fit to Fog networks. Mechanisms for which no application can be found in Fog networks will be discarded. Another major step during this stage is to narrow down the focus to a single incentive mechanism. From the previous analysis, one mechanism that could potentially provide (the most) beneficial results will be chosen.

3. **Design and Implementation.** The chosen mechanism will most likely have to be modified and/or adapted to be implemented as an incentive mechanism in Fog computing. From the information acquired in the previous steps, an incentive mechanism will be conceptualized. This stage also covers the design of that mechanism. After this has been determined and documented, a prototype of the mechanism will be implemented.

4. **Evaluation.** The final step is the evaluation. This phase will start by using a software that simulates a Fog network. During this step, it will be observed, if the implemented incentive mechanism has any, and if yes, what kind of positive and negative influence on the nodes in the network. There are a couple of potential approaches to the evaluation: For instance, the individual benefits of a single node can be monitored, and/or the benefits of the network as a whole can be observed. Cost, available resources, and/or latency will be considered as viable comparison metrics.

1.3 Structure of the Thesis

Following the methodology described in the previous section, the remainder of the work will be divided into six chapters. Chapter 2 provides background on the most important concepts and technologies needed for the further understanding of the thesis, such as IoT, the Fog, and incentive mechanisms. Chapter 3 presents some research papers, from different domains that deal with this topic, and whose discoveries could be beneficial for finding an appropriate incentive mechanism for the Fog. A new/modified incentive mechanism suitable for the Fog network is presented in the Chapter 4, whose implementation follows in Chapter 5. Chapter 6 deals with the evaluation of the implemented incentive mechanism. This is where the mentioned metrics will be analyzed and compared to see what effect did the proposed mechanism have on the nodes in the network. Chapter 7 brings the conclusion, as well as some possible future work.

CHAPTER 2

Background

2.1 Internet of Things

2.1.1 History

The concept of IoT has first been introduced in 1999 by Kevin Ashton, as a title of a presentation he held, when he linked the new idea of Radio Frequency Identification (RFID) technology to a company's supply chain [8]. He described the IoT as a network of physical objects (things) that have sensors and are connected to the Internet. Today, 23 years later, an exact (or a common, for that matter) definition of IoT is still not coined. Different researchers and organizations provide different definitions, depending on the perspective taken. The core concept is nonetheless clear – everyday objects are able to communicate with one another over the Internet to achieve a common goal, by equipping them with identifying, sensing, network and processing capabilities [80].

As previously mentioned, the initiator of IoT is RFID technology, that enabled devices to function without human interaction. This technology has first been applied during the Second World War, to identify friendly aircraft [23]. In the early 2000s, companies realized that by attaching RFID tags to products in the initial stages of manufacturing, they can drastically reduce supply chain costs [10]. Manpower needed for tracking and monitoring the product could be replaced with a simple tag of negligible cost. The tag follows the product down the supply chain, to the retail stores, all the way to the customer. The purpose of the tag varies, depending on the stage in the supply chain. At first, it can be used for tracking and/or environmental condition checking. Once it reaches the stores, the tag could serve as a price tag, while, once purchased by the customer, it can be used as a warranty information source.

Since then, IoT has advanced, making everyday devices, such as smoke detectors, window blinds, home appliances etc., *smart* by attaching technology, making them directly accessible (from other devices or people) via the Internet. In this system, devices perceive for themselves and respond faster and better than humans would, using analytics and business intelligence. This happens without human interaction, and quite often without human awareness.

Wide application, constantly reducing cost of devices, higher availability and continuous advancements in sensor technology embedded in IoT devices, created a growing interest in IoT.

2.1.2 Application

One of the main purposes of IoT devices is to help improve (the quality of) our lives. As such, they found application in many areas. There are examples of IoT systems all around us, which according to Hassan et al. [35] can be classified into six categories: healthcare, environmental, smart cities, commercial, industrial, and infrastructure. The most commonly-used device in healthcare is some kind of a wearable used for tracking, and monitoring of certain factors. Another option could be biosensors that are attached to the body, for personal remote health monitoring. Smart farming, smart agriculture, climate changes monitoring that use smart device, Wireless Sensor Network (WSN), or smart home system are further examples from the environmental domain. Smart cities include smart homes, smart buildings, urban computing traffic, monitoring security and emergencies. Digital forensics using smartphones and computers, Big Data processing with MapReduce devices, real-time low power routing protocol with WSN devices are some of the potential focus areas in this domain. Hassan et al. propose shopping systems and retail that use smart devices and IoT sensors as sub-domains of commercial applications. Smart grid and smart metering employing (industrial) sensors and mobile devices are mentioned as part of the industrial application. The infrastructural domain focuses on real-time performance and energy-efficient utilization of smart devices, wearable devices, smart sensors etc.

An example of the application domains of IoT is shown in the Figure 2.1.

2.1.3 Example Scenario

The previously mentioned example of smart transportation (Fig. 2.2) (Section 1.1) is a good example of how the IoT helps to facilitate our lives, improving liveability, workability and sustainability [57]. These connected vehicles, that are able to "communicate" with the environment and to transmit data, can bring great changes to the field of autonomous driving, making our everyday commute faster and more comfortable. Parking in a busy city can be troubling. Enabling the vehicle to interact with its surroundings can make this experience much easier, providing live data on parking spaces, and automating the parking process. This concept is called Vehicle-to-Everything (V2X) [42]. Based on the data collected from their surroundings, vehicles can make informed decisions, considering every aspect of traffic at the same time. According to the European Telecommunications Standards Institute (ETSI) [4], there are four types of V2X, i.e., four categories of connectivity based on IoT in vehicles:

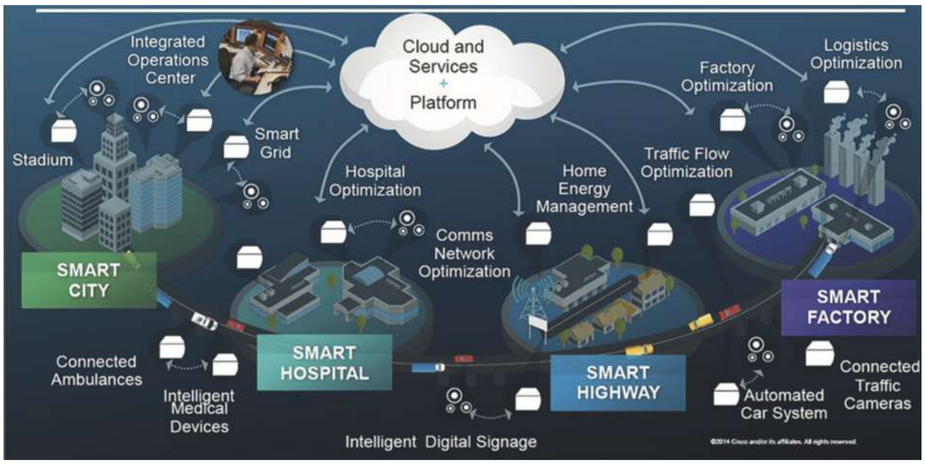

Figure 2.1: Applications of IoT [10]

- Vehicle-to-Infrastructure (V2I) - Interaction between vehicles and external objects
- Vehicle-to-Network (V2N) - Interaction between vehicles and V2X
- Vehicle-to-Pedestrian (V2P) - Interaction between vehicles and a vulnerable road user, such as pedestrian or cyclist
- Vehicle-to-Vehicle (V2V) - Interaction between vehicles

In order to achieve a safe driving experience, it is important that a vehicle communicates with all traffic participates, and to do that in a timely manner. Since most of the traffic participants are constantly moving, real-time data processing is crucial to avoid potentially dangerous scenarios. Satellite-based Global Positioning System (GPS) is used for location determination, while built-in sensors detect physical objects in the surrounding.

2.1.4 Architecture

To put it simply, IoT is the concept of connecting devices with the Internet or each other. Using different kind of sensors, IoT devices can produce and transmit data in real-time, that can further be stored/analyzed/processed/shared. IoT devices that can capture streaming data for rapid analysis and perform immediate actions or processing of the data, are called *edge devices* [34].

Service-Oriented Architecture (SoA), implemented in IoT, ensures interoperability among the heterogeneous devices [59]. In [45] a four-layer SoA is presented, with the following layers:

- **Sensing layer** - integrated in the IoT devices, and used to collect information from the environment

Figure 2.2: Smart transport [51]

- **Network layer** - the infrastructure that enables wireless or wired connection among IoT devices
- **Service layer** - creates and manages services required by users/applications
- **Interfaces layer** - methods for application interaction

Sensing Layer

The interconnected world-wide network formed by the IoT devices enables them to communicate and to be controlled remotely. At the sensing layer, the focus is on usage of tags and sensors to automatically sense (the changes in) the environment and share this information among devices. The RFID tags or barcode readers and sensors are wireless devices responsible for collecting raw data. Essentially, they make the devices "things" of an IoT system. All these objects in an IoT system can be uniquely identified and tracked in the digital domain. Main functions of the sensing layer are to sense, actuate, identify, interact and communicate.

Network Layer

The main task of the network layer is to connect all the things, routing the data from the sensor to either other devices, or the next layer. It plays the main role in sharing the data with the connected devices. The network layer also provides aggregating functionalities from Information Technology (IT) infrastructures that can transmit the data to decision-making units. A precondition is to have large storage capabilities to store massive amounts of data received each second.

Service Layer

All service-oriented activities are performed at the service layer. Such activities include information exchange, storage and analysis, device analysis, data management, data mining etc. This layer can also be observed as a management service layer [10] since its purpose is to filter through and extract important in formations, from the immense raw data collected. The services in the service layer run directly in the network to efficiently locate new services for an application.

Interfaces Layer

This layer is responsible for the utilization of the data collected by the sensors. Displaying the data to the user, and letting users interact with the results (filtering, aggregation, graphical representation etc.) via, e.g., a smartphone is the main task of this layer.

In our smart transportation system example, sensors are continuously sensing the movement of the traffic participants (sensing layer), and sending data through wireless communication to the database (network layer). This data is then filtered, processed and analyzed (service layer) and the result, i.e., the relevant data, is sent back to the edge devices, in this case vehicles, where they are presented to the user, and considered by the traffic participants for the next steps (interfaces layer).

2.2 Cloud Computing

IoT devices, although very useful in our daily lives, do not have the ability to perform complex tasks on their own. They are characterized by limited storage and processing capabilities that raise concerns regarding their reliability, performance, security, and privacy [55]. This is where the virtually unlimited storage and processing capabilities offered by the Cloud computing prove useful, solving many IoT problems at least partially. Low cost of practically infinite storage [71] and processing capabilities permitted a new computation model, which offers virtualized resources to be leased in an on-demand fashion.

2.2.1 Cloud and IoT

The complementary characteristics of IoT and the Cloud, presented in Table 2.1, play a significant role in the integration of the two. Many researchers saw a possibility to gain great benefits in specific application scenarios by combining them [6, 7, 31], creating a paradigm called *CloudIoT* [17, 13].

The relationship between Cloud and IoT is a symbiotic one. IoT can use the resources it lacks, and Cloud offers. Simultaneously, Cloud can broaden its scope to deal with real-world objects in a more distributed and dynamic manner, and deliver new services in a variety of real-world scenarios. In simplest terms, more often than not, Cloud can

Table 2.1: Complementary aspects of Cloud and IoT [17]

	IoT	**Cloud**
Displacement	pervasive	centralized
Reachability	limited	ubiquitous
Components	real-world things	virtual resources
Computational capabilities	limited	virtually unlimited
Storage	limited to none	virtually unlimited
Role of the Internet	point of convergence	means for delivering service
Big Data	source	means to manage

provide the intermediate layer between the things and the applications, hiding all the complexity and functionality.

Researches have categorised the motivation driving the integration of the Cloud and IoT, i.e., *drivers*, into three main categories: *communication*, *storage* and *computation* [17].

Communication. The two main aspects of the communication category are data and application sharing. The CloudIoT enables the delivery of personalised ubiquitous applications through the IoT, and, at the same time, applies automation to both data collection and distribution at low cost. Using customizable portals and built-in apps, the Cloud provides a cost-effective and efficient way to connect, track, and manage anything from anywhere at any time [14]. High-speed networks enable effective monitoring and control of remote items [14, 28, 61], their location [28, 61], their communications [28], and real-time access to the data produced [14].

It should also be considered, that even though Cloud can significantly improve communication in IoT, it can also represent a bottleneck in some scenarios – limitations can be encountered when transferring huge amounts of data from edge devices to the Cloud.

Storage. Large numbers of the things making up the IoT network produce an enormous amount of data every minute. In 2019, Cisco predicted that by 2021, 850 Zettabytes (ZB)[1] will be generated by all people, machines, and things. One of the most important drivers for CloudIoT is certainly the large-scale and long-lived, low-cost, on-demand storage provided by the Cloud. As such, Cloud offers opportunities for data aggregation [28], integration [88] and sharing with third parties [88]. Once it reaches the Cloud, data can be analyzed, protected, visualized, stored for later processing etc.

Computation. The (small) size of the edge devices results in reduced processing and energy resources which prevents complex, on-site data analysis and/or processing. The collected data is transmitted to more powerful nodes (the Cloud) where data manipulation can happen.

[1]one zettabytes = one trillion gigabytes

2.2.2 Architecture

The Cloud computing architecture follows a layered computing model with four layers [90]:

- **Hardware layer**
  This layer is in charge of the Cloud's physical resources, such as physical servers, routers, switches, power, and cooling systems, and is usually implemented in the data centers.

- **Infrastructure layer**
  The infrastructure layer, also known as the virtualization layer, partitions physical resources using virtualization technologies like VMware [1] to create a pool of storage and computing resources.

- **Platform layer**
  The platform layer consists of operating systems and application frameworks, building on top of infrastructure layer. Minimizing the burden of deploying applications directly into Virtual Machine (VM) containers is the main goal of this layer.

- **Application layer**
  At the top of the hierarchy is the application layer, consisting of the actual Cloud applications.

Types of Cloud

In the literature [17, 90], the following types of Cloud have been identified: *Public Clouds* intended for the open use by the public. Services are made available to organizations and users over a public network through a browser. They are location independent, reliable and highly scalable, but less secure and not customizable. *Private Clouds* intended for an exclusive use of an organization, group or individuals. It is owned, managed and operated by that organisation/group/individual. They have limited scalability and are restricted to an area, however they can be customized to fully fit one's needs. *Community Clouds* are usually shared by more organisations that have a common interest. An example would be universities, that use them for learning and research. *Hybrid Clouds* present a combination of different Cloud types, with core activities hosted on a private Cloud, while other are outsourced to a public Cloud or a community Cloud.

2.3 Fog Computing

It is undeniable that Cloud services have clear advantages, nevertheless, they also have a major drawback: Cloud data centers are centralized and, as a result, are often located distant from the end user, resulting in high access latency. While this is sufficient for many application domains such as enterprise or Web applications, some more modern application areas require additional properties [15]. Our smart transportation system is such an example. Such applications are usually deployed on edge devices. In that case, edge devices provide low access latency due to the physical proximity, but now we have the problem of limited resources and processing capabilities, which is intolerable in the mentioned scenario. This is where *Fog computing* comes into play, allowing us to achieve low latency granted by the edge devices, while at the same time having access to infinite resources offered by the Cloud [15].

Fog computing is an extension of Cloud computing, offering Cloud resources closer to the devices, i.e., at the edge of the network. It serves as a layer between the Cloud and the underlying edge devices, solving issues like mobility support, geo-distribution, location awareness and low latency, that Cloud faces [26]. Fog enables real-time data distribution, which is particularly useful for time-sensitive services, such as healthcare or transportation related topics [5]. It can also provide preparatory smart actions, before notifying or transferring data further to the Cloud. Another intended use of the Fog network is the temporary data processing, which does not require data to be sent to the Cloud to do a simple, temporary task, but can be done easily, efficiently, and quickly in the Fog network.

In November 2015, ARM, Cisco, Dell, Intel, Microsoft and Princeton University Edge Computing Laboratory founded OpenFog Consortium to stimulate interest in Fog computing and its development [58]. The name, *Fog*, comes from the vizualization that fog describes the clouds close to the ground, relating to the clouds up in the sky in the Cloud network (see Fig. 1.1).

Fog computing is a valuable addition to Cloud computing, not a replacement. It allows for edge processing for certain application components (such as latency-sensitive ones), while retaining the ability to interact with the Cloud (for, e.g., delay-tolerant and computational intensive components) [54]. The Fog can not operate in standalone mode. It plays a significant role in the areas where Cloud computing meets its limitations such as latency-sensitive scenarios. Connected vehicles (i.e., transportation system) [74], fire detection and firefighting [84], smart grid [74], and content delivery [81] are some examples for such domains. Privacy is another issue for which Fog offers a better solution. In Cloud applications, personal data is collected in a central place. As opposed to Fog application that keep personal data at the edge, forwarding only aggregated and properly anonymized/protected data to the Cloud.

Edge vs. Fog Computing

A term often used interchangeably with Fog computing is *Edge computing*. If that is correct depends strongly on the source. Some researches say that they both refer to the same concept [68], while others argue that they are in fact two different notions [69]. Edge computing is concerned with the computation done at the edge of the network, without any Cloud service. Fog computing is either the same as Edge computing, or is defined as a combination of Cloud, edge, and any intermediate nodes [15]. In this paper, we will take the later to be true.

2.3.1 Characteristics

Compute, storage, and networking resources present the building blocks of both the Cloud and the Fog. Here, we will discuss the characteristics that make Fog a non-trivial extension of the Cloud [16, 85]:

- Location awareness: Location awareness, in which nodes can be deployed in different locations, is supported in Fog computing. In a Fog network, the nodes are aware of their location and can use this information for further processing. The information is extremely beneficial in some applications, such as our smart transportation system. Devices like smart traffic lights, vehicles and other traffic participants, heavily rely on these values, since the information required by these nodes depends on their location. Obtaining this information, for mobile and geographically distributed devices, can be difficult in the Cloud, since Cloud offers more global and centralized services [75].
- Low latency: In order for data from edge devices to be processed it has to be transferred to the few and far located Cloud data centers. Even though this process is relatively fast, for some applications this is not sufficient (e.g., smart transportation system, health care etc.). Fog, offering resources closer to the edge, ensures lower latency for such critical domains.
- Geographical distribution: In contrast to the centralized Cloud, the services and applications in Fog network benefit from/require widely distributed deployments, since they are not stationary, but are mobile. An example would be streaming to vehicles, through access points along tracks.
- Real-time interactions: Instead of using the batch processing utilized by the Cloud, Fog applications employ real-time interactions between fog nodes.
- Heterogeneity: Fog nodes, or end devices come from different manufactures, and have different specifications (routers, switches, access points, user devices etc.). The Fog is capable of supporting and managing all these in a uniform and consistent way, while working on different platforms these devices need to be deployed to.
- Scalability/Flexibility: The Fog provides distributed and dynamically allocated processing and storage capabilities, in order to meet the constantly changing requirements of the network.
- Large number of nodes: A side effect of having the nodes distributed across the network is a very large number of nodes. Data from all these nodes has to be collected and analyzed.
- Interoperability: Certain services, such as streaming, necessitate the collaboration of several service providers. Hence, Fog nodes need to be able to interoperate with different domains and across different providers.

- Cloud support: Processes that require low latency are done in the Fog network. Some data, however, might need to be saved in a more permanent storage or demands more complex data processing, in which case data is transferred to Cloud data centers.

2.3.2 Example Scenario cont.

Let's go back to our example scenario from Section 2.1.2. In V2X, each traffic participant is an IoT device, producing massive amount of data (i.e., a stream of data). At the same time, they connect to other traffic participants, trying to establish data-synchronization to provide safe and comfortable travels. In order to archive that, it is crucial that the data from moving objects is controlled in real time. Transmitting the (dynamically generated) data, from all devices in such a system, to the Cloud, waiting for them to be processed, and sending the result back to the end devices in real time is challenging, to say the least. Besides demanding low latency, these scenarios require that the location is taken into account when making important decisions, i.e., these devices need to know how close other participants in traffic are (e.g., distance from the vehicle to pedestrian). This is where Fog computing offers satisfactory solutions. Fog nodes can be used to execute processes that do not need processing that is too complex, but demand fast responses that take mobility into consideration. Tasks that require more computational power, or should be stored permanently are forwarded to the Cloud [75].

2.3.3 Fog Nodes

Cisco describes a fog node as a "mini Cloud", placed at the edge of the network, that is implemented through a range of interconnected edge devices [48]. The OpenFog Reference Architecture (RA) paper [58] states that the computational, networking, storage and acceleration elements of the new model, in which computation is moved closed to the ground, possibly right up to the IoT sensors, are known as fog nodes. These nodes are not entirely fixed to the physical edge, but rather should be viewed as a fluid network of connectivity. The nodes in a Fog system form a grid to provide load balancing, resilience, fault tolerance and minimization of Cloud communication [22]. They use two modes for communication: laterally (peer-to-peer, east to west) and up and down (north to south), and are able to discover, trust, and make use of service offered by other nodes, in order to sustain reliability-availability-serviceability [22]. A fog node can be any device with computing, storage and network connectivity. Examples would be industrial controllers, switches, routers, video surveillance cameras etc.

2.3.4 Architecture

The Fog bridges the gap between the Cloud and the IoT devices to enable a service continuum [21], creating a new opportunity for services, called *Fog-as-a-Service* [9]. In such a system, a service provider establishes a collection of nodes (tenants) across its geographic footprint and acts as their landlord. These nodes each have computation,

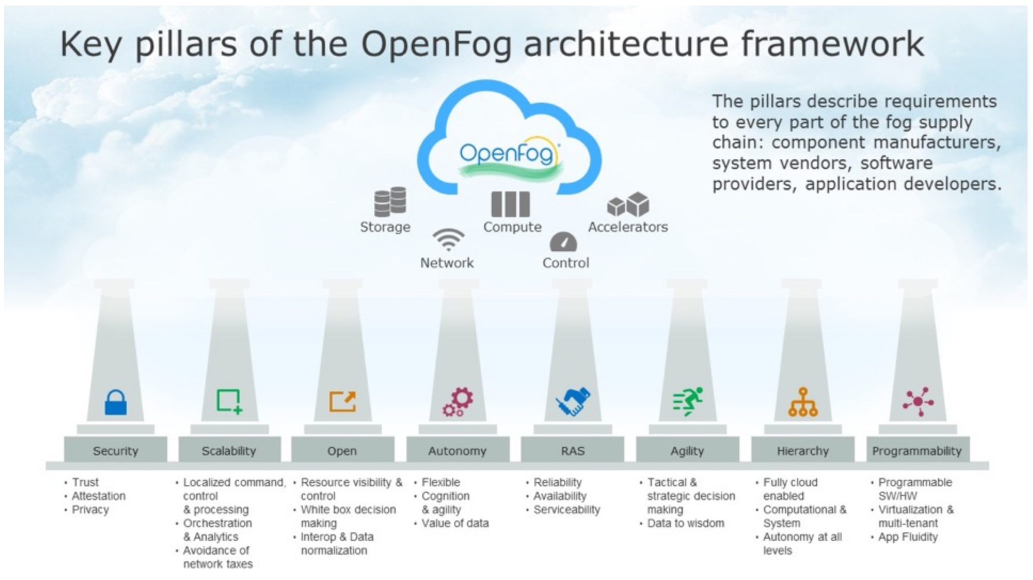
Key pillars of the OpenFog architecture framework
The pillars describe requirements to every part of the fog supply chain: component manufacturers, system vendors, software providers, application developers.
OpenFog
Storage
Compute
Accelerators
Network
Control
Security
• Trust
• Attestation
• Privacy
Scalability
• Localized command, control & processing
• Orchestration & Analytics
• Avoidance of network taxes
Open
• Resource visibility & control
• White box decision making
• Interop & Data normalization
Autonomy
• Flexible
• Cognition & agility
• Value of data
RAS
• Reliability
• Availability
• Serviceability
Agility
• Tactical & strategic decision making
• Data to wisdom
Hierarchy
• Fully cloud enabled
• Computational & System
• Autonomy at all levels
Programmability
• Programmable SW/HW
• Virtualization & multi-tenant
• App Fluidity

Figure 2.3: Pillars of Fog Computing [58]

networking and storage capabilities, that can, in comparison to expensive Clouds, be deployed and operated by small companies at different scales, depending on the needs of their customers [21].

A mobile phone, for example, might serve as a fog node for wearable devices, providing local control and analytics. While driving, the vehicle can act as a fog node for the user's phone, allowing various smartphone features to be shifted to the vehicle, such as display, user interface, audio, and phone book. For the moving vehicles, roadside unit can adopt the role of a fog node.

The architecture of Fog computing, according to the OpenFog Consortium, is driven by the eight core principles called pillars (Fig. 2.3) [58]. These pillars define the criteria for component manufacturers, system vendors, software providers, and application developers in the fog supply chain.

OpenFog presents the Fog architecture description as seen in the Figure 2.4. The OpenFog RA description is a composite of viewpoints of various stakeholder and perspectives that are utilized to meet the needs of a specific fog computing deployment or scenario [58].

Software View presented in the top three layers shown in the architecture description, includes Application Services, Application Support, and Node Management & Software Backplane.

System View presented in the middle layers shown in the architecture description, includes everything from Hardware Platform Infrastructure up to Hardware Virtualization.

Node View presented in the bottom layer shown in the architecture description, includes Protocol Abstraction Layer and Sensors, Actuators, and Control.

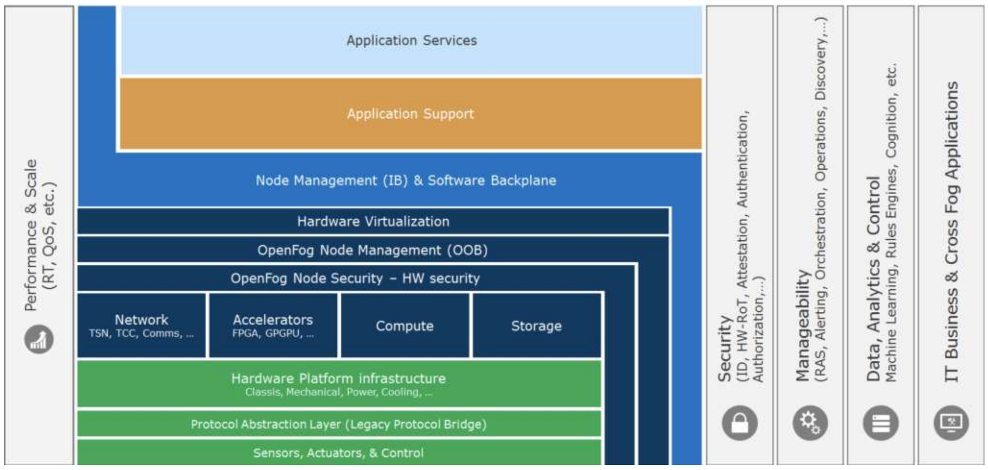
Application Services
Application Support
Node Management (IB) & Software Backplane
Hardware Virtualization
OpenFog Node Management (OOB)
OpenFog Node Security – HW security
Network
TSN, TCC, Comms, ...
Accelerators
FPGA, GPGPU, ...
Compute
Storage
Hardware Platform infrastructure
Classis, Mechanical, Power, Cooling, ...
Protocol Abstraction Layer (Legacy Protocol Bridge)
Sensors, Actuators, & Control
Performance & Scale (RT, QoS, etc.)
Security (ID, HW-RoT, Attestation, Authentication, Authorization,...)
Manageability (RAS, Alerting, Orchestration, Operations, Discovery,...)
Data, Analytics & Control Machine Learning, Rules Engines, Cognition, etc.
IT Business & Cross Fog Applications

Figure 2.4: Fog Computing Reference Architecture [58]

Besides the architecture of the Fog system itself, there are five cross-cutting concerns in the Fog, which apply to all the layers of the architecture. They are Performance, Security, Manageability, Data Analytics and Control, and IT Business and Cross Fog Applications [58] (Fig. 2.4).

CHAPTER 3

State of the Art

3.1 Incentive Mechanisms

NIST defines an incentive mechanism as "*a means of providing blockchain network users an award for activities within the blockchain network*" [82]. Another definition says an incentive mechanism is "*a treatment or measure to motivate and encourage people (i.e., to participate in a learning network)*" [19]. These definitions that speak of specific cases (a blockchain network, a learning network, people), can indeed be extended to include other networks and other participants. Many researchers and organizations have dealt with this topic, in various domains: from the impact of incentive mechanisms on project performance [53], to incentives in Social computing [66, 67], P2P systems [41, 64], Crowdsensing [83], etc. The questions such as what incentive mechanism is chosen, how it is implemented and what results does it give, all provide very relative answers, depending on the concrete use case. Meaning, that a mechanism that provides the best results in P2P systems might not be suited for Fog computing at all.

Finding an incentive mechanism that would deliver the greatest outcomes for Fog networks, engaging more nodes to participate in the network activities, is the purpose of this thesis. To achieve this, papers examining incentive mechanisms from many fields, including, but not limited to the above-mentioned domains, are analyzed, and a single mechanism that could potentially be the most effective is chosen, and adapted to fit the Fog's needs.

A brief overview of feasible methods is provided in this chapter.

3.2 Related Work

3.2.1 Project Performance

As already mentioned, incentive mechanisms can be applied to any area in life. Research with a more general approach, presented by Meng and Gallagher [53], analyzes how incentive mechanisms work, whether the use of incentives has a significant influence on project performance, as well as which incentive approaches are more effective. With the help of a questionnaire and a case study, incentives were proven to be a powerful tool for promoting best practices and ensuring project success. As a first step, they examined four payment methods in terms of their impact on cost performance. Next, they compared time and quality performance in incentive vs. non-incentive projects. Lastly, time and quality performances were compared in projects using single incentive mechanism vs. the ones where multiple incentive mechanisms were used. Concluding, a case study, examining how to appropriately apply incentives in practice was conducted.

The survey reported four types of incentives: time, cost, quality and safety, as well as some disincentives[1], either separately or combined with incentives. Cost incentives were usually incorporated with the payment method, while time and quality incentives cover both incentives and disincentives.

The case study conducted was a complement to the survey, and lead to the conclusion that the use of multiple incentives is complicated to manage but nevertheless their usage can improve the overall performance of a project, if project members are willing to put in extra work. Another significant discovery says that multiple incentive mechanisms can help to increase project performance overall, whereas a single incentive may be more successful in a specific performance field. They also deduce that incentives should be used together with disincentives to provide a more positive effect on project performance.

3.2.2 Data Acquisition and Distributed Computing

In the field of smartphone collaboration, Duan et al. [25] suggest an incentive mechanism for a client to motivate the collaboration of smartphone users on both data acquisition and distributed computing applications. For data acquisition applications, they introduce a reward-based collaboration mechanism, in which the interaction happens in two stages. In Stage I, the client announces the total reward to be shared among collaborators, as well as the the minimum number of collaborators needed. During Stage II, users individually choose whether to be a collaborator or not. They demonstrate that if the client is aware of the users' cooperation costs, they may choose to include just those with the lowest costs by offering a small total reward. However, if users can hold their private cost information from the client, the client needs to offer a larger reward to get enough collaborators.

To achieve collaboration in distributed computing, contract theory is used to study how a client decides on different task-reward combinations for many different types of users.

3.2.3 Crowdsourcing

Crowdsourcing is a business model where tasks are accomplished by the general public, i.e., the crowd. It is an online practice that uses the crowd's aggregate abilities and skills

[1]Incentives are defined as a reward and disincentives as a penalty.

to achieve specific goals. Focusing on the fact that the success of Crowdsourcing systems relies on the level of collaboration of the users, Kattmada et al. have written a paper [40], that gives an overview of user motives and incentives, and present appropriate incentive mechanisms to trigger these. They identified *(i)* learning/personal achievement, *(ii)* altruism, *(iii)* enjoyment/intellectual curiosity, *(iv)* social motives, *(v)* self-marketing, *(vi)* implicit work, and *(vii)* direct compensation as motives relevant to the Crowdsourcing environment. They explain that each motive can be triggered by one or more incentives. For example, access to the knowledge and feedback of experts or peers would be a suitable incentive for learning (*i*). Social motives (*iv*) could be activated by the will to attain social status and/or respect by organizers and peers, as well as the desire to present a good social image.

In their paper, they classified the incentive mechanism into four categories: reputation systems, gamification, social incentive mechanisms, and financial rewards and career opportunities (Fig. 3.1). The middle of the circle is populated by the user motives in different colors. In the outer circles, with matching colors, are incentives suitable for each motive presented. They are mapped to the incentive mechanisms that sustain them, shown in the image's four corners. Examples of the Crowdsourcing (CS) platforms are positioned according to the incentive mechanism they implement. As an example, Reddit is located between Social Incentive Mechanisms and Reputation system, because it incorporates both social elements and features a reputation system. They further describe each of the categories, along with providing recommendation on their careful design.

3.2.4 Crowdsensing

Yang et al. use smartphones, which hold sensing (e.g., accelerometer, compass, GPS, microphone etc.), collecting, and analyzing capabilities, as the base of their Crowdsensing research [83]. In it, they describe a new parading, Crowdsensing, as a wireless network of millions of personal smartphones exploited to sense, collect, and analyze data of human activities and surrounding environments, without the need to deploy thousands of static sensors. Since participation in Crowdsensing tasks has some drawbacks for the users (such as battery and computation power usage, as well as potential privacy threads), a satisfying compensation must be provided in exchange for their involvment.

The authors describe two types of incentive mechanisms for a crowdsensing system: A crowdsourcer-centric model and a user-centric model. In the former, the crowdsourcer possesses the absolute control over the payment, and users can only adapt their activities to the crowdsourcer's request. In the latter, the roles are reversed, meaning users have more control over the payment they will receive. A user proclaims the lowest price at which it is willing to sell a service, after which the crowdsourcer selects a subset of users, and pays them an amount that is no lower than the user's declared price. For the crowdsourcer-centric model, an incentive mechanism is designed that uses a Stackelberg game [29], in which the crowdsourcer is the leader and the users are the followers, while both are players in the game. In the first stage of the game, the crowdsourcer announces

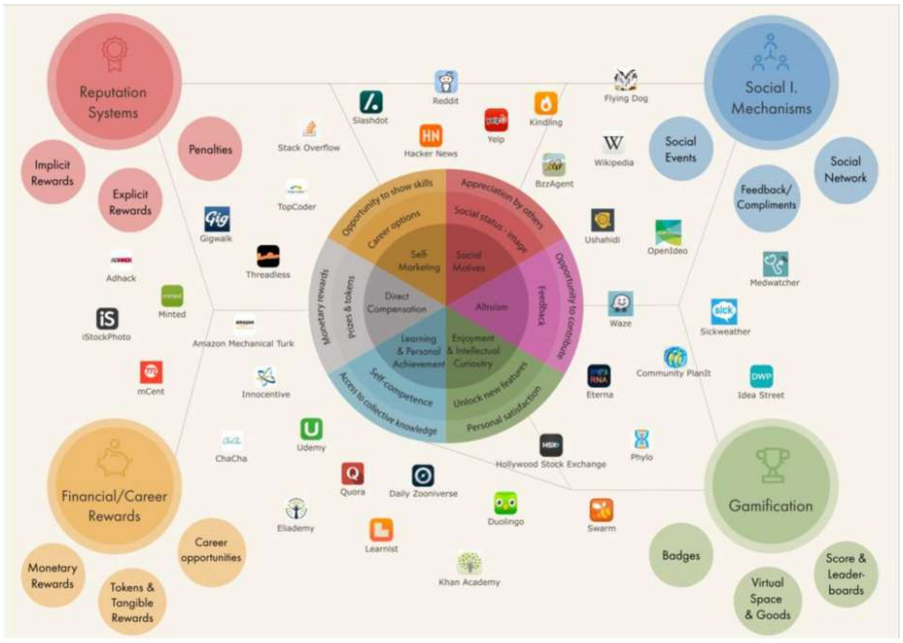

Figure 3.1: User motives, incentives and incentive mechanisms [40]

its reward $\mathcal{R}$ (the strategy of the crowdsourcer), following strategizing of their sensing time in order to maximize their utility by the users, in the second stage. It is then showed how to compute the unique Stackelberg Equilibrium, at which point the crowdsourcer's utility is maximized, and none of the users can unilaterally boost its utility by diverging from its existing strategy. For the user-centric model, they design an auction-based incentive mechanism, that takes bids from the users as input, chooses a subset of users as winners, and determines the payment to each winning user. This approach has proved to be computationally efficient, individually rational, profitable, and truthful, as stated by the authors.

3.2.5 Peer-to-Peer

Rius et al. [64] claim that one of the most critical aspect in the design of P2P systems is developing an incentive technique liable for the encouragement of cooperation and resource sharing among the participants. As a solution, they offer an incentive mechanism based on credits with a two-level topology. They implemented non-negative credit function with a historic term used to differentiate between newcomers and old collaborative peers, with the goal to prevent ID-changing cheating.

Their system describes three types of roles, a peer in the network can assume: managers (M) that are in charge of managing the system and assigning the tasks, workers (W) that provide the resources for task execution, and masters (MS) – a peer that submitted a job to the system and is currently monitoring its execution. Peers grouped together in a neighbouring area make up the low level of the topology. The association is made up of one manager and n workers, depending on the network properties, bandwidth and latency. Areas interconnected through the managers by means of an overlay make up the upper level. The credit-based incentive scheme, called *Weighted*, is proposed at the low level. A peer that launches a job has to pay for it to be executed. In the event that the peer does not dispose of the full amount of the credits required, the job will be nevertheless executed and the worker nodes will be paid in full. In this case, the initiator node will pay everything it has (leaving its account at zero) and the system will create the rest of the required credits for the workers. Tasks launched by peers with credits on their account will be prioritized, while the tasks coming from nodes without credit will be selected in a First In, First Out manner, and executed only when the system is idle.

They claim that the proposed incentive mechanism surpasses alternative approaches, increasing the system performance up to 50%[2] on average.

Kaune et al. [41], explore the relationship between P2P system and cooperation in human society. The similarities they found between P2P systems and human societies, along with the studies that showed that the concept of reputation was successful in encouraging cooperation in human societies, lead them to design a new reputation-based incentive scheme, encouraging honest nodes to participate, while blocking the free-rider nodes. They present a distributed reputation infrastructure, where reputation values are presented by a globally binary digit that can be either 0 – representing a good (G), or 1 – representing a bad (B) standing. Those values are assigned to the nodes based on their last action in the role of service provider. A reputation transition, depending on *i)* the current reputation of the service provider, *ii)* the current reputation of the service consumer, and *iii)* the action taken by the service provider (cooperate (C) or deny cooperation (D)) , evaluates the goodness of the action. Consequently, each tuple is mapped to either 0 or 1 defining the new reputation value for the given peer. A decision function is used by each peer individually to decide how to behave towards requesting service consumer.

They conclude by saying that the simulation they conducted showed that nearly all peers who wanted to successfully launch service, have to contribute to the system, resulting in the elimination of the free-riders. They also stated that malicious peers, interested only in disrupting the network, were quickly ostracized.

3.2.6 Fog Computing

A couple of research papers addressing the issue of incentive mechanism have been published, focusing on one specific aspect of the Fog network.

The paper published by Nazih et al. [56] focuses on Vehicular Fog Computing (VFC), tackling the problem of resource allocation in a VFC environment. Since they do not

[2]Increase in the marginal participation cause by reinvestment policy, since the more credits reinvested in the system, the more peers will be able to launch their jobs.

presume that vehicles just assume the role of a Fog node unconditionally, they propose a game theoretic based incentive mechanism that motivates vehicles at the edges to share their computational resources. They start by considering that every UserEquipment (UE)[3] chooses a Data Service Operator (DSO) that has a shared pool of computing resources. The DSO then proposes a price to the UE, while trying to sign a contract with the granted vehicle for its idle resources. This scenario contains multiple DSOs, which results in them competing to offer an acceptable price. A joint optimization framework is presented, that combines a Stackelberg game, to model the interaction between the DSOs and UEs, and contact theory, to manage the relation between the DSOs and vehicles. In the first stage, demands coming from UEs are set as a pricing relationship modeled as a Stackelberg game. Validation of the proposed price, as well as the size of the computing resources purchased by the UE also takes place at this time. Stage two includes the creation of a contract with multiple contract items, where each contract item specifies a relationship between the amount of computing resources requested from the vehicle, and the corresponding payment for its contribution.

They claim that the simulation results demonstrate that devoting the computing resources of vehicles to the UEs demands considerably enhances the performance of a VFC in terms of resource-sharing.

Zeng et al. [89] use the framework of contract theory to devise negotiation between task publisher and Fog nodes as an optimization problem. This is done as an incentive mechanism to encourage nodes to participate in computation offloading, by sharing their idle computing resources. The authors describe the following idea: Fog nodes that participate in computation offloading, and accomplish tasks in time, are offered a monetary reward by the task publisher. The task publisher, however, does not know in advance how many nodes are willing to participate, and to what extend are they willing to contribute. Their solution: the proposed contract defines the monetary reward for various latency requirements, with higher incentives being provided to Fog nodes that can execute jobs within shorter latency. The optimal designed contract is the Nash equilibrium solution achieved by task publisher and Fog nodes. In this case, the utility of the task publisher, as well as that of the Fog nodes is maximized and can not be further enhanced on their own.

Simulation findings indicate that an optimal contract can increase task publisher utility, while also ensuring individual rationality and incentive compatibility across Fog nodes.

3.3 Discussion

The proposed Fog computing incentives have some constraints, and cannot be applied in general without certain modifications, e.g., the method proposed by Nazih et al. [56] focuses only on VFC, and does not consider other applications. The goal of our paper is to create an incentive mechanism that would have no such constraints, and could apply

[3]Any device used directly by an end-user to communicate

to the whole Fog network. Other incentive mechanisms described in this section come from different domains, that are all more or less different than Fog computing, e.g. in architecture, topology, services etc. The most important aspects of Fog computing, and how they are different than the already-mentioned areas will be portrayed in the next section.

CHAPTER 4

Design

4.1 Requirements Analysis

In this section, the most important aspects of Fog computing that should be taken into consideration when designing an incentive mechanism, are discussed. We start by discussing a typical system architecture for the Fog (Section 4.1.1). We then move on to address major characteristics of Fog computing and how they influence the design of an incentive mechanism, namely heterogeneity (Section 4.1.2), proximity awareness (Section 4.1.3), scalability (Section 4.1.4), and volatility (Section 4.1.5).

4.1.1 System Architecture

As already depicted in Fig. 1.1, Fog architectures are usually hierarchical, consisting of three layers, out of which two are relevant for the development of an incentive mechanism. The lower level is inhabited by the edge devices, while the upper level consists of Fog nodes [77].

- *Edge Devices Layer*: consists of a large amount of small heterogeneous devices, usually focused on one aspect of the process. They are equipped with sensing abilities and low store/processing capabilities. The limited resources of the nodes in this layer are insufficient to perform more complex operations, which is where Fog nodes, also called processing nodes, are engaged. Thanks to their sensors, edge devices possess information related to their respective context, which enables them to act as source nodes for the processing nodes, i.e., nodes in the upper level.

- *Fog Nodes Layer*: contains less devices that execute processing task at the edge of the network, offering virtualized resources closer to the edge devices. Fog nodes are also responsible for identification of source nodes.

This architecture is significantly different from other system types where incentive mechanisms play an important role. P2P systems do not have a layered architecture. Such systems rely on a flat hierarchy, in which all nodes are observed as equals. Every node in the network can create direct links with any other network member, in order to share or download information [60]. In crowdsourcing, there are two main actors, namely an initiator and the participants [36]. The initiator designs and launches the crowdsourcing action, distributes the resources to the participating nodes, and is in charge of collecting and evaluating the result. Participants, at the very least, offer their resources to the task initiator. If that is required, they also do some kind of processing tasks, and (optionally) report the results back to the initiator.

4.1.2 Heterogeneity

This is one of the main, and most important, characteristics of Fog computing. It refers to both heterogeneity of the devices themselves, as well as resource heterogeneity of the nodes. The former is concerned with how many different types of nodes are present in a network (smartphone, smart watch, remote controlled air-condition, traffic lights, autonomous vehicles, health monitoring wearables etc.), while the latter deals with the very wide range of resource capacities the processing nodes possess. These capabilities can be in regard to Central Processing Unit (CPU) and memory, but also regarding host services and applications [39]. Another significant aspect of heterogeneity is that Fog nodes are deployed and working on a variety of environments and platforms [16]. Therefore, supporting node heterogeneity becomes essential, not only in the existing network, but also has to be taken into consideration when creating an incentive mechanism, to ensure that a Fog system runs smoothly. The network, as well as the mechanism, should know which node needs to be attracted for which position in the hierarchy, and which tasks they should be in charge of.

Contrary to the Fog, other systems do not take heterogeneity into account when distributing tasks. In P2P networks, the system assigns each node equal responsibilities regarding routing messages and storing data, even though not all nodes in the system possess the same capacities [76]. In crowdsourcing, the initiator launches a task without regard of resources of individual nodes [36].

4.1.3 Proximity Awareness

Since communication and information distribution are more efficient between nodes that are close-by (both physically and logically), than those who rely on centralized intermediaries located far away [30], making sure that the processing nodes are located as close as possible to the IoT devices is another important goal of Fog computing. This enables reduction in communication latency. That means, when creating an incentive mechanism, we want to acquire the node that is in close proximity with as many nodes as possible. Those are the ones that interest us the most, since they will have the biggest effect on the performance of the network. However, this task can be challenging in

Fog computing, since proximity measurements may clash with other aspects [39]. For example, a node could be in proximity with many nodes and would be a great asset to the system, but at the same time offers very little processing power, which is also an important factor when entering the Fog network.

P2P and crowdsourcing networks do not usually consider proximity as factor when distributing their messages, which results in an arbitrary long travel distance of the messages. Some more recent P2P overlay infrastructures, such as Tapestry [91] and Pastry [65] propose an approach where a proximity metric among pairs of nodes is measured, and choose the nodes that are nearby to include in their routing table. This comes at the cost of a more expensive overlay maintenance protocol [20].

4.1.4 Scalability

As already discussed, the number of IoT devices is increasing constantly and rapidly [73]. To be able to properly handle the ever-growing demand of edge devices, the number of Fog compute nodes needs to rise as well. Keeping in mind that processing nodes span from the edge of the network to the Cloud, the Fog network may need to scale to a great extent. What makes this process even more difficult is that new nodes can join or leave the network at any time. In addition, as discussed in Section 4.1.3, interests of the Fog system have to be taken into account when including new nodes.

Both P2P and crowdsourcing networks are scalable by design, since the networks can expand productivity with near-zero marginal cost. This aspect can be inherited from the incentive mechanisms from these fields.

4.1.5 Volatility

Volatility plays a big role in Fog computing. Nodes in the system can come and go at any time, meaning that computing resources in the Fog can appear and disappear rapidly [12]. The vehicular Fog is a great example of volatility: the connection between edge devices and Fog nodes (e.g., cars and roadside unit) is established for a short period of time, until the car passes by, during which period the next car has already joined the network and so on. The system needs to take this aspect into consideration, and enforce an appropriate mechanism that is able to detect the arrival and removal of resources.

As a decentralized system, P2P has also a very volatile topology [87]. The same is true for crowdsourcing systems. Once again, with regard to this aspect, incentive mechanisms from these domains can be adapted.

4.2 Design

When choosing an incentive mechanism for a Fog network all the aspects discussed in Section 4.1 need to be taken into consideration. Most of the mechanisms described in Chapter 3 do not cover all these topics, and therefore would not be a good fit, i.e., would need a lot of modification. In their paper, Rius et al. [64], compared to other mentioned approaches, consider the most of the discussed requirements, and propose a mechanism that reflects them. Consequently, we have decided to use this paper, and the incentive mechanism for a P2P network they describe, as a reference for this thesis.

In their earlier paper [63], they present a global P2P credit-based scheduling model that captures P2P user dynamics, efficiently penalizes free-riders, and encourages peer engagement. This incentive mechanism is implemented in a decentralized architecture for distributing computation with a tree-like architecture called CoDiP2P [49, 50]. The paper analyzed in Section 3, i.e., [64], provides an extension to their work by abandoning the tree topology, and making it possible for the model to run on any type of structure, given it can be sub-grouped and managed by a super-peer. Based on the mechanism proposed in these two papers, our take on an incentive mechanism for the Fog network will be presented in this chapter.

4.2.1 The Framework

In this section, we define framework conditions, as well as the assumptions made, in order to have a better understanding of the presented incentive mechanism.

The system consists of logical areas that are interconnected through one single point of contact. Each logical area has an arbitrary amount of nodes (a device in the Fog network) and operates independently from other areas. When joining the Fog network, new nodes can be attached to an existing area (if the capacity is not at a maximum), or cause a creation of a new area if all current areas are full. This quality, alongside the fact that node disconnection does not cause restructuring, speaks to the scalability of the system. A more detailed discussion on the architecture is provided in Section 4.2.2.

We define two kinds of roles in an area (also in the system in general) that a node can have: *manager* and *worker*. Each area has one manager, which is also a point of contact for other areas, and many worker nodes. At any given point of time, any node can submit a task for execution (this node is then called *submitter*). A manager's main obligation is to schedule and assign tasks making up the submitted job to the worker nodes in the same or neighboring areas (taking proximity awareness into consideration), after which the manager receives a commission. Worker nodes are in charge of processing the scheduled tasks, for which they are compensated. Only worker nodes can submit a job to the system, paying a certain amount of credits for this action. Further evidence on roles in the system are presented in Section 4.2.3.

In order to keep track of all the submitted jobs, the manager stores these in a *Queue*. The queue is a local storage of all submitted jobs in a certain area, to which only a manager of that area has access. Once a submitter launches a job, it lands in the queue, where it is prioritized (a debate on job prioritization is conducted in Section 4.2.5). As long as the queue is not empty, the manager takes the first element of the queue and processes it, earning a "processing fee" as a reward.

The processing performed by the manager includes scheduling and assigning tasks from said job to the workers, based on the requirements of the tasks, as well as the workers. Heterogeneity of the system is seen through the fact that each worker disposes of certain quantity of computational resources, that can be measured in CPU, memory or bandwidth. These criteria play a significant role when calculating the computational potential of a worker. At the same time, each task has different difficulty level (determined by the computing capacity required for its execution), meaning the manager has to find an appropriate worker that can handle the task at question. Taking into account that one worker can only process one task at a time, we can consider the full potential of the worker. By defining workers as mono-task workers, a worker is marked busy as soon as it has a task currently executing. On the other hand, a worker is free when no task has been assigned to it, or it has finished the execution of the assigned task. Section 4.2.5 provides additional examination of this topic.

The main idea behind this framework is to use a non-negative credit-based system to motivate the nodes to participate in resource sharing. To accomplish their goal (have a certain job executed), they need to pay for it in credits. The success of a job launching depends on the acquired credits of the submitter. As a means to acquire said credits, they need to execute tasks submitted by other nodes. Therefore, the principal purpose of the suggested credit policy is to increase system throughput. Reinvestment is another option supporting this. Managers can reinvest some of the credits they receive from processing the submissions into the system, stimulating the workers even more. Besides increasing credit flow through the system, it is also a good technique to tackle volatility in Fog systems (see 4.2.5 for supplementary report on this topic).

4.2.2 Architecture

Rius et al. [64] introduce a system with a two-level topology, where the worker nodes, grouped under one manager, make up an area. The upper area is an overlay by which the managers of the areas communicate with each other.

An *area* is a logical space made up of a specified amount of workers governed by a manager. The number of workers, represented by $\mathbb{N}$, depends on the network properties, bandwidth, and latency, and can vary for all areas in the network. Per area, there is one manager. An example of an area is depicted in Fig. 4.1.

Although forming of these areas is not the focus of this thesis, this paragraph provides an overview of the architecture. This logical area corresponds to a modified version of the architecture described in [49], that has scalability, distributed management, self-organization, and heterogeneous resource management as main goals. The structure supports three main operations:

1. Insertion: When a node joins a network, it communicates with the node closest to it, which returns the address of the manager of its area. After receiving the request, the manager checks if its area has a free site for the new node. If that is the case,

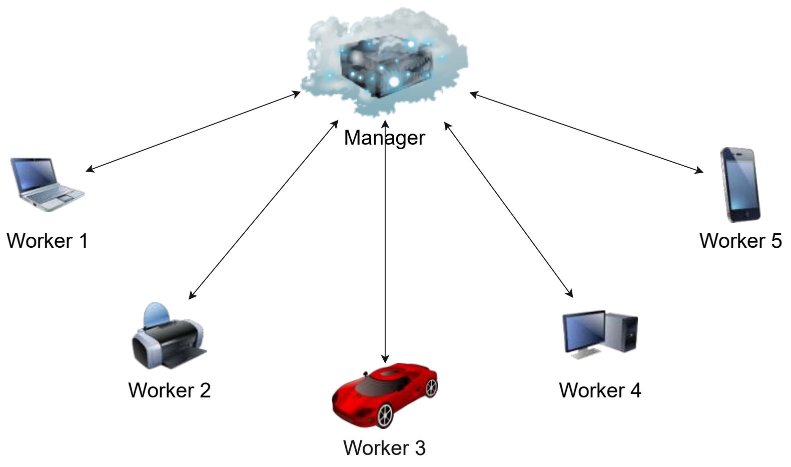

Figure 4.1: An area consisting of one manager and $\mathbb{N} = 5$

it proceeds to become a new worker in that area. If there is no room in its area, the manager forwards the request to the neighboring managers which do the same thing for their areas. If all areas are full, the managers try to find a worker that can be changed to a manager so that a new area can be created to locate the new node.

2. Maintenance of the System: Every T seconds a manager sends *ManagerAlive* message to the workers in its area. The workers respond with information about their available computational resources.

3. Output: If a node does not send its statistical information, the manager assumes that the worker has (voluntarily or involuntarily) left the network. In that case the node is simply detached from the network[1].

An area presents the lower level of the mentioned two-level topology. Multiple areas can be interconnected through the managers by the means of an *overlay*, composing the upper layer, where each area can have a different number of $\mathbb{K}$ neighbors, i.e., links. Fig. 4.2 shows an example of such a topology.

Once a worker node submits a job, the job arrives at the manager of that area. The manager checks if enough resources are available in their local area to process the job. If the answer is positive, the manager assigns the tasks to workers from ita area as explained in Section 4.2.5. If the local area can not handle the job, the manager communicates with their neighboring areas and tries to find one that can (discussed in Section 4.2.4).

Number of nodes in a Fog system can rapidly increase, and can reach very high numbers. By splitting the network into two logical layers, we increase performance of the system.

[1]Failure of the manager node will be discussed in 7.2

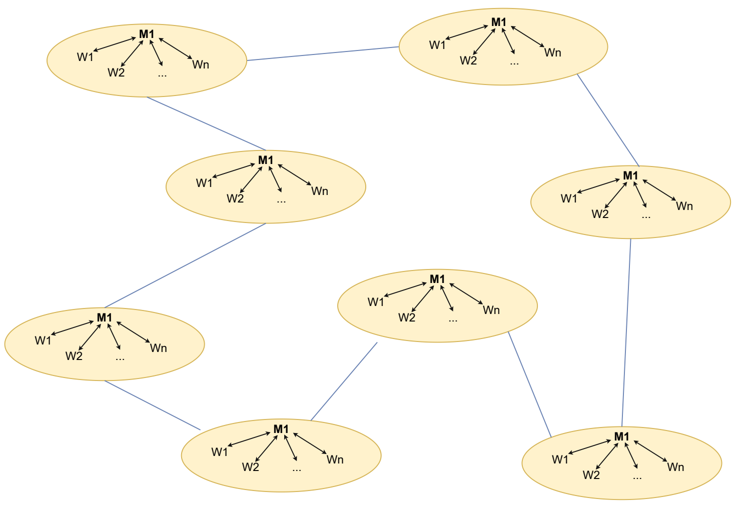

Figure 4.2: A system overlay where $\mathbb{K} = 2$

Manager is the center entity of information in an area, and possesses knowledge about all the nodes in its area (number, CPU, load, status etc.), as well as all about the jobs submitted to the area (submitter node, timestamp of submission, number of tasks etc.). If we would have only one manager for all worker nodes in the system, pulling and storing all that information could become troublesome. In addition, since there would be only one manager, the submitted jobs would most likely have to "travel" a greater distance to reach the manager. Lastly, submitters would have to wait longer for its job to be processed, since the manger is mono-task and can only schedule one task at a time.

The proposed architecture takes all this into account, and divides the network in multiple areas with one manager per area, handling a certain amount of nodes, usually receiving jobs of submitter nodes from its own area (that means physically the closest). Managers can assign tasks parallel with other area's managers, significantly increasing job throughput.

4.2.3 Roles

This incentive mechanism relies mostly on *managers* that have multiple functions (Fig. 4.3). The following are the responsibilities of a manager:

- *Job queue management:* After a node submits a job to the system, it lands in the manager's queue, where it is prioritized.
- *Job scheduling*: The manager node acts as an intermediary between the node that submitted the job, and those which will process it (also respectively called sender and receiver). After it has received the job in the queue, and it has been accepted, a manager makes sure to find workers to process the job's tasks. For this function, managers earn commission.

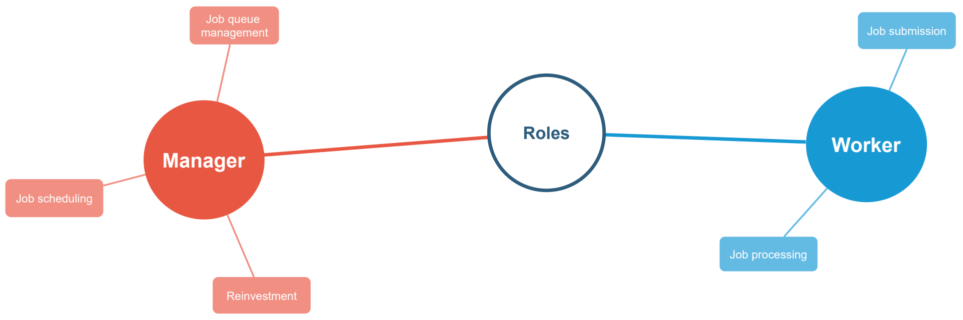

Figure 4.3: Roles a node can have in our system

- *Reinvestment:* Credits earned through commission can be reinvested in the system, as a means to motivate the nodes to participate even more, i.e., give their resources to the system.

Any node in the system that is not a manager, is called a *worker* (Fig. 4.3). Workers can either:

- *Submit a job* to the system they want to be processed, for which they have to pay a certain amount of credits.
- *Process a task* of the job submitted by a worker, for which they will receive a certain amount of credits.

The manager position is reserved for the older nodes (workers) in each area. This way we ensure some Quality of Service (QoS) for the network, minimize churn rate (discussed in 7.2), and hinder manager ID-changing (Identity-changing) (discussed in the next section).

An alternative would be to always choose the most powerful node. This approach could also guarantee some QoS, but it does not solve the other mentioned problems. If a node leaves the network and decides to rejoin (to gain better reputation, or for any other reason), it starts off as a completely new node, meaning that all the nodes in the area are older, and have a better chance of becoming a manager one day. Awarding the most powerful nodes, and not taking age into consideration does not decrease the total number of nodes leaving the network (the churn rate), unlike the suggested approach. By giving older nodes the role of manager, we discourage nodes to disconnect from the network and reconnect.

4.2.4 Upper-level Incentive Mechanism

The upper-level Incentive Mechanism, also called global, focuses on three scheduling criteria the managers consider during task allocation: *Computing Capacity with Neighbors*, *Distance* and *Reputation*.

Computing Capacity with Neighbors (CCN)
Keeping in mind the fact that each manager knows the number, CPU power and load of each worker in its area, *Computing Capacity (CC)* for its local area can be easily calculated. If we assume that managers have the ability to delegate duties to their neighboring areas, we can also consider the CC for these areas. We define CCN of an area i as the weighted local CC, increased by the average CC of the neighboring areas:

$$CCN_i = \alpha \cdot CC_i + (1-\alpha) \cdot \frac{\sum_{j=0}^{K_i} CC_j}{Ki} \tag{4.1}$$

The α parameter regulates the amount of tasks distributed to the powerful areas. The greater the α, the more tasks are assigned to the local area and fewer are distributed to the neighboring managers, and vice versa. Experiments carried out in [64] show that load balancing, task propagation and information sharing performance reach a maximum when $\alpha = 0.35$. This results in a more balanced and efficient scheduling, since setting the value greater than 0.35 leads to the scheduling policy taking only local CC into consideration, neglecting the neighboring resources[2].

Distance
In scenarios where communication plays an important role, *distance* between nodes is a crucial aspect. Such case is our smart transport example scenario. In the computational world, distance usually signifies the latency between two users, in our case – nodes. Since calculating the distance between the submitter and each potential worker would be too expensive, an attribute called *Mass Center (MC)* is introduced. This property represents the area's degree of dispersion. The higher the MC, the less dispersed the workers in an area are, and vice versa. It is based on the CC of the worker, and defined as the weighted average of the relative position of the worker in an area. This means that a powerful worker's distance will have a greater impact on the area's MC than the distance of a node with a low CC. The proximity of each manager to the mass center is hence highly evaluated when a job is submitted. The MC of an area i is calculated as:

$$MC_i = \frac{\sum_{j=0}^{N_i} (\frac{dist_{max} - dist_j}{dist_{max}} \cdot CC_j)}{\sum_{j=0}^{K_i} CC_j} \tag{4.2}$$

The distance between two nodes is calculated using the Vivaldi [24] system. Vivaldi accurately predicts the communication latency between two hosts based on the distance, whereby the distance presents the distance between two synthetic coordinates assigned by the method.

Reputation
The *Reputation (R)* of a manager is defined as the probability of the said manager's

[2]For more information on this parameter, and how it effects the task propagation, load balancing and information spreading, please refer to [64].

successful service invocation. This is relatively easy to calculate, once the manager has already participated in the scheduling process, and we define it as the ratio between the total assigned tasks by the manager of an area (TT) and successfully completed tasks by the assigned workers of the area (ST):

$$R_i = \frac{ST_i}{TT_i} \tag{4.3}$$

The issue arises with the ones that had no assigned tasks previously – the new managers. There is no perfect solution to this, since setting the initial value close to 1 would result in prioritization of the new managers, that could potentially be dangerous. At the same time, setting the value close to 0 would put new managers at a disadvantage, for no concrete reason at all. Considering all this, as well as the fact that reputation is only one of the three criteria used to qualify the resources controlled by a manager, we decided to keep the initial reputation value at 0.5.

This could raise the question that if a manager earns a reputation lower than 0.5, that it could try to disconnect and rejoin a network as a manager, to gain a better initial reputation. However this is prevented by the manager-position assigning discussed in the last section.

Scheduling Criteria

All these values are used in the *Scheduling Criteria (V)*, on which the scheduling procedure of tasks across the network of interconnected managers is based. Per area i, it is calculated as the sum of weighted normalized MC of that area, weighted normalized CCN of the area and weighed reputation of the area's manager.

$$V_i = (\frac{MC_i}{MC_{max}}) \cdot \beta_1 + (\frac{CCN_i}{CCN_{max}}) \cdot \beta_2 + R_i \cdot \beta_3 \tag{4.4}$$

where $\beta_1 + \beta_2 + \beta_3 = 1$, and the weight assigned to those attributed is dependant from the job attributes. It would be important that β_1 is higher in scenarios such as our example scenario, where communication is the most important aspect. β_2 should be considered in jobs that require high computation intensity, while β_3 could be used to guarantee QoS.

Global Scheduling Algorithm

The manager will update all the Scheduling Criteria (V_i) from its K_a neighbors before applying Algo 4.1. The algorithm checks if the workers in the current area can handle the load. This calculation is done using the μ parameter, which can be user-defined and presents the percentage how many workers should be free, for an area to process these tasks. If the number is closer to 1, the algorithm tends to assign the tasks to the local area, while a number closer to 0 usually results in task propagation across the upper layer, scanning the system for the best workers. Experiments in [64] showed that by knowing 10% of the system, the scheduling reaches 80% of the optimum performance.

Algorithm 4.1: Global Scheduling Algorithm

```
1  Require: $(Job)$: Input Parameter;
2  if $\#Tasks\ of\ the\ Job < \mu \cdot Workers_a$ then
3      Schedule Tasks using Local Scheduling Algorithm;
4  else
5      if $exists\ V_i\ so\ that\ V_i > V_a$ then
6          $Manager_{highest} = Manager$ with highest $Scheduling\ Criteria$;
7          submit job to the Manager$_{highest}$
8      else
9          Schedule Tasks using Local Scheduling Algorithm;
10     end
11 end
```

That number increases to 90% when 50% of the system is familiar. They concluded that the optimal value for μ is between 10 and 20 (i.e., between 0.1 and 0.2), since at that point the scheduling reaches 80-85%, respectively, of the optimum scheduling[3]. Everything above that is considered too expensive. Task allocation in the local area (lines 3 and 9) is described in the following section. Line 5 checks if any of the neighbors have a higher Scheduling Criteria (V_i) than the manager executing the algorithm (V_a). If that is the case, the job will be forwarded to the manager with the highest Scheduling Criteria (line 7). If line 5 returns false, the job stays with the current manager, and should be executed locally (line 9). This V_a comes from formula 4.1, where $\alpha = 1$ (so that only information about the own local area are taken into consideration).

4.2.5 Lower-level Incentive Mechanism

At the lower level (also: area-level), a credit-based incentive scheme, implementing a non-negative credit function is proposed.

Credit Management

When a new node joins the system, it starts with zero credits. The main motive behind this is to prevent ID-changing attacks. ID-changing attacks are occurrences in which a participant reconnects to the system with a new identifier to be treated as a new user [64]. There are two potential scenarios in which this would be beneficial:

1. If the system awards newcomers to motivate them to participate in the network. In this scenario, after spending their initial reward, a node could leave the network and connect again with a new ID, obtaining the initial credits anew. This can happen repeatedly, thus enabling free-riding[4].

[3]For further information on this parameter, please refer to [64]
[4]A circumstance in which a node uses the system resources without contributing to it.

Algorithm 4.2: Job Admission Algorithm

```
1 Require: (Job): Input Parameter;
2 Manager receives Job from Sender;
3 if Area is idle then
4     Manager schedules Job;
5 else
6     Manager queues Job;
7 end
```

2. Allowing nodes to have a negative credit balance, which would in some way penalize them (e.g., lower priority in job processing), could also push nodes to reconnect and get around this disadvantage.

To properly handle this in our system, a few mechanisms have been implemented. First, newcomers always start with zero credits. This way, rejoining the system does not bring any gain. By not making it possible for a node to have negative credits, i.e., not having a disadvantage in comparison to the newcomers, the node's interest in reconnecting decreases. Lastly, a historical term, recording the number of collaborations made by each worker, discourages nodes to practice ID-changing. This serves to distinguish between the malicious ID-changing free-riders and collaborative nodes who have used up all of their credits by submitting work to the system. That means, to launch their jobs, newcomers will have to first share their resources, and execute foreign tasks.

Nevertheless, since the resources of the system are volatile, meaning if they are not assigned they are wasted, if the system is idle, i.e., there are no jobs in the queue, a job launched by a node with not enough credits, will also be accepted, and executed. Consequently, workers processing the job, as well as the manager will receive the deserved amount of credits. Since they can not receive the credits from the submitter, the incentive mechanism will produce the credits needed.

The behaviour of the system, with regard to volatility is represented in the *Job Admission Algorithm* (Alg. 4.2), which checks if the system is in an idle state (line 3). If that is the case, the submitted job is processed straight away (line 4), otherwise the job is added to the queue (line 6).

An alternative to the process in which the system itself creates the required credits could be that the submitters have to actually buy them. The main goal of this incentive mechanism is to motivate the nodes to become a contributing part of the network, by processing the tasks and earning credits for their effort. These gathered credits come into play when a node needs to submit a task to the network. Workers that do not execute tasks, do not receive credits. Instead of saying that the system will make up for the difference of submitter's missing credits, we could make nodes buy the credits they need.

Another aspect of credit management is reinvestment. Managers, being rewarded for each job they schedule, may accumulate a lot of credits. In the described system, they can reinvest some of those credits into the system, to stimulate collaboration. The additional credits coming from the manager serve to stimulate workers to execute more tasks. The more tasks the workers execute, the more credits they receive, the more tasks are submitted for launching. The managers are rewarded with additional credits when more jobs are scheduled. Thus, by reinvesting in the system (awarding workers), the managers earn credits[5]. The purpose of such design is to increase system throughput.

In comparison to other approaches, this mechanism does not distribute the credits uniformly in the network, but they are divided between the participating nodes, thus not rewarding free-riding inactive nodes.

Local Scheduling Algorithm

For a worker node to be even considered for a job, it needs to first define the price of the job. This is done via a method that takes the CPU, memory, and bandwidth, or any combination, into account. This method intends to rate the relative value of computational resources of each node, which are presented in credits and portray the price a submitter has to pay for the job execution. These prices are used in the scheduling algorithm described in this section and are stored in the manager of the respective area as respective cost *Value*[5].

The queue, which keeps track of the submitted jobs, takes the following criteria into account during prioritization:

- Credits available in the submitter's account.
- The number of jobs launched by the submitter, also called *historical term*.
- Time elapsed during which the job has been waiting in the queue to be served.

The purpose of this mechanism is to penalize free-riders by assigning them the lowest priority in the queue. We ensure that peers can only enhance the launching priority by completing system tasks using this methodology.

The *Local Scheduling Algorithm* (Alg. 4.3) explains how tasks are distributed to the worker nodes. It takes the next highest prioritized job (line 3) and uses a reverse Vickery algorithm [79] to choose the workers to execute the job. The job is divided into tasks, and each task is assigned to a different worker in the area. Considering only workers that have enough resources to handle the task at hand, the manager chooses the worker with the lowest cost value in the area to process the task (line 7). This worker is in return awarded with the difference between the second lowest value and its own (line 10). For its scheduling duties, the manager is compensated with the difference between the

[5]The method itself is not defined, since it depends on the end user and can be customized.

Algorithm 4.3: Local Scheduling Algorithm

```
1  Require: ($Job$): Input Parameters;
2  AvgValueDif = Average difference between sorted values offered by the workers;
3  $Manager$ gets the next $Job$ from its $Queue$;
4  $Profit\_M = 0$, $Rein = 0$;
5  $V_{max}$ = Maximum $Worker$ cost in the area;
6  foreach $Task$ in $Job$ do
7      $Worker_{lowest}$ = Free $Worker$ with the lowest $Value$ in the area ;
8      $Value_{lowest}$ = Lowest free $Worker$ cost in the area;
9      $Value_{2nd\_lowest}$ = Second lowerst free $Worker$ cost in the area;
10     $Profit\_Worker_{lowest} = Value_{2nd\_lowest} - Value_{lowest}$;
11     $Profit\_M = Profit\_M + (V_{max} - Value_{lowest}) \cdot (1-\delta)$;
12     $Rein = Rein + (V_{max} - Worker_{lowest}) \cdot \delta$;
13     $Manager$ sends $Task$ to $Worker_{lowest}$;
14     Submitter pays $Worker$ the $Profit\_Worker_{lowest}$ amount;
15 end
16 Submitter pays $Manager$ the $Profit\_M$ amount;
```

maximal and the minimal value cost in the area (line 11). That is why it will always choose the best option, i.e., the one in which it gains the biggest profit. The reinvestment policy described in the section above is portrayed by the δ factor in lines 11 and 12, which takes some of the credits earned by the manager and saves them in the *Rein* variable. This represents the total number of credits that can be reinvested in the system by the manager, which can be done using different reinvestment policies. The proposed incentive mechanism distributes credits in a non-uniform manner based on the worker's contribution to the system and, in particular, their computational demands. Accordingly, δ allows us to specify the portion of the manager's credits that will be distributed among the workers. The submitter pays for each task execution individually, after successful termination (line 14). After all tasks of the job have been executed, the submitter pays the manager fee (line 16).

By using the reverse Vickery auction, we prevent node cheating in which workers offer values lower than their respective cost. If a worker attempts to get selected by decreasing their cost *Value*, in the case that the second lowest cost is higher, selecting it will result in a negative profit. Thus, cheating is discouraged.

During the job scheduling step, we need to consider a couple of edge cases:

1. Workers with the lowest and the second lowest value offer the same price
2. There is only one free worker available

In these scenarios worker executing the task, as well as the manager would not receive any credits, considering that the result of the subtraction would be 0. To avoid that in

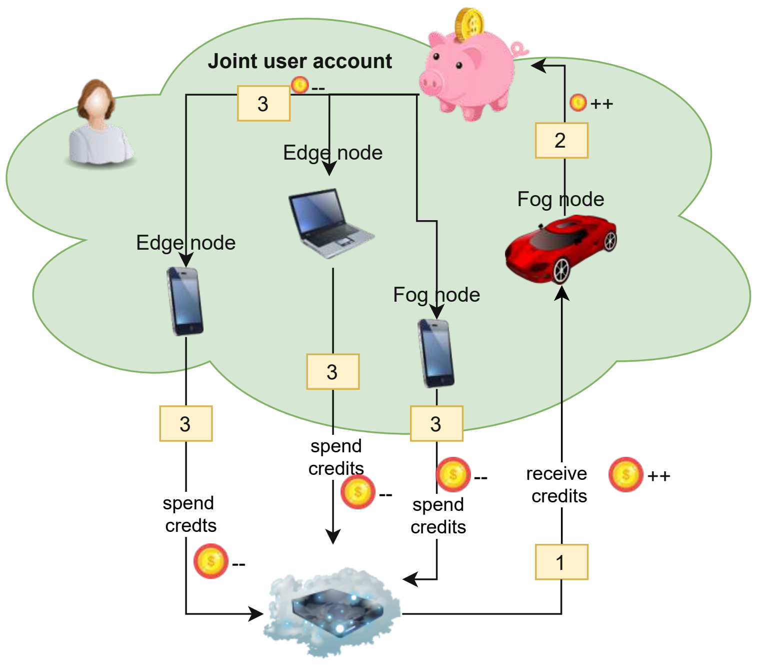

Figure 4.4: Credits shared amongst devices of one user

the first scenario, the algorithm will take the next (third) lowest element, then the fourth and so on. If all remaining workers have the same value (which is, in the sense of profit, equivalent to scenario 2), both worker and manager will be awarded an average value difference, calculated in line 2.

Shared credits

A Fog network, having two types of devices (edge devices and Fog nodes), faces some issues regarding the described mechanism. Edge devices, being solely a data source, do not participate in job execution. Consequently, the stated behaviour is restricted only to Fog nodes, meaning only Fog nodes can earn credits. In other terms, jobs submitted by edge devices would always have the lowest priority in the manager queue (since they do not provide any credits). As a solution to this conduct, we have designed a *joint user account* option. Such an approach enables the owner to have one account and use the credits from it for all their devices (Fig. 4.4). As an example, a user can have one Fog node in an area, executing foreign tasks, and earning credits (label 1 in Fig. 4.4). These credits are all collected in the joint account of the owner (label 2 in Fig. 4.4), and can be used by any node (edge or Fog) of that user (label 3 in Fig. 4.4).

At this point, the question "why not use Fog nodes from own account to execute own tasks" could be raised. A couple of reasons exist why this is indeed not the desired behavior. First, as suggested in the 4.2.2 section, it can happen that nodes from the same owner do not reside in the same area. This is the case when more user nodes connect to the network after some time, while in the meantime the total number of nodes for its closest area is already reached. In such an event, the newcomer node has to join another area. Furthermore, enabling such an option could significantly hurt our incentive mechanism, since Fog nodes in the system could decide to execute only tasks coming from their account. The workers could execute some foreign tasks at the beginning, earning enough credits for an edge node from the same account to submit a task. This task will then be executed by the Fog node of the same user, not allowing the credits to leave the account. If the submitters are not too demanding (submitting only so many tasks as the worker(s) of the same user can process), the worker(s) will have no reason to contribute to the rest of the network, creating sub-areas of their own. Put in another way, Fog nodes that do not have joint account would have harder time executing their tasks.

Due to the aforementioned reasons, we have decided that our incentive mechanisms does not take joint accounts into consideration during job scheduling process, and every submitter and worker is considered as an independent individual[6].

4.2.6 System Flow

In this section, the flow from system startup is described:

1. At system startup, none of the users have credits. They are all considered equals, and no single node, i.e., their job, is prioritized over another. At this point, the job queue uses a First In, First Out (FIFO) scheduling policy.

2. Considering that the system is idle, the first node to launch a job is allowed to do so for free. However, the system creates the necessary credits to reward the workers executing the job.

3. The job lands with the manager, which decides whether its local area has enough resources to execute the whole job. If the answer is positive, a worker node is selected (based on criteria explained in Section *4.2.5*) and the job is forwarded to the worker. However, if the answer is negative, the job is conveyed to the manager of the neighboring area that has the most available resources (i.e., the highest chances of processing the task), which repeats the same process.

4. After the worker receives its compensation for the executed task, it becomes the only node in the system that has some credits (beside the manager). As of this moment, the jobs launched by this node is executed first, making the job scheduling policy a priority-based queue. In case that this node does not submit any new jobs, the system behaves once again as stated in Step 1, i.e., it selects jobs in a FIFO manner.

[6]Joint user account option plays an important role only as a means for the edge nodes to be able to pay for their tasks.

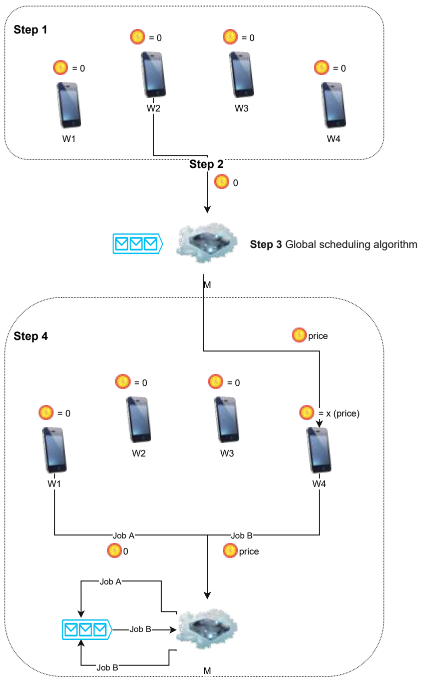

Figure 4.5: Abstracted system flow

5. If the current credit balance of a node amounts to less than needed for job execution, (after the job has been selected based on the priority-based queue) the node will pay with all its credits, leaving the balance at zero. The rest of the credits the executing node is entitled to are created by the system. This increases the credit growth in the system, making sure nothing happens to the node if it runs out of money.

Surely, the approach in which the system accepts and executes task from the nodes not having (enough) credits could be exploited by some nodes, creating a free-rider problem. However the risk of this happening is very small, since the only scenario in which the task of a node with not enough credits would be accepted, is if the system was idle. In that case, rather than the resources going to waste, accepting and executing the task brings gains to the whole system – the submitter's task is executed, workers get paid for task execution, and the manager earns commission.

Figure 4.5 portrays this flow inside one area, in which there are enough resources to process the submitted job. It is to understand that the workers, as well as the manager, from step 4, are the same entities as in step 1, i.e., 3, but are presented as new objects solely for better readability.

4.3 Discussion

This chapter provided an overview of the most important aspects of Fog computing, with regard to developing an effective incentive mechanism. The bigger part of the section described a new approach on how to motivate nodes to join a Fog network. The main idea is to enforce a non-negative credit-based reward mechanism that awards contributing nodes, and penalizes free-riders. That is achieved by implementing a two layer architecture that partitions the system into multiple areas, each of which has one manager, and N worker nodes that submit and/or execute computing tasks. Inside one area, a submitter launches a job it needs executed, which lands with the manager, i.e., in the manager's job queue. In case there are no jobs in the queue, meaning the system is idle, the manager takes the job and processes it promptly. Otherwise, the job is compared to the jobs in the manager's waiting queue and accordingly prioritized. The manager takes the highest prioritized job from the queue and starts the scheduling process. In the event that the job load can not be handled by the nodes in the current area, the job is forwarded to the neighboring area in which there is the highest probability of it being successfully executed. As long as the workers of the current area have enough capabilities to take care of the submitted jobs, the manager continues with the scheduling algorithm. For each task of the job, it finds the worker that demands the lowest payment for its services and assigns the task to the worker. The task executor is awarded with credits upon satisfactorily termination, and the manager earns commission for its employment. In addition, the manager receives reinvestment credits that can be used (as per user-defined functions) to further motivate participating nodes.

The two-layer architecture proposed for a P2P system, by Rius et al. [64], was used as a starting point for this thesis. Fog systems, being different than a P2P network, could not fully apply the proposed approach and demanded multiple changes. The biggest difference between these two systems (with regard to incentive mechanism) is that in P2P networks, only one type of nodes exist, so all nodes are considered equal. In Fog, a system node can either be an edge node or a Fog node, and they have to be treated accordingly. In our described mechanism, Fog nodes earn credits, used in job prioritization during the submission action, by executing tasks. Since edge nodes do not execute tasks, they do not receive any credits. To tackle this problem, we defined a joint account model, in which edge nodes can use credits earned by the Fog nodes owned by the same user. To further accommodate needs of Fog, the global scheduling algorithm required modifications. If a high number of workers would be occupied by the submitted job, the manager tries to find a neighbor that has a higher Scheduling Criteria, and could execute the job. If that is not possible, the job stays with the current manager, and waits for the a worker (during this phase the formula for CCN was also modified to better fit our desired network). In the local scheduling process, modifications had to be implemented as well. In the case that during the calculation of the profit, it occurs that a worker or the manager would receive zero credits (in situations already discussed in the previous section), we implemented a fallback option, where the affected node will receive the average of the value differences between sorted workers. This way, we make sure all nodes are paid for their services. Another significant distinction is the payment of the submitter node for each task immediately after its execution, it does not wait for all tasks to be handled to make the payments. In such manner, processing nodes receive their payment as soon as they are done with execution. The manager profit is paid after all tasks of the job are executed.

CHAPTER 5

Implementation

Now that we have proposed an incentive mechanism that would (in theory) be fitting for a Fog network, we want to test it as to see how it behaves in such environments in praxis. To that goal, we have implemented a program that simulates the described performance, allowing us to test the conduct, the power and the limitations of the system. This chapter offers details on the implementation of our simulator.

5.1 Existing Simulators

Several simulators in the field of Fog computing have been proposed, such as *iFogSim* [33] (followed by the extended version *iFogSim2* [47]), *MyiFogSim* [46], *EdgeCloudSim* [70], *YAFS* [43] etc. All these simulators support some form of cost, energy and network model. Most of them are built on top of CloudSim [32] and have a tree-like network topology (except for YAFS that is implemented in Python, and has a graph topology). They are all event-based, meaning simulation is based on events and not on the packets sent over the network. *iFogSim*, besides simulating fog computing infrastructures, enables execution of simulated applications in order to assess latency, energy consumption, network usage, and resource management. *FogNetSim++* [62] is built on top of OMNet++ [78], and includes a traffic management system evaluation for scalability and effectiveness demonstration, in terms of memory and CPU. Network parameters such as execution delay, packet error rate, handovers and latency are provided. *FogNetsim++* has support for sensors, fog nodes, distributed data centers, and a broker node in a static or dynamic environment. The broker's job is to monitor other devices and their demands. It allows resource scheduling algorithms to be executed, as well as provides an energy model and several price models.

In order to correctly test our incentive mechanism, we need to observe worker behaviour, monitoring contributions made to the system. One of the most important metrics we need to consider is the number of executed tasks per worker, as well as the number of submitted jobs. We need to analyze which nodes are usually selected for task processing, and which nodes never get selected. Job prioritization is another topic for the evaluation. We should try to find the limitation of the system, and see if there are any loopholes in the proposed system. Another important aspect is the ratio between the jobs executed locally and the jobs forwarded to the neighboring areas.

None of the above-mentioned simulators deal with these issues in depth. They have some other metrics in their focus, and lack the capabilities to address the issues at hand. That is why we have decided to implement a new, simple simulator that puts the emphasis precisely on these measurements.

5.2 Our Simulator

The simulator used for the experiments is a newly developed multi-thread program, written in Java. In enables parallel execution of multiple tasks (by different workers), job submissions, as well as the uninterrupted manager operations. The program converts the algorithms described in the previous section into Java code, and simulates task execution, while logging and storing data in a MongoDB database, providing us with an overview of what is happening with the data at all times.

5.2.1 Premises

This subsection explains the premises on which the system is built (partially discussed in Section 4.2.1):

- Managers do not execute tasks or submit jobs. Both operations can only be done by a worker node. The manager's only responsibility it to schedule and assign already submitted jobs/tasks.
- The worker submitting a job is not marked as busy, meaning that a node that is submitting a job can be selected to execute a certain task. Likewise, a node that is already executing some task, can at any point submit a job to its manager.
- As soon as a job is submitted to the manager, any worker node in the system can execute its tasks, including the submitter. This is a very unlikely scenario – if the submitter has enough resources to execute the task, it would not need to submit it to the network. However, we do not concern ourselves with why the submitter did not execute its job locally. Once the job is stored in the manager's queue, it does not differentiate between the submitter and "the rest".
- Payment operations are assumed to be atomic. The same is true for job prioritization, retrieving jobs from the queue and the assignment operation.
- Task execution is simulated by a sleeping thread. Based on the task difficulty, the worker-thread executing the task at hand will be put to sleep for a certain period of time.

5.2.2 Workflow

The flow chart in Figure 5.1 visualizes the system workflow. Threads are shown in yellow, decision boxes (mostly determining the frequency of the subsequent event) in brown, simplified actions are purple, and termination processes are red. Once the area thread is started, the manager thread is launched once, while submitter threads can be scheduled to run n times. All subbranches of the area thread are happening simultaneously, until the *terminate threads* action (in the middle line), after which all threads in the system start their closing process. Manager thread does its process (left branch) for each job it has to schedule.

The system workflow is carried out in the following steps (the respective step are marked in Fig. 5.1):

1. The system provides multiple program arguments that configure various values throughout the program. On startup, based on the provided arguments, a manager and a list of workers for each area are created (and stored). In this step, the local Computing Capacity of each area is calculated and stored in the manager. In addition, neighboring areas are defined and noted in the manager.

2. After the areas are set up, all areas start their own *area thread*, which functions as a container for the manager and job submissions.

3. Each area thread starts a *manager thread*, and schedules *submitter threads* to be executed. The operations now continue in the newly started threads.

4. Meanwhile, the area thread sleeps for a specific amount of time, after which it starts the shutting down process, terminating the manager thread, while allowing for a cooldown phase in which running threads finish their tasks.

5. The scheduled submitter threads are periodically started with a single mission: to create and submit a job, after which they close.

6. During this time, the manager thread continuously looks for new jobs in its queue. As soon as there is a job to be scheduled, the manager checks the capability of its area and decides to either process the job locally or to forward it to a neighboring area.

7. Once all tasks have locally been assigned to workers (inside the manager thread), a new *job execution thread* is launched, allowing the manager thread to continue its work.

8. For each task-worker combination, a *worker thread* is started, whose only job is to simulate task execution, by making the thread sleep for a certain period. As soon as the worker thread “wakes up”, i.e., the sleeping period runs out, the worker is granted its profit, and the submitter pays for this specific task.

9. After all tasks of the job are executed, the submitter pays the manager, and closes the job execution thread.

5.2.3 Threads

As already mentioned, in order to enable parallel performance of all the nodes in the network, multi-threading with special focus on synchronization was implemented. Besides one area and one manager thread per area, all other threads are short-lived, and have a very specific job to do, after which their terminate successfully. This section provides a detailed description of the responsibilities the threads in the system have.

Area Thread. Each area starts its own thread that acts as a container for all node processes. After all managers and their workers have been initialized on program startup, area threads start by calculating MC and CCN values, followed by a single manager thread launch (displayed in line 2 of Listing 5.1). Using *ScheduledExecutorService* from the *java.util.concurent* package, job submission (via submitter thread) is scheduled every one, three and ten seconds, with respective delays of zero, one, and four seconds (lines 6-8). Subsequently, the threads sleeps for the provided amount of time, before it starts the thread termination process. Line 12 initiates an orderly shutdown, allowing the previously submitted tasks to finish execution, but no new tasks will be accepted, meaning no jobs will be submitted. Subsequently, the executor blocks until all tasks have completed execution, or the timeout occurs in line 13. We then provide a cooldown phase in which the manager still has time to finish processing the queued jobs, before interrupting the manager thread in line 17 (proper error handling is done in the manager thread). With that the area thread closes. All area threads start and end at the same time.

Submitter Thread. The submitter thread is a fairly simple java class, with a straightforward function – to submit a job. It starts by retrieving a random worker node from its area, and creating a new job with a random task list. The job is persisted to the database, and added to the submitter's list of jobs, while increasing the historical term. Furthermore, the job is added to the manager's queue where it is automatically sorted. Both manager and worker node are updated in the database, and the thread closes.

This action covers, implicitly, the job prioritization process as well. A manager's job queue is defined as *PriorityBlockingQueue* of *Jobs*. This allowed us to define a comparison function, that compares the new job with the jobs already in the queue, and sorts them at once. In the *Job* class (Fig. 5.2), we define *compareTo* method with the implementation showed in Listing 5.2[1]. As discussed in Section 4.2.5, the jobs are first sorted based on the available credits of the submitter (lines 6-7). The second criterion is the historical terms of the submitter (lines 12-13), in which we prioritize the nodes that have already contributed to the system, paying for previous job executions. Lastly, in line 18, we sort the jobs based on time elapsed waiting in the queue.

[1]To keep the code part short and simple, some parts of the implementation were obscured.

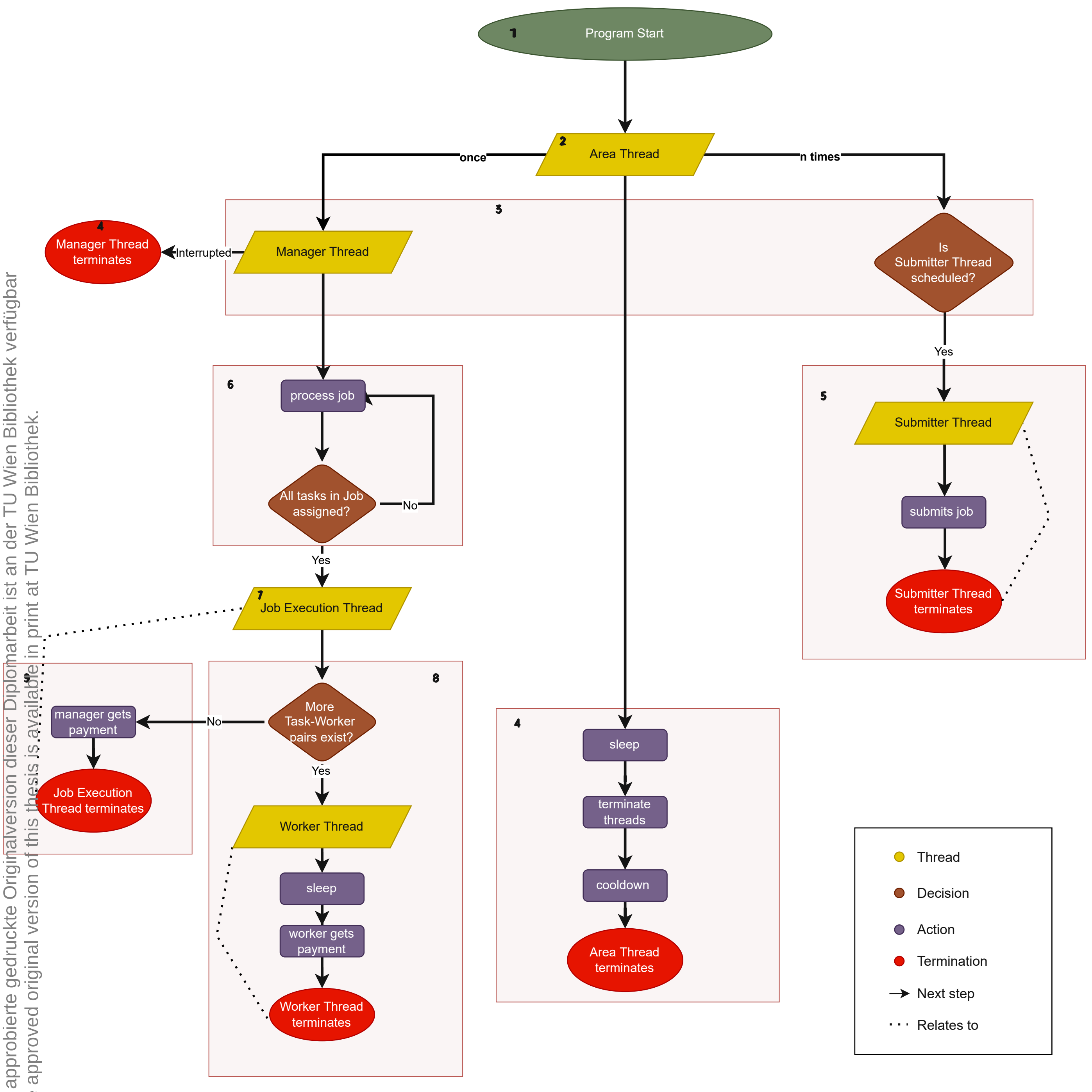

Figure 5.1: System workflow, in terms of thread operations

Listing 5.1: Partial code from AreaThread

```
ManagerThread managerThread = new ManagerThread(manager, DELTA_CONSTANT);
managerThread.start();

Runnable runnable = () -> new SubmitterThread(area.getId()).start();

executor.scheduleAtFixedRate(runnable, 0, 3, TimeUnit.SECONDS);
executor.scheduleAtFixedRate(runnable, 1, 5, TimeUnit.SECONDS);
executor.scheduleAtFixedRate(runnable, 4, 10, TimeUnit.SECONDS);

Thread.sleep(runtime * 1000);

executor.shutdown();
executor.awaitTermination(10, TimeUnit.SECONDS);

Thread.sleep(10000);

managerThread.interrupt();
```

Manager Thread. One manager thread is started per running area, which takes care of the manager operations described in the Local Scheduling Algorithm in previous chapter. The main concern of the manager is to schedule and assign tasks to workers, for which it is compensated. On thread startup, the manager enters a loop where it looks for a job in its queue. As soon as a job lands in the queue, the manager removes the job from it, and commences with processing. The first thing the manager does is to update its Scheduling Criteria (Eq. 4.4). This includes its local CC, CCN (Eq. 4.1), MC (Eq. 4.2), and reputation (Eq. 4.3). In the next step, the manager decides if it should process the job locally, i.e., does it have enough resources available, or should the job be forwarded to a neighboring area. In the latter case (left branch in Fig. 5.2), the manager with the highest Scheduling Criteria is found. Provided that is the current node, the job will be processed by the local area in the next step. Otherwise, the job is added to the job queue of the selected manager. In both cases, the thread starts from the beginning, expecting new jobs to execute.

Supposing the area can handle the job load, the manager proceeds to perform the Local Scheduling Algorithm (right branch in Fig. 5.2). After all tasks of the job have been assigned as described in Section 4.2.5, a new thread dealing with the execution is launched. The manager continues to update its values and to store them in the database, before it starts from the top, waiting for new jobs to handle.

Job Execution Thread. The job execution thread can be observed as a wrapper for job execution. It ensures all tasks are executed, and everybody gets paid after the process is finished. It also makes sure to update the manager's profit and reputation after successful termination. The thread receives a map with all tasks and their assigned workers, as well as the profit workers and the manager should receive. For each worker in the map, a new

Listing 5.2: Job comparison method

```java
@Override
public int compareTo(Job job) {
    // .... more code

    // first criterion
    if (currentHasEnoughCredits && !comparingHasEnoughCredits) return -1;
    if (!currentHasEnoughCredits && comparingHasEnoughCredits) return 1;

    // .... more code

    // second criterion
    if (currentHistoricalTerm > comparingHistoricalTerm) return -1;
    if (currentHistoricalTerm < comparingHistoricalTerm) return 1;

    // .... more code

    // third criterion
    return isCurrentBefore ? -1 : (isCurrentAfter ? 1 : 0)
}
```

worker thread is started that deals with individual task execution. The job execution thread waits for all worker threads to finish, ahead of updating the submitter's and the manager's database entries.

Worker Thread. The worker thread has a very simple job – to sleep for a certain amount of time determined by the task difficulty discussed in Section 5.2.6. After the time passes, the worker node is rewarded its profit, and submitter's credits are decreased and persisted. The thread gives signal to the job execution thread that it has finished, and closes.

5.2.4 Models

The class diagram presented in Fig. 5.3 portrays the model structure in the described system. An area contains multiple nodes – one manager and many worker nodes, both subclasses of the abstract Node class. The worker node has a list of jobs waiting for execution (all submitted jobs from this worker), whereas a manager node has a list of jobs waiting to be executed. A job has a list of tasks to be executed by the worker nodes.

5.2.5 Program Arguments

In order to make the program customizable, and to make experimentation easier, we introduced multiple program arguments that can be provided. If no explicit value is supplied for an argument, the default value is used.

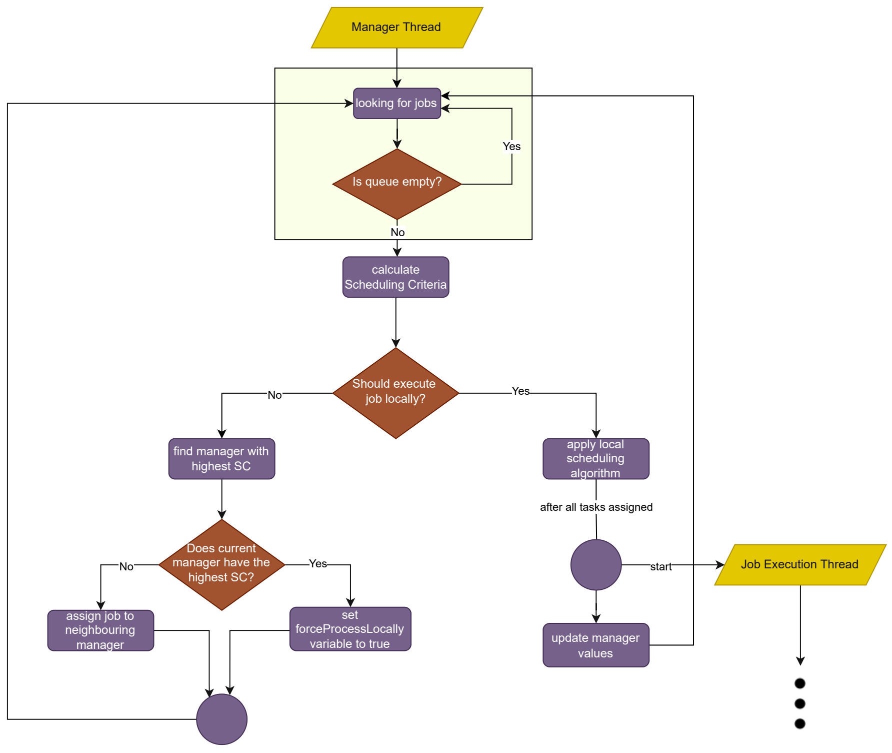
Manager Thread
looking for jobs
Is queue empty?
Yes
No
calculate Scheduling Criteria
Should execute job locally?
No
Yes
find manager with highest SC
apply local scheduling algorithm
after all tasks assigned
Does current manager have the highest SC?
No
Yes
start
Job Execution Thread
assign job to neighbouring manager
set forceProcessLocally variable to true
update manager values

Figure 5.2: ManagerThread processes in a flow chart

- Runtime: An integer that determines how long the program should run, i.e., after which time should the submitters stop launching jobs and the manager should stop listening for new jobs, given in seconds. Default value is 50.

- Delta Constant: A double value that defines the delta constant used in the Local Scheduling Algorithm (see Section 4.2.5). Default value is 0.5.

- Minimum number of workers: An integer value setting the minimum amount of workers in an area. Default is 1.

- Maximum number of workers: An integer value setting the maximum amount of workers in an area. Default is 50.

- Number of areas: An integer that determines the number of areas in the system. Default is 1.

- Number of neighboring areas: An integer value that identifies how many neighbors should each area in the network have. Default value is 0.

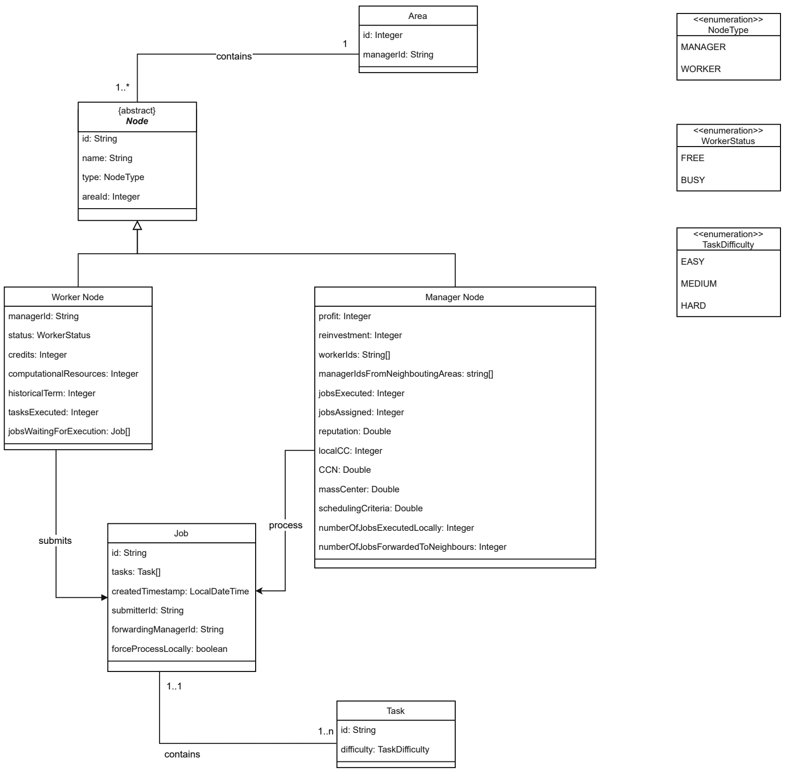

Figure 5.3: Class diagram

5.2.6 Predefined Values

To be able to run our experiments, and focus on the aspects important to the incentive mechanism proposed in this thesis, some valuation had to be predefined and/or randomly generated.

- The number of workers in each area is determined by randomly choosing a value between the (provided) minimum and maximum.
- The computational resources offered by each worker are generated randomly. The range will be experimented on in the next chapter of the thesis.
- The number of tasks in a job is selected randomly by the program. The range will be experimented on in the next chapter of the thesis.
- For simplicity, we decided to provide three different task difficulty levels: *easy*, *medium*, and *high*. Easy difficulty requires 10 computational resources for processing, and the executions of easy tasks "last" 1 second. Medium tasks require 20 resources, and require 2 seconds for execution, while hard ones "cost" 50 resources and finish in 5 seconds.

- The distances of workers in an area, used in Eq. 4.2, are also randomly elected, in the range of 100 to 300 ms.
- As already mentioned, the optimal value for α (Eq. 4.1) is 0.35, so we chose this value as a final constant.
- Considering that in our example scenario, communication plays the most important role, we decided to set β_1 from Eq. 4.4 to 0.7, while β_3, as next important aspect, is defined at 0.2, and β_2 is set to 0.1.
- The μ constant from global scheduling algorithm (Section 4.2.4) is fixed at 0.2 (as experimentation suggested).
- While looking for a free worker, to avoid endless loops, the manager thread will terminate after 10 seconds, if it does not find a free worker.

5.2.7 Problems Encountered

Unquestionably, the biggest problem during the development phase was the synchronization between threads. Considering that a large amount of threads, all writing and reading to and from the database, can be running in parallel, the problem of data overwriting, resulting in working with outdated values, was encountered. This could be resolved with **synchronized** blocks in Java, where each critical database interaction is put in one of such blocks.

Another issue occurred after area thread termination. At the point where the runtime has run out, immediate program termination resulted in many unfinished (unexecuted and unassigned) tasks/jobs. To deal with that, we introduced a cooldown phase after runtime expired, to allow threads to finish up their processes. Experimentation on this cooldown period will be addressed in the next chapter.

5.3 Critical Discussion

As is often the case with new approaches, our algorithm also has some limitations and potential for improvement. In this section, such issues will be discussed, but we will not deal with their implementation, we leave that for the future work (Section 7.2).

When a worker node submits a job for execution, it does not know how much its execution will cost. The current implementation takes tasks from the job, one by one, and executes them, for which the submitter pays. After all tasks of the submitted jobs are executed, the submitter pays the manager its processing fee. Before the job launch, the submitter can not know how much will it have to pay, since that heavily depends on the available workers in the network at the given moment, and their “prices”. An alternative would be for a submitter node to set an upper boundary representing the maximum it is willing to pay for the process. In case that is not possible within the provided range, the job would not be executed in the current area.

Another aspect that needs more attention is fault tolerance during job execution. As already stated, in the current system the submitter pays, and workers are awarded after each task execution. From the view of the workers executing the tasks, that is the desired behaviour, since they should be paid for their effort. On the other hand, observing from the submiiter's perspective, what if something happened during the execution of one of the tasks, and the job has not been processed fully? The submitter would have had already paid for certain tasks to be executed, but its request (the whole job) would have not been successfully processed. Finding a solution that would work for both workers processing and the one submitting the job, could be demanding and we leave that to future work.

CHAPTER 6

Evaluation

Testing the system in a controlled and governed environment allows us to analyse the results from different scenarios, observe the edge cases, detect any vulnerabilities and discover limitations of the system. This chapter delivers a thorough examination of the system, with regard to the mechanism behaviour and the results.

The main focus of this evaluation is to understand how the system behaves with different input data. We will analyse the number of jobs created, executed and unfinished. The focus will lie on the number of tasks each node executes, as well as the profit it receives. We will discuss the main causes for node contribution and try to find the reasons for the lack of it. Communication across multiple areas, including job forwarding will also be a debated topic.

Given that the general evaluation setup was already explained in the last chapter (see Section 5.2.1), the focus of this section lies on the experimentation scenarios. We will begin the analysis with a lower number of workers, tasks, runtime etc., and throughout the assessment, these numbers will increase, trying to find the limitations of the system.

6.1 Single Area

We will first consider the scenario where only one area exists. Such situation would be relevant in a smaller Fog network, in which case we do not expect too many nodes to join, or too much traffic to be generated, so one area, i.e., one manager to handle the load, would suffice.

Table 6.1: Job collection after program termination[1]

_id	submitterId	forceProcessLocally	numberOfTasks
6233ab86b881ed3dec9aca3f	6233ab85b881ed3dec9aca18	false	4
6233ab88b881ed3dec9aca40	6233ab85b881ed3dec9aca2d	false	3
6233ab89b881ed3dec9aca41	6233ab85b881ed3dec9aca32	false	2
6233ab8ab881ed3dec9aca42	6233ab85b881ed3dec9aca18	false	3
6233ab8cb881ed3dec9aca43	6233ab85b881ed3dec9aca21	false	4
6233ab8db881ed3dec9aca44	6233ab85b881ed3dec9aca25	false	3
6233ab8fb881ed3dec9aca45	6233ab85b881ed3dec9aca1c	false	3
6233ab92b881ed3dec9aca46	6233ab85b881ed3dec9aca21	false	3
6233ab92b881ed3dec9aca47	6233ab85b881ed3dec9aca26	false	3
6233ab94b881ed3dec9aca48	6233ab85b881ed3dec9aca37	false	2
6233ab95b881ed3dec9aca49	6233ab85b881ed3dec9aca32	false	2
6233ab97b881ed3dec9aca4a	6233ab85b881ed3dec9aca20	false	4
6233ab98b881ed3dec9aca4b	6233ab85b881ed3dec9aca1c	false	3
6233ab9bb881ed3dec9aca4c	6233ab85b881ed3dec9aca39	false	3
6233ab9cb881ed3dec9aca4d	6233ab85b881ed3dec9aca29	false	1
6233ab9eb881ed3dec9aca4e	6233ab85b881ed3dec9aca27	false	1
6233ab9eb881ed3dec9aca4f	6233ab85b881ed3dec9aca2f	false	1
6233aba1b881ed3dec9aca50	6233ab85b881ed3dec9aca1e	false	3
6233aba1b881ed3dec9aca51	6233ab85b881ed3dec9aca35	false	4
6233aba4b881ed3dec9aca52	6233ab85b881ed3dec9aca26	false	3

The program is started with the following arguments:

```
    --runtime=30 --minWorkers=5 --maxWorkers=50 --delta=0.5
```

stating that we want it to run for 30 seconds (+ 10 seconds cooldown period). The area should have a random number of workers between 5 and 50, and the delta constant should be set to 0.5, meaning half of the credits earned by the manager will be set aside for the reinvestment, and the other half are the profit.

6.1.1 Jobs

During the 30 seconds of runtime, the area thread has scheduled and launched 20 submitter threads. The same number of jobs has been created and stored in our database (see Table 6.1), summing up to a total of 55 tasks. The field *forceProcessLocally* is always false, since we have only one area, and in no case would we even consider forwarding a job to the neighboring area, meaning no forcing is needed either. Each job contains an id of its, randomly selected, submitter. In addition, the list of randomly generated tasks for each job is saved. In this scenario, there could be between one and five tasks per job. These tasks can be either easy, medium or difficult and accordingly need different times and computational resources for their executions (see Section 5.2.6).

[1]For simplicity, some fields were obscured, and instead of showing all task objects, only the number of tasks in each job is displayed.

Listing 6.1: Manager values after 30 seconds runtime

```
{
    "name": "Manager 1",
    "profit": 23953,
    "reinvestment": 23953,
    "jobsExecuted": 20,
    "jobsAssigned": 20,
    "numberOfJobsExecutedLocally": 20,
    "numberOfJobsForwardedToNeighbours": 0,
    "queue": [],
    ...
}
```

6.1.2 Manager Node

After each job submission, assignment and execution, the manager is updated. Listing 6.1 shows the values, the manager holds at the program end[2]. Considering that the profit and reinvestment were equally divided ($\delta = 0.5$), it is not surprising that the profit and the reinvestment have the same value, in this case – 23953[3]. That is the amount the manager has earned while scheduling tasks. Taking into account that only one manager exists in the system, no calculations for the Global Scheduling Algorithm (described in Section 4.2.4) such as CCN, MC etc., are required, the manager has no neighboring areas, and the number of forwarded jobs stays at zero. There are no jobs in the manager queue, meaning all submitted jobs were assigned to workers. By implementing a 10 seconds cooling phase, we allowed all assigned tasks to finish their execution, so there are no open processes.

6.1.3 Worker Node

In this test run, 41 workers were generated, with a random number of computational resources between 5 and 1000 (Fig. 6.1), whereby values between 9 and 978 were assigned (see Fig. 6.2). The same graph provides a quick overview of characteristics[4] of the credit distribution among workers in the system. Figure 6.3 shows the same information in terms of the number of jobs submitted, and the number of tasks executed[5]. Following

[2]Only data relevant to this scenario are shown, the rest is obscured.

[3]In our implementation, we do not implement a reinvestment policy. This can be user-defined, and modified to user desires and use case. Our experiment serves to show how much does the manager have at its disposal to use for a reinvestment strategy.

[4]By characteristics, we mean: the median, the lower quartile, the upper quartile, the lower whisker and the upper whisker [27].

[5]The only reason these values are separated into two graphs is the high variation in value, which would not allow data to be interpreted in a meaningful way if merged into a single graph.

the system termination, no node had jobs waiting for execution, demonstrating that all jobs assigned by the manager were indeed executed, and all workers were free.

Table 6.2 reveals a more detailed worker information, restricting to those fields significant for our current discussion. In it, we can see how many computational resources each node has, how many jobs it submitted (*historicalTerm*), how many credits it has earned, and the amount of the tasks it executed. The nodes are sorted based on the number of executed tasks, descending.

Right at first glance, we can notice that there are two workers that executed the most tasks (each 9 tasks), one of which, *Worker 1.23*, has also earned by far the most credits (933). The reason lies in the fact that this particular worker has the lowest "price" for its computational resources – 14, excluding *Worker 1.38* that does not have enough resources to execute any task[6]. That is five times lower than the next lowest value of a worker – 70. Since our algorithm is programmed to always look for the cheapest option, it will always select the Worker 1.23, so long it is available. Bearing in mind that this node has only 14 resources, it is important to mention that it can only process the tasks with difficulty level *easy*[7]. There was a total number of 13 easy tasks submitted to the system, and 9 from those were assigned to and processed by the Worker 1.23. Considering that easy tasks are processed in one second, this worker was quickly available to take up a new task, which is how it manager to execute so many of them.

We can also notice that there are a couple of workers which, even though they participated in the task execution, wound up having very little (potentially zero) credits at the end.

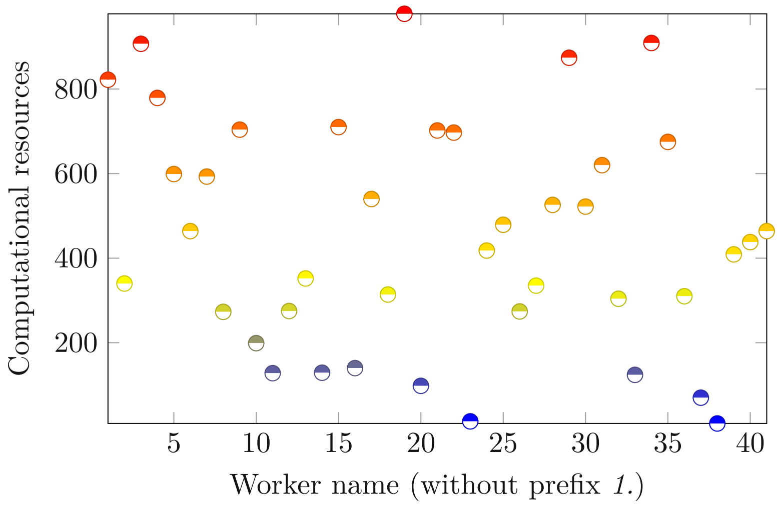

Figure 6.1: Randomly generated computational resources for the workers, displayed as a scatter plot

[6]The simplest task requires 10 computational resources (see Section 5.2.6)
[7]Easy task require 10, medium 15 computational resources (as explained in Section 5.2.6).

Table 6.2: Partial excerpt from worker node collection after program termination, sorted on the number of executed tasks

_id	name	resources	historicalTerm	credits	tasksExecuted
6233ab85b881ed3dec9aca29	Worker 1.20	98	1	86	9
6233ab85b881ed3dec9aca2c	Worker 1.23	14	0	933	9
6233ab85b881ed3dec9aca20	Worker 1.11	128	1	3	8
6233ab85b881ed3dec9aca3a	Worker 1.37	70	0	276	8
6233ab85b881ed3dec9aca36	Worker 1.33	124	0	109	6
6233ab85b881ed3dec9aca25	Worker 1.16	140	1	177	5
6233ab85b881ed3dec9aca1f	Worker 1.10	199	0	297	4
6233ab85b881ed3dec9aca23	Worker 1.14	129	0	44	4
6233ab85b881ed3dec9aca1d	Worker 1.8	273	0	2	2
6233ab85b881ed3dec9aca16	Worker 1.1	822	0	0	0
6233ab85b881ed3dec9aca17	Worker 1.2	340	0	0	0
6233ab85b881ed3dec9aca18	Worker 1.3	907	2	0	0
6233ab85b881ed3dec9aca19	Worker 1.4	779	0	0	0
6233ab85b881ed3dec9aca1a	Worker 1.5	599	0	0	0
6233ab85b881ed3dec9aca1b	Worker 1.6	464	0	0	0
6233ab85b881ed3dec9aca1c	Worker 1.7	593	2	0	0
6233ab85b881ed3dec9aca1e	Worker 1.9	704	1	0	0
6233ab85b881ed3dec9aca21	Worker 1.12	275	2	0	0
6233ab85b881ed3dec9aca22	Worker 1.13	352	0	0	0
6233ab85b881ed3dec9aca24	Worker 1.15	710	0	0	0
6233ab85b881ed3dec9aca26	Worker 1.17	540	2	0	0
6233ab85b881ed3dec9aca27	Worker 1.18	314	1	0	0
6233ab85b881ed3dec9aca28	Worker 1.19	978	0	0	0
6233ab85b881ed3dec9aca2a	Worker 1.21	702	0	0	0
6233ab85b881ed3dec9aca2b	Worker 1.22	697	0	0	0
6233ab85b881ed3dec9aca2d	Worker 1.24	418	1	0	0
6233ab85b881ed3dec9aca2e	Worker 1.25	479	0	0	0
6233ab85b881ed3dec9aca2f	Worker 1.26	274	1	0	0
6233ab85b881ed3dec9aca30	Worker 1.27	335	0	0	0
6233ab85b881ed3dec9aca31	Worker 1.28	526	0	0	0
6233ab85b881ed3dec9aca32	Worker 1.29	874	2	0	0
6233ab85b881ed3dec9aca33	Worker 1.30	522	0	0	0
6233ab85b881ed3dec9aca34	Worker 1.31	620	0	0	0
6233ab85b881ed3dec9aca35	Worker 1.32	304	1	0	0
6233ab85b881ed3dec9aca37	Worker 1.34	909	1	0	0
6233ab85b881ed3dec9aca38	Worker 1.35	675	0	0	0
6233ab85b881ed3dec9aca39	Worker 1.36	310	1	0	0
6233ab85b881ed3dec9aca3b	Worker 1.38	9	0	0	0
6233ab85b881ed3dec9aca3c	Worker 1.39	409	0	0	0
6233ab85b881ed3dec9aca3d	Worker 1.40	438	0	0	0
6233ab85b881ed3dec9aca3e	Worker 1.41	464	0	0	0

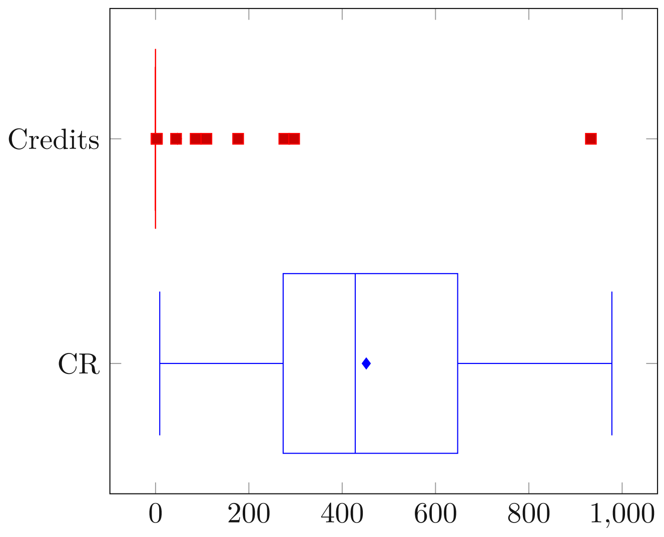

Figure 6.2: Information on computational resources (CR) and credits distribution

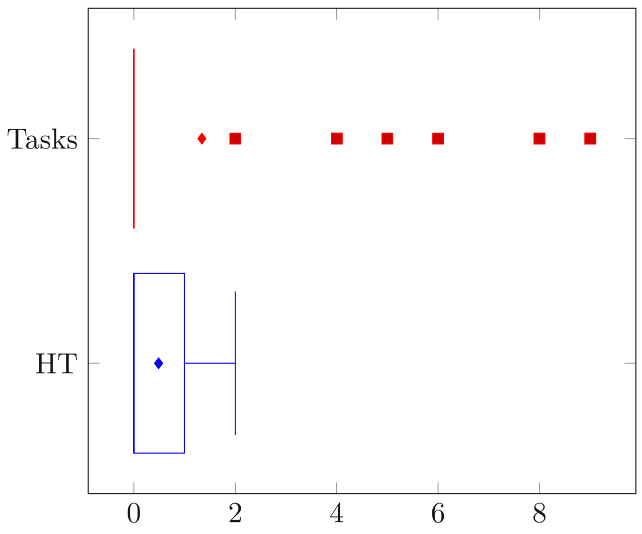

Figure 6.3: Information on historical term (HT) and executed tasks distribution

These are usually workers that have submitted jobs for execution. This means that the execution of the jobs they submitted costed almost as many credits as they earned for their task execution. However, even if they currently posses little credits (or potentially none), their historical term (the number of submitted jobs) is higher than in some other nodes, meaning that their job would be prioritized compared to the jobs of the nodes which also do not have enough credits to pay for it, but have not submitted a job either.

Out of 41 workers introduced in the system, 9 contributed to the network by processing tasks[8], which is around 22% of all the workers. Put another way, the average number of executed tasks is 1.3 (the diamond mark in 6.3). The same 22% percent of the workers have a credit balance higher than zero. These numbers come as a result of launching 20 jobs (total of 55 tasks) in an area of 41 workers, where jobs are submitted periodically, and task execution has a relatively short duration, allowing the cheap workers to quickly finish their tasks, and offer their resources in the new scheduling phase. An area in which more processing power is required, or similarly, the tasks were more complex (demand longer processing), would result in deployment of more different workers.

Another element worth mentioning is the historical term[9], and its dependency on the credits quantity. We can observe that workers which have higher historical terms, usually have less credits (or none). This is justified by the fact that job submissions are expensive. Thus, to submit more jobs, nodes need to execute more tasks and earn more credits,

[8]Technically, job submissions are also considered contribution to the network, but are not our focus in the current analysis.

[9]Historical term value analysis per se will not take place, considering it is a result of a random selection process.

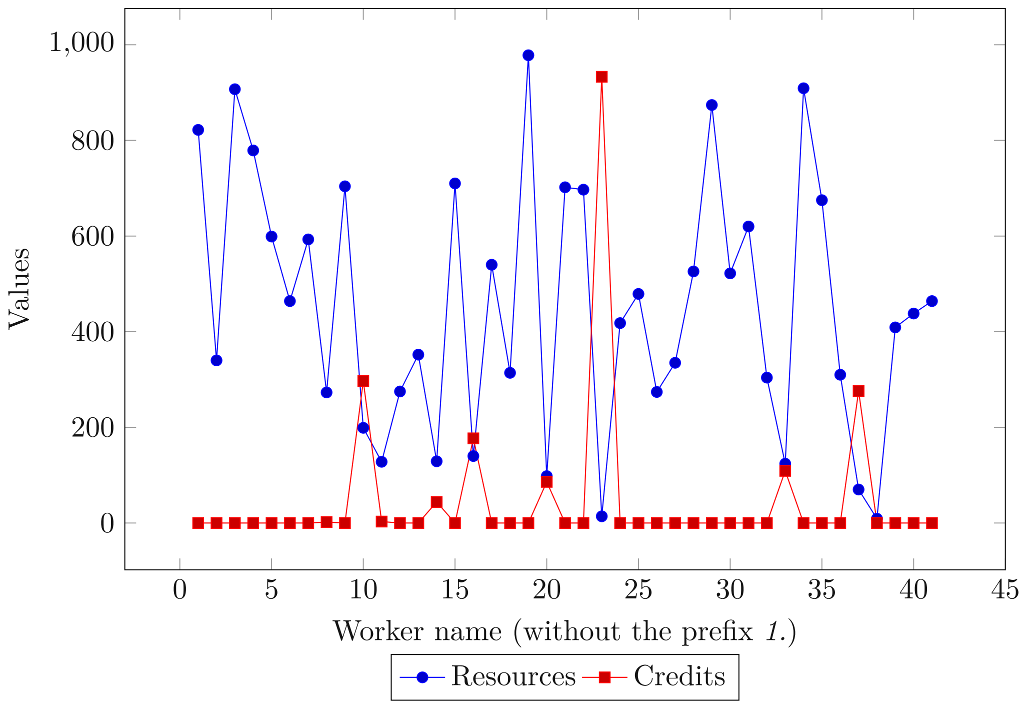

Figure 6.4: Graphical representation of credits received compared to computational resource of nodes after simulation termination

contributing further to the system, and achieving exactly that what the main goal of this thesis is.

In general, it can be concluded that nodes with lower computational resources were selected more often for task execution, compared to the workers with higher computational resources. Since the work load in the system was not very high, meaning not all nodes had to be employed, half of the nodes (those with the higher resource prices) were never selected in the assignment process. This is nicely visualized in Figure 6.4. We can see that, the higher the blue dots are (price of the resources), the lower the red squares are (credits earned), and vice versa. As already mentioned, in the case were we would have more tasks (or less workers) in an area, a higher percentage of workers would be engaged, regardless of their price. This will be observed and discussed in the next evaluation scenario.

6.2 Single Area with High Workload

In the previous scenario, 20 jobs were launched in an area with 41 associated workers. That would count as an area with pretty low workload, which is also the reason we had a lot of nodes that did not get the chance to participate in the task execution. In this simulation we will analyze an area with a higher volume of work, by reducing the amount

Listing 6.2: Manager values after 30 seconds

```
{
    "name": "Manager 1",
    "profit": 7262,
    "reinvestment": 7262,
    "jobsExecuted": 20,
    "jobsAssigned": 20,
    "numberOfJobsExecutedLocally": 20,
    "numberOfJobsForwardedToNeighbours": 0,
    "queue": [],
    ...
}
```

of available workers in the area, while maintaining the number of submitted jobs. The same could be achieved by preserving the number of workers, and increasing the number of jobs and/or tasks instead.

This time, we want to have a maximum number of 20 workers, and will allow a minimum of two workers. The delta variable and the runtime will stay the same:

```
    --runtime=30 --minWorkers=2 --maxWorkers=20 --delta=0.5
```

6.2.1 Jobs

In the course of those 30 seconds, 20 jobs were submitted, each with one to five tasks, same as in the previous test case. These 20 jobs generated 53 tasks in total. Being that, once again, we have only one area, no force execution is done, and the *forceProcessLocally* field is false in all the jobs.

6.2.2 Manager Node

Listing 6.2 shows manager values at the end of program execution. The manager has once more accomplished to schedule all submitted jobs, so there were none left in the queue. We are still dealing with only one area, therefore all the jobs were executed locally. The manager profit has significantly changed to that of the previous case – 7262, compared to 23953, which is the result of the lower range of computational resources offered by the nodes in this scenario, and will be discussed further.

6.2.3 Worker Node

Worker collection looks a lot different this time around. Only seven workers were generated to execute 53 tasks, resulting in more (all!) nodes being involved in the system operations. Same as before, graphs in Fig. 6.5 and Fig. 6.6 provide an overview of value distribution

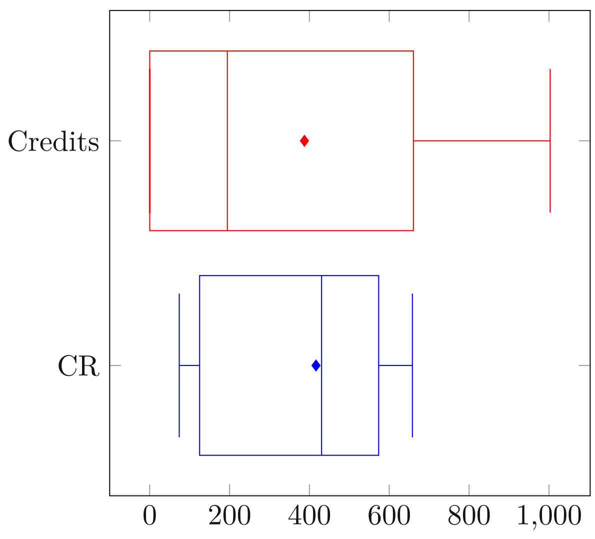

Figure 6.5: Information on computational resources (CR) and credits distribution

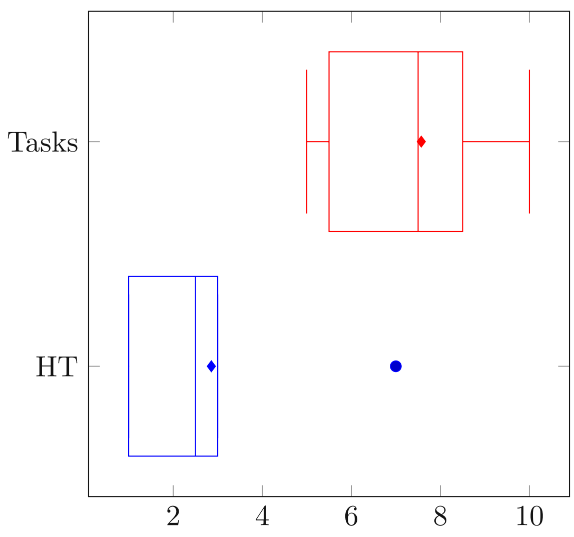

Figure 6.6: Information on historical term (HT) and executed tasks distribution

regarding computational resources, credits earned, jobs submitted and tasks executed, respectively. By comparing Fig. 6.4 with Fig. 6.1 from the previous test run, we can understand why was there such a big difference between the profits of the two managers. A more restricted computational resource range (as is the case in Fig. 6.4), means the manager earns a lower profit[10]. The gap between the average and the maximum earned credits was lowered, since we do not have so many nodes with zero credits. Another detail that stands out in these graphs, compared to the ones from the last example, is that the average number of the executed tasks amounts to 7.6 – much higher than 1.3 from the previous scenario.

Table 6.3 enables us to examine the results of worker nodes more thoroughly. From it, we can conclude that *Worker 1.7* executed the most tasks, 10 to be precise. However, in this instance, unlike in the last run, the worker that executed the most tasks is not the worker with the most credits. *Worker 1.6* has earned 1003 credits within 30 seconds. The reason it has so many credits is seen in the fact that it has submitted only 1 job, in contrast to nodes higher up in the table. As an example, *Worker 1.4* has processed nine tasks, but launched seven jobs for execution, which costed more that it received for the completed tasks. Nevertheless, we have a great example of a successful incentive mechanism in *Worker 1.7*. This node executed 10 tasks, earning enough credits to have its jobs executed, leaving the rest for later job submissions.

A visual representation of the credits earned put into relation with resource prices is portrayed in Fig. 6.7.

[10]The manager is rewarded the difference between the highest computational price and the price of the selected worker node (see Section 4.2.5)

Table 6.3: Partial excerpt from worker node collection after program termination, sorted by the number of executed tasks

_id	name	resources	historicalTerm	credits	tasksExecuted
6233a291b89b3259d17e829b	Worker 1.7	176	3	814	10
6233a291b89b3259d17e8298	Worker 1.4	605	7	0	9
6233a291b89b3259d17e8297	Worker 1.3	74	3	507	8
6233a291b89b3259d17e829a	Worker 1.6	363	1	1003	8
6233a291b89b3259d17e8296	Worker 1.2	498	2	195	7
6233a291b89b3259d17e8299	Worker 1.5	542	1	0	6
6233a291b89b3259d17e8295	Worker 1.1	658	3	194	5

This test run delivered 100% node contribution. All nodes were meaningful supporters of the system. In fact, on several occasions, all worker nodes in the network were busy, so the manager had to wait for a worker to finish its task execution in order to assign a new task. This is where the cooldown phase shows useful. Jobs are constantly being submitted, and tasks executed. However the manager can not assign them immediately (either busy with the assignment of the previous job, or no workers are free), so the jobs accumulate. By giving the system participants another 10 seconds after the jobs have stopped submitting, we give them a chance to finish their processes – task scheduling and execution.

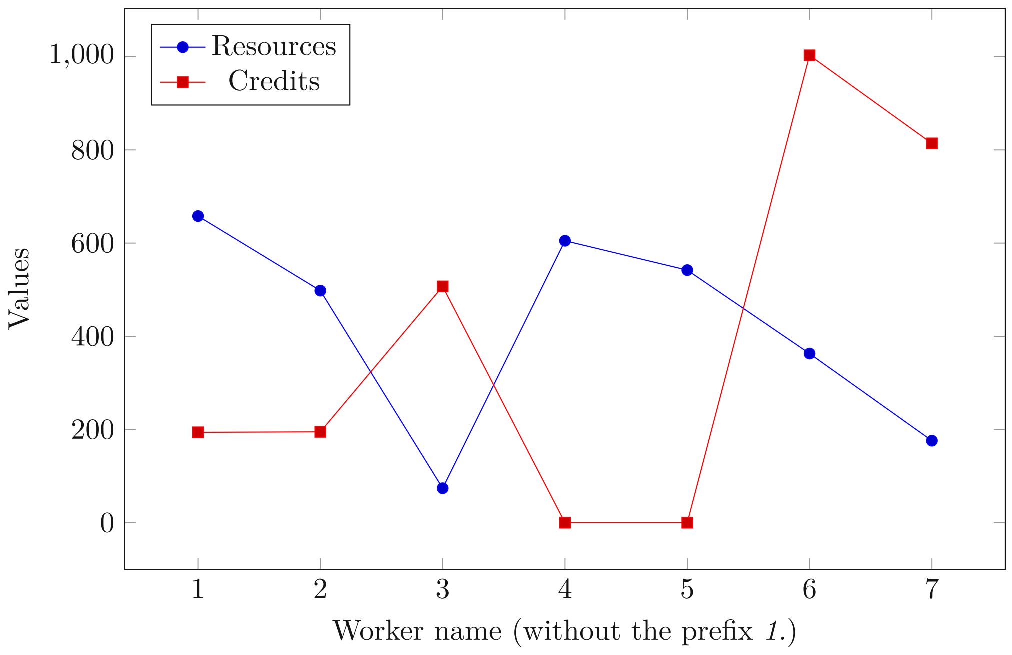

Figure 6.7: Graphical representation of credits received compared to computational resource of nodes after simulation termination

6.3 Multiple Areas

This step of the evaluation process will define three areas, with interconnected managers (two neighbors each), creating a scenario in which one area has considerably less computational capacity, and will have to forward its more demanding jobs to a neighboring manager.

The following parameters are used to start the program:

```
--runtime=30 --minWorkers=5 --maxWorkers=50 --areas=3 --neighbours=2
```

Within 30 seconds of runtime, 60 jobs were submitted to three different areas. Not all of these jobs have finished their execution, and not all were processed in the area they were submitted to.

6.3.1 Manager Node

Listing 6.3 displays manager values for all created areas upon program termination. In *Area 1*, 12 workers were generated, *Area 2* contains 20 workers, while *Area 3* has 47 available workers. Consequently, *Manager 1* has executed nine jobs locally (from the 19 jobs that were submitted in Area 1), with another ten being forwarded to a neighboring area. The manager of the second area had enough capabilities to handle all of its 19 jobs locally, while Area 3 assigned and processed 29 jobs in total (19 own, and 10 forwarded).

The profit earned by the managers reflects the number of executed tasks in its area. The manager that locally assigned the least credits, was also awarded with the lowest profit[11]. The manager of the area in which 29 tasks were successfully handled, has accordingly received the biggest payment.

Remembering the check whether the job should be processed locally from the Global Scheduling Algorithm in Section 4.2.4, and the decision to set μ constant to 0.2 in Section 5.2.6, it comes at no surprise that Area 1 can only schedule jobs with one or two tasks. All the submitted jobs that have a higher number of tasks, were forwarded to the neighboring manager with the highest Scheduling Criteria. In our scenario that is *Manager 3*. In Listings 6.3a, 6.3b, 6.3c, Scheduling Criteria, along side values needed for its calculation, of each area are displayed. It can be quickly concluded that the area with the highest amount of workers, and with that highest local computing capacity, has also the highest Scheduling Criteria.

As long as all assigned jobs are successfully accomplished (such is the case in our test), the reputation of the managers will stay at one, and the queues empty. Decreasing the cooldown period would most probably result in unfinished jobs, with the reputation lower than one, and jobs waiting in the queue.

[11] Job forwarding does not result in a monetary reward.

Listing 6.3: Manager values

(a) Manager of area 1 upon program termination

```json
{
    "areaId": 1,
    "name": "Manager 1",
    "profit": 4401,
    "reinvestment": 4401,
    "workerIds": [12],
    "managerIdsFromNeighbouringAreas": [2],
    "jobsExecuted": 9,
    "jobsAssigned": 9,
    "reputation": 1,
    "localComputationalCapacity": 5228,
    "computedCapacityWithNeighbours": 12617.525,
    "massCenter": 47.25092640014461,
    "schedulingCriteria": 0.17734994416453057,
    "numberOfJobsExecutedLocally": 9,
    "numberOfJobsForwardedToNeighbours": 10,
    "queue": [],
}
```

(b) Manager of area 2 upon program termination

```json
{
    "areaId": 2,
    "name": "Manager 2",
    "profit": 15583,
    "reinvestment": 15583,
    "workerIds": [20],
    "jobsExecuted": 19,
    "jobsAssigned": 19,
    "reputation": 1,
    "localComputationalCapacity": 10569,
    "computedCapacityWithNeighbours": 12730.575,
    "massCenter": 114.09910396199935,
    "schedulingCriteria": 0.2777049337212561,
    "numberOfJobsExecutedLocally": 19,
    "numberOfJobsForwardedToNeighbours": 0,
    "queue": [],
}
```

(c) Manager of area 3 upon program termination

```json
{
    "areaId": 3,
    "name": "Manager 3",
    "profit": 32550,
    "reinvestment": 32550,
    "workerIds": [47],
    "managerIdsFromNeighbouringAreas": [2],
    "jobsExecuted": 29,
    "jobsAssigned": 29,
    "reputation": 1,
    "localComputationalCapacity": 22918,
    "computedCapacityWithNeighbours": 13155.325,
    "massCenter": 435.23453820345634,
    "schedulingCriteria": 1.008691296711742,
    "numberOfJobsExecutedLocally": 29,
    "numberOfJobsForwardedToNeighbours": 0,
    "queue": [0],
}
```

Table 6.4: Percentage of contributing nodes in the whole system

	task execution in %	positive credit balance in %	job submission in %
Area 1	25	8	75
Area 2	45	40	70
Area 3	34	32	36

6.3.2 Worker Node

Figures 6.8, 6.9, 6.10, and 6.11 depict the most important worker information for all three areas, upon simulation termination. Already, in the first graph, representing the credits earned (upper-left), we can see an immense discrepancy between the credits earned in Area 1 compared to those earned in Area 3. This occurrence can be explained by the fact that only one single worker, out of the 12 in the area, owns credits after the 30 seconds of runtime. In total, three workers with the lowest computational resources have participated in the execution of the nine jobs processed in Area 1. The rest was forwarded to other areas. The worker that executed the most tasks in Area 1 (6) is the only one with a positive credit balance. At the same time, that is also the worker with the lowest computational resources of its area. Area 2 occupies the middle ground between the two extremes. Its average lies at 64 credits, with a maximum of 308 credits, and exactly 40% nodes with positive credit balance. The worker that has the maximum credits in Area 3 (949) is the worker node with 17 computational resources, and 12 executed tasks. Once more, we have a situation where a node is much cheaper than the rest, and it quickly handles the appointed tasks, rapidly rejoining the assignment process as a free node. In Area 3, around 32% percentage of the workers were rewarded credits.

Computational resources were relatively evenly dispersed across the system, which is why they did not play a big role in the tasks assignment, or the payment process. The average amount of executed tasks in the area is pretty similar in all three areas, and spans between 0.9 (in Area 1) and 2 (in Area 2). By looking at the provided graphs, a relationship between the executed tasks and the final credit balance of nodes can clearly be derived. Another value that directly affects the amount of credits earned is the historical term. On average, every node submitted 1.6 jobs in the Area 1, while that number was significantly lower in the Area 3 – 0.4. Since each of those job submission cost credits, it explains the low credit balances in Area 1.

Table 6.4 provides a quick overview of the nodes contributing to the network. It shows how many node from each area executed a task, finished the simulation with a positive credit balance, and submitted a task, respectively, showed in percentage.

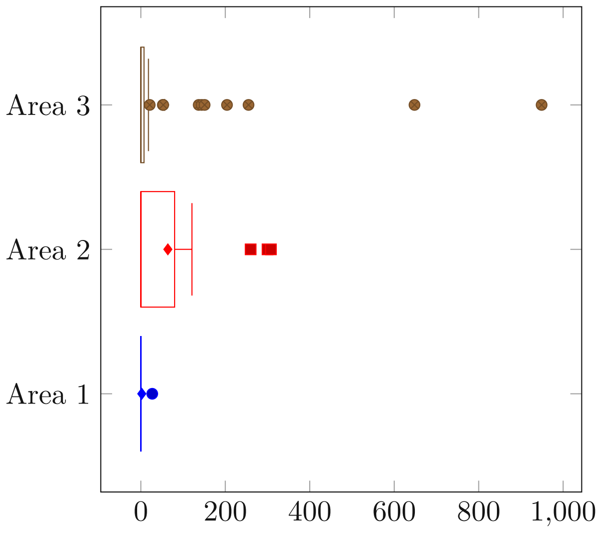

Figure 6.8: Distribution of credits in all three areas

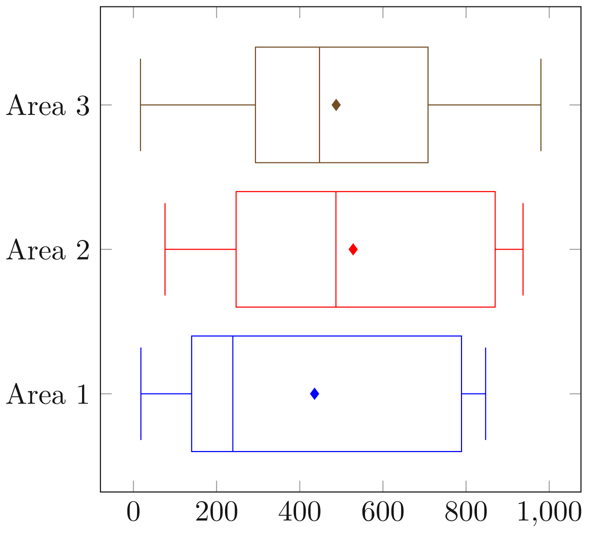

Figure 6.9: Distribution of computational resources in all three areas

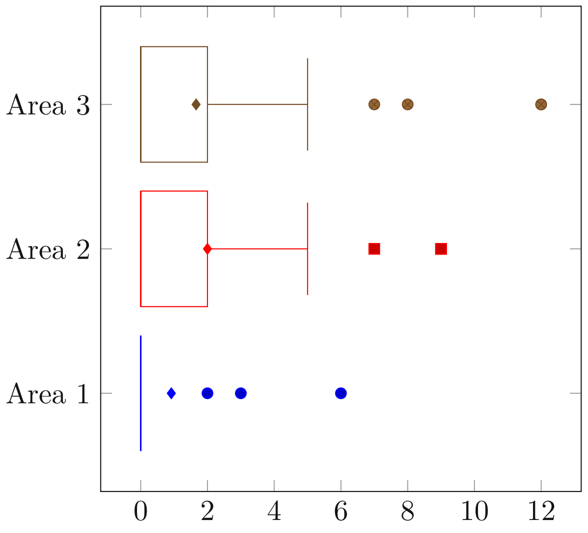

Figure 6.10: Distribution of the number of executed tasks in all three areas

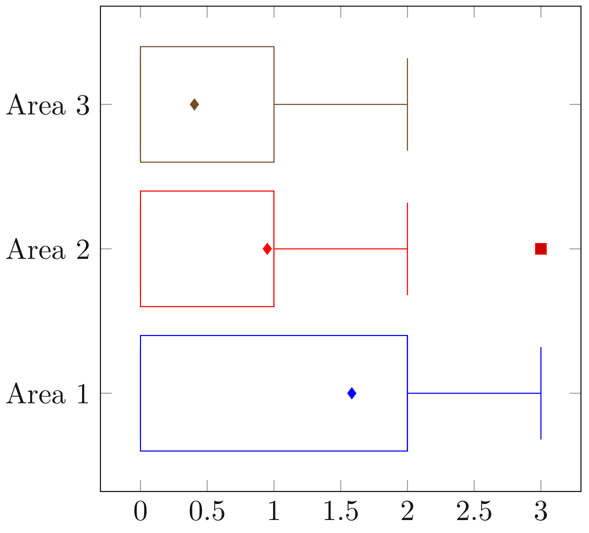

Figure 6.11: Distribution of the historical term in all three areas

6.4 Benchmarking

The previous test cases covered all the main ideas of the implemented incentive mechanism. This section serves to expand on this analysis, and to scale up the system, observing the behaviour of the program.

For the next test, we wanted to run the simulation with 30 areas. After program termination we, realized that current parameters set in the program, were not optimized for large number of areas. Namely, 30 areas assigned 908 workers (10 to 50 workers per area), and submitted 600 jobs. After 30 seconds of runtime, approximately 4.4 jobs were executed by each area, while an average of 16 did not get the chance to be assigned. The source of such behaviour is found in synchronization. We have quite a lot of threads running at the same time, all accessing database information, and processing those in parallel. To make sure that we always have the most current state of the database, and to prevent database data from being overwritten, for every value change, we first have to retrieve the newest data from the database, modify it and then persist it. These sets of instructions in Java have to be implemented in a synchronized block. This makes sure that other threads have to wait to access the data, until the current one is done using it. When we have 30 manager-, 600 submission-, and many job execution threads, all accessing the manager database collection it can take some time to process all the information.

6.4.1 Reducing the Number of Submitted Jobs

One way to deal with this problem is to reduce the workload of the system. Our first try was to reduce the number of the jobs being submitted. By removing line 7 in Listing 5.1[12], we reduced the number of threads interacting with the database. In doing so, we managed to decrease the average number of unfinished jobs from 16 to 5, and to increase the amount of successfully executed jobs from 4.4 to 9.5.

This time, 911 workers were available to processing 419 jobs. Fig. 6.12 represents the variations of manager profits across the areas, while Figure 6.13 shows a more detailed comparison of the area metrics. Looking at it, we can confirm our previous hypothesis – the node contribution depends heavily on the area workload. In the areas where more nodes are associated (with the same job workload), the percentage of worker node contribution to the system is lower, than in those area where less workers are connected[13].

6.4.2 Incresing the Cooldown Phase

An alternative to dealing with a big workload is to enable for a longer cooldown period, to give the nodes enough time to carry out the expected operations. Leaving the reduced

[12]This line schedules job submission every 5 seconds, by starting a new SubmitterThread.

[13]Areas 4 and 27, where number of executed jobs is zero is a special occurrence, where an area is too small to execute its jobs.

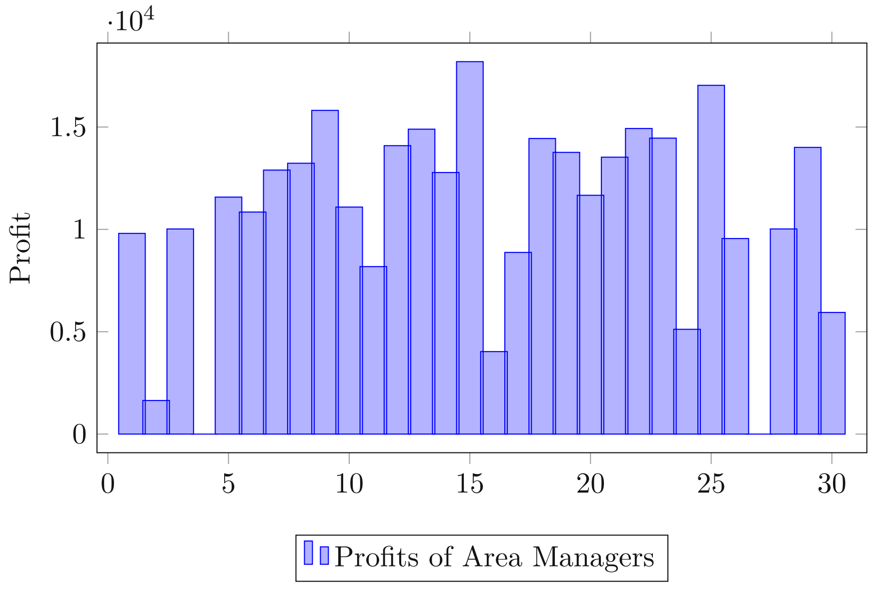

Figure 6.12: Profits of the manager in each area

number of jobs submitted (done in the previous test), in this run through we will, additionally, allow for a 30 seconds cooling period (instead of 10).

This test allowed for 100 areas, each with two neighboring areas, to generate 5435 worker nodes (5 to 100 per area), and to submit 1390 jobs. Observing the result of this simulation, we can establish that the implemented program reached its limits. In the defined 30 seconds runtime, plus the 30 seconds cooldown phase, managers of the 100 areas executed *1.56* jobs on average, while *12.97* jobs were left waiting in the queue. The explanation for this occurrence can be found in the the above-mentioned synchronization issue – too many threads required database interactions, and were blocking job scheduling.

6.5 Discussion

Experimentation scenarios ran and analyzed in this chapter helped us to elucidate program limitations, and get a better understanding of how the system behaves in certain scenarios. In the course of this evaluation, we managed to identify major factors in node participation, and those that prevent a node from being a contributing part of the network.

Per Algorithm 4.3, outlined in Section 4.2.5, the proposed incentive mechanism always chooses the worker node with the lowest price for its computational resources. That could undoubtedly be seen in the executed test cases. Those workers usually only execute the easiest tasks (earning the least profit), but are finished faster, and quickly ready for a new task assignment, not allowing other nodes with higher computational resources to be elected.

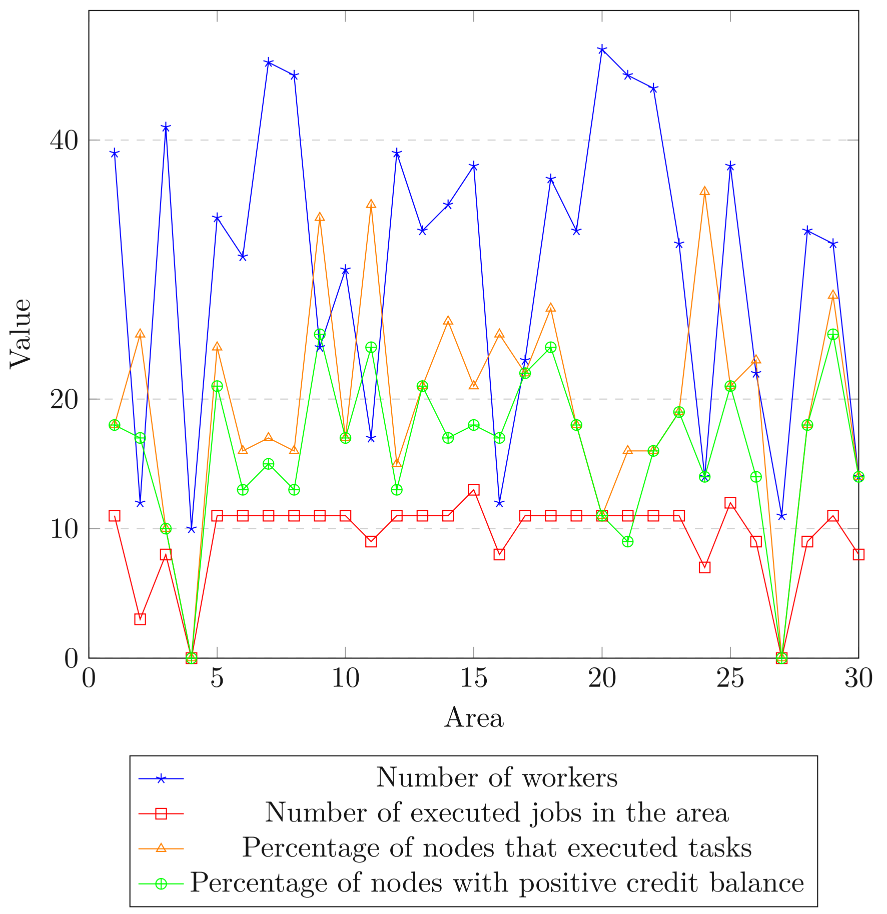

Figure 6.13: Graphical representation of the final value state in all 30 areas

The performed simulations revealed that the node engagement in task execution depends heavily on the workload of an area. In the areas where the ratio of the submitted jobs (i.e., tasks) and their respective worker nodes was lower, the involvement of the nodes in the system processes was much higher. Higher workload, and consequently higher node involvement can be achieved in four ways:

- Reduced number of the workers in an area, for the same number of jobs
- Increased number of the submitted jobs, for the same amount of worker nodes
- Increased number of the job tasks, for the same amount of workers and jobs
- Increased execution time of the tasks, for the same amount of workers, jobs and job tasks

All of these possibilities achieve the same thing – worker nodes with low computational resources are busy, so we need to select the ones with the higher values. In this case, our mechanism does not care about the price[14] – the task has to be executed.

Our second test case showed how the reduced number of workers in an area increased the number of node participation in task execution. Having more jobs, i.e., tasks to execute, more workers need to be assigned. Since our algorithm considers only free nodes during the task assignment, the nodes with higher resources would also need to be selected.

In this chapter, we also discussed the credits the worker nodes earn for successful task execution. These credits represent a currency for our system, considering that a job submission is paid with those credits. All nodes that execute a task receive a specific amount of credits. Hence, the more tasks a node executes, the more credits it will earn. However, if a node submits a lot of jobs during its runtime, as we have already considered, it can happen that at the program termination, it has very little or none credits available. This is the desired behaviour, since that means that the node has contributed to the system by submitting jobs, allowing for other workers to execute those tasks. In addition, those nodes now have a high historical term, meaning their next submitted jobs will be highly prioritized.

Furthermore, although out of scope in this thesis, an additional motivation for the worker nodes can be implemented, awarding the cooperating nodes more notably. This can be achieved by implementing a user-defined reinvestment policy that would take a part of the manager's earnings and reinvest them into the system, increasing the profit of the execution nodes.

From the conducted analysis, we can additionally recognize that there are some potential scalability issues, that will be discussed in the future work (see Section 7.2).

[14]Free workers are still ordered and selected by increasing computational price.

CHAPTER 7

Conclusion

7.1 Summary

The IoT refers to the ever-growing network of interconnected devices made accessible via the Internet. In the past decade, these devices found application in various domains, constantly increasing the requirements for their employment. This is how the Fog was brought into existence, as a response to the expanding demands of IoT devices. Fog computing is still a relatively novel and unexplored paradigm. For it to be successful, nodes need to want to join and participate in the network operations. This is where we saw a need (and an opportunity) for an effective incentive mechanism.

This thesis was aimed at developing a pertinent and engaging incentive mechanism, with the goal of motivating network nodes to join a Fog network.

We began this thesis by defining the relevant concepts, explaining IoT use cases, and distinguishing between Cloud and Fog computing, employing an examination of their capabilities and purposes. Through an example of a smart transportation system, we illustrated the necessity for real-time data processing and the demand for supplementary research on Fog computing, including how to attract more nodes to join the Fog.

Taking the novelty of the Fog into consideration, the analysis of the related work was dedicated to exploring and analyzing existing approaches from other domains. Studies from crowdsourcing, crowdsensing and P2P network were considered and assessed, comparing the underlying systems and their requirements with those of the Fog.

The central part of the thesis proposed a new incentive mechanism to stimulate network nodes to join a Fog network and become a contributing part of it. This was accomplished in two phases: *(i)* documenting the idea, i.e., the design of the system, and *(ii)* revealing the associated implementation.

The design process commenced with an outline of the requirements of the Fog, in terms of an incentive mechanism, and comprised the system architecture, heterogeneity, proximity awareness, scalability, and volatility as the main features. A comprehensive description of a non-negative credit-based reward mechanism with a two-layer architecture followed. The lower layer represents a single area with numerous workers submitting and executing jobs, and one manager in charge of scheduling and assigning those jobs. The upper layer constructs an overlay through which the managers of the neighboring areas communicate with each other.

Global and local scheduling algorithms were introduced, moderating the credit allocation. For each task of the job submitted to the network, the manager applies the local scheduling algorithm that finds the node with the lowest cost for task execution and assigns the task to it. After the task has successfully been processed, the worker node receives its payment in credits, which the submitter pays. Upon satisfactory execution of all tasks of a job, the manager is rewarded a profit for its effort (also paid by the submitter). In the case, a local area can not handle the job load, it is forwarded to one of the neighboring managers via an overlay, which does the same in its local area. The credits earned are utilized during the job submission process as job prioritization criteria.

The introduced mechanism was then practically tested with the help of a simulator built in the course of this thesis. The focus of these experiments was put on identifying the key elements influencing the node participation, or the lack of it. The results revealed that the percentage of the nodes contributing to the system is determined greatly by the workload of the area. In other terms, if an area has more workers than the jobs submitted, only the part of the network with lower computational resources will be employed for their execution. On the other hand, if a larger amount of jobs is submitted to an area than there are workers in it, we can be sure that almost all workers will be included in their processing. The second test scenario shows that in an environment where seven workers are in charge of executing 53 tasks, node participation will be at 100%. Possibilities in how to ensure a fairly high workload are also considered.

7.2 Future Work

During the thesis, a couple of issues arose that were not handled completely (i.e., are considered out of scope) and would be a good topic for the futur. This section tackles such matters.

Churn rate: Plays a big role in networks such as the Fog. Considering nodes can unpredictably connect and disconnect from the network at any given time, making sure the system stays stable is an ongoing challenge for network designers. In our system, workers can come and go as they please (as explained in Section 4.2.2). Nodes that can not leave the network without causing substantial problems are the manager nodes. Managers, as the main point of contact, store information of the whole area, schedule the jobs, and communicate with other areas. Disconnecting, without a proper fault tolerance mechanism in place could potentially be disastrous. We leave this issue to future work, but offer some ideas on how could it be properly handled.

Once a node is promoted to become a manager, another node (a worker) from its area could be determined as a backup. All the information the manager node possess would be replicated and stored in the replication node. In such a case, once the manager disconnects, the chosen node would immediately be promoted to become the manager and continue the work of its predecessor. The rest of the system would receive an update that the address of the node has changed, but they would not experience any disruptions in their processes.

A general problem with such an approach, of having a centralized manager, is that there exists a single point of failure. An alternative would be to use the blockchain in the place of the manager. Instead of one node storing all necessary information centralized and confiding in a single node, the data the manager possesses could be distributed and available to anyone. Thus, not only increasing the fault tolerance, but the integrity of the data as well.

Payment boundaries: As discussed in Section 5, the current implementation does not offer any information to the submitter as to how much the job processing will cost in total. In the existing design, the submitter has to pay the workers and the managers their profits, regardless of how much it costs. An appealing feature would be an option for the submitter to set the upper boundary of how much it is willing to pay for the job execution. If that limit is overrun, the job would not be executed.

Partial job execution: In the case that some parts (i.e., tasks) of the submitted job are not successfully terminated, the submitter still has to pay for the others that were. Provided partial job execution is not wanted, or does not cause the aspired effect, it would mean that the submitter has not received the already-paid-for service. Again, some sort of fault tolerance mechanism would be desirable.

Scalability of the implementation: The current implementation has certain scalability limits, as discussed in Section 6. In larger scenarios with many areas, a great amount of running threads, all interacting with the database and constantly locking the resources, results in a very slow job scheduling process. In such scenarios, where many area are expected to be running at the same time, it would be crucial to find an alternative solution.

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

CC	Computing Capacity. 35
CCN	Computing Capacity with Neighbors. 35
CPU	Central Processing Unit. 28
CS	Crowdsourcing. 21
DSO	Data Service Operator. 24
ETSI	European Telecommunications Standards Institute. 8
FIFO	First In, First Out. 42
GPS	Global Positioning System. 9
Identity-changing	ID-changing. 34
IoT	Internet of Things. 1
IT	Information Technology. 10
MC	Mass Center. 35
NIST	National Institute of Standard and Technologies. 2
QoS	Quality of Service. 34
R	Reputation. 35
RA	Reference Architecture. 16
RFID	Radio Frequency Identification. 7
SoA	Service-Oriented Architecture. 9

UE	UserEquipment. 24
V2I	Vehicle-to-Infrastructure. 9
V2N	Vehicle-to-Network. 9
V2P	Vehicle-to-Pedestrian. 9
V2V	Vehicle-to-Vehicle. 9
V2X	Vehicle-to-Everything. 8
VFC	Vehicular Fog Computing. 23
VM	Virtual Machine. 13
WSN	Wireless Sensor Network. 8
ZB	Zettabytes. 12

Bibliography

[1] vmware. `https://www.vmware.com/products/esxi-and-esx.html`. Last accessed 5. July 2021.

[2] Fog computing and the internet of things: Extend the cloud to where the things are. Technical report, Cisco, 2015.

[3] White paper: Cisco annual internet report (2018–2023). Technical report, Cisco, 2018.

[4] G. T. 22.185. Service requirements for v2x services. (RTS/TSGS-0122185vf00), 2018.

[5] M. Aazam, P. P. Hung, and E.-N. Huh. Smart gateway based communication for cloud of things. In *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pages 1–6, 2014.

[6] R. Aitken, V. Chandra, J. Myers, B. Sandhu, L. Shifren, and G. Yeric. Device and technology implications of the internet of things. In *2014 Symposium on VLSI Technology (VLSI-Technology): Digest of Technical Papers*, pages 1–4, 2014.

[7] N. Alhakbani, M. Hassan, M. Hossain, and M. Alnuem. A framework of adaptive interaction support in cloud-based internet of things (iot) environment. 09 2014.

[8] K. Ashton. That 'intenet of things' thing. *RFiD Journal*, (22):97–114, 2009.

[9] H. Atlam, R. Walters, and G. Wills. Fog computing and the internet of things: A review. *Big Data and Cognitive Computing*, 2, 04 2018.

[10] M. Attaran. The internet of things: Limitless opportunities for business and society. *Journal of Strategic Innovation and Sustainability*, 12, 07 2017.

[11] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 10 2010.

[12] A. Azcorra, L. Cominardi, C. Bernardos, A. de la Oliva, and H. G. Abreha. Monitoring in fog computing: state-of-the-art and research challenges. *International Journal of Ad Hoc and Ubiquitous Computing*, 36:114, 01 2021.

[13] S. M. Babu, A. J. Lakshmi, and B. T. Rao. A study on cloud based internet of things: Cloudiot. In *2015 Global Conference on Communication Technologies (GCCT)*, pages 60–65, 2015.

[14] P. R. B.B., P. Saluja, N. Sharma, A. Mittal, and S. Sharma. Cloud computing for internet of things & sensing based applications. pages 374–380, 12 2012.

[15] D. Bermbach, F. Pallas, D. Pérez, P. Plebani, M. Anderson, R. Kat, and S. Tai. A research perspective on fog computing. 11 2017.

[16] F. Bonomi and R. Milito. Fog computing and its role in the internet of things. *Proceedings of the MCC workshop on Mobile Cloud Computing*, 08 2012.

[17] A. Botta, W. Donato, V. Persico, and A. Pescapè. Integration of cloud computing and internet of things: A survey. *Future Generation Computer Systems*, 56, 10 2015.

[18] J. Brown, J. Finney, C. Efstratiou, B. Green, N. Davies, M. Lowton, and G. Kortuem. Network interrupts: supporting delay sensitive applications in low power wireless control networks. 09 2007.

[19] D. Burgos, H. Hummel, C. Tattersall, F. Brouns, and R. Koper. *Design Guidelines for Collaboration and Participation with Examples from the LN4LD (Learning Network for Learning Design)*. January 2008.

[20] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. June 2002.

[21] M. Chiang and T. Zhang. Fog and iot: An overview of research opportunities. *IEEE Internet of Things Journal*, 3(6):854–864, 2016.

[22] O. Consortium. Introduction and overview at w3c open day. 05 2017.

[23] P. Crepaldi and T. Pimenta. *Introductory Chapter: RFID: A Successful History*. 11 2017.

[24] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. SIGCOMM '04, page 15–26, New York, NY, USA, 2004. Association for Computing Machinery.

[25] L. Duan, T. Kubo, K. Sugiyama, J. Huang, T. Hasegawa, and J. Walrand. Incentive mechanisms for smartphone collaboration in data acquisition and distributed computing. *Proceedings - IEEE INFOCOM*, 03 2012.

[26] M. Díaz, C. Martín, and B. Rubio. State-of-the-art, challenges, and open issues in the integration of internet of things and cloud computing. *Journal of Network and Computer Applications*, 01 2016.

[27] C. Feuersänger. *Manual for Package pgfplots*. LaTeX.

[28] G. Fox, S. Kamburugamuve, and R. Hartman. Architecture and measured characteristics of a cloud based internet of things api. pages 6–12, 05 2012.

[29] D. Fudenberg and J. Tirole. *Game Theory.* MIT Press, Cambridge, MA, 1991. Translated into Chinesse by Renin University Press, Bejing: China.

[30] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere. Edge-centric computing: Vision and challenges. *SIGCOMM Comput. Commun. Rev.*, 45(5):37–42, Sept. 2015.

[31] M. Gomes, R. Righi, and C. André da Costa. Future directions for providing better iot infrastructure. *UbiComp 2014 - Adjunct Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 51–54, 09 2014.

[32] T. Goyal, A. Singh, and A. Agrawal. Cloudsim: simulator for cloud computing infrastructure and modeling. *Procedia Engineering*, 38:3566–3572, 2012. INTERNATIONAL CONFERENCE ON MODELLING OPTIMIZATION AND COMPUTING.

[33] H. Gupta, A. V. Dastjerdi, S. K. Ghosh, and R. Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in internet of things, edge and fog computing environments. *CoRR*, abs/1606.02007, 2016.

[34] N. Hassan, S. Gilani, E. Ahmed, I. Yaqoob, and M. Imran. The role of edge computing in internet of things. *IEEE Communications Magazine*, PP, 05 2018.

[35] R. Hassan, F. Qamar, M. Hasan, A. Hafizah, A. Aman, and A. S. A. Mohamed Sid Ahmed. Internet of things and its applications: A comprehensive survey. *Symmetry*, 10 2020.

[36] T. Hossfeld, S. Wunderer, A. Beyer, A. Hall, A. Seufert, C. Gassner, F. Guillemin, F. Wamser, K. Wascinski, M. Hirth, M. Seufert, P. Casas, P. Tran-Gia, W. Robitza, W. Wascinski, and Z. Houidi. White paper on crowdsourced network and qoe measurements – definitions, use cases and challenges. workingpaper, Institut für Informatik, 05 2020.

[37] M. M. Islam, S. Morshed, and P. Goswami. Cloud computing: A survey on its limitations and potential solutions. 07 2013.

[38] D. Jain, P. Krishna, and V. Saritha. A study on internet of things based applications. 06 2012.

[39] V. Karagiannis, S. Schulte, N. Desai, and S. Punnekkat. Addressing the node discovery problem in fog computing. 04 2020.

[40] A. Katmada, A. Satsiou, and I. Kompatsiaris. Incentive mechanisms for crowdsourcing platforms. In F. Bagnoli, A. Satsiou, I. Stavrakakis, P. Nesi, G. Pacini, Y. Welp, T. Tiropanis, and D. DiFranzo, editors, *Internet Science*, pages 3–18, Cham, 2016. Springer International Publishing.

[41] S. Kaune, K. Pussep, G. Tyson, A. Mauthe, and R. Steinmetz. Cooperation in p2p systems through sociological incentive patterns. 12 2008.

[42] M. Kawser, S. Sajjad, S. Fahad, S. Ahmed, and H. Rafi. The perspective of vehicle-to-everything (v2x) communication towards 5g. 19:146–155, 04 2019.

[43] I. Lera, C. Guerrero, and C. Juiz. Yafs: A simulator for iot scenarios in fog computing. *IEEE Access*, 7:91745–91758, 2019.

[44] Q. Li, F. Zeng, Q. Wu, and J. Yang. An incentive mechanism based on bertrand game for opportunistic edge computing. *IEEE Access*, 8:229173–229183, 2020.

[45] S. Li, L. Xu, and S. Zhao. The internet of things: A survey. *Information Systems Frontiers*, 04 2014.

[46] M. M. Lopes, W. A. Higashino, M. A. Capretz, and L. F. Bittencourt. Myifogsim: A simulator for virtual machine migration in fog computing. In *Companion Proceedings of The10th International Conference on Utility and Cloud Computing*, UCC '17 Companion, page 47–52, New York, NY, USA, 2017. Association for Computing Machinery.

[47] R. Mahmud, S. Pallewatta, M. Goudarzi, and R. Buyya. Ifogsim2: An extended ifogsim simulator for mobility, clustering, and microservice management in edge and fog computing environments, 2021.

[48] E. Marin-Tordera, X. Masip, J. Garcia Almiñana, A. Jukan, G.-J. Ren, J. Zhu, and J. Farre. What is a fog node a tutorial on current concepts towards a common definition. 11 2016.

[49] D. Martínez, I. Vilardell, J. Rius, F. Giné, F. Solsona, and F. Guirado. Codip2p: A peer-to-peer architecture for sharing computing resources. volume 50, pages 293–303, 01 2008.

[50] D. C. Martínez, J. R. Torrento, I. B. Vilardell, F. G. d. Sola, and F. S. Tehàs. A new reliable proposal to manage dynamic resources in a computing p2p system. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 323–329, 2009.

[51] S. Mazur. Eine einführung in smart transportation: Vorteile und beispiele. `https://de.digi.com/blog/post/introduction-to-smart-transportation-benefits`, December 2020. Last accessed 18. June 2021.

[52] P. Mell and T. Grance. The nist definition of cloud computing. Technical report, National Institute of Standards and Technology (NIST), September 2011.

[53] X. Meng and B. Gallagher. The impact of incentive mechanisms on project performance. *International Journal of Project Management*, 04 2012.

[54] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, and P. A. Polakos. A comprehensive survey on fog computing: State-of-the-art and research challenges. *IEEE Communications Surveys Tutorials*, 2018.

[55] S. Naveen and M. R. Kounte. Key technologies and challenges in iot edge computing. 12 2019.

[56] O. Nazih, N. Benamar, and A. Addaim. An incentive mechanism for computing resource allocation in vehicular fog computing environment. pages 1–5, 12 2020.

[57] NEC. What is smart transportation? `https://www.nec.co.nz/market-leadership/publications-media/what-is-smart-transportation/`, Jan 2021. Last accessed 17. June 2021.

[58] OpenFog. Openfog reference architecture for fog computing. February 201.

[59] H. Panetto and J. Cecil. Information systems for enterprise integration, interoperability and networking: Theory and applications. *Enterprise Information Systems*, 7:1–6, 02 2013.

[60] M. Parameswaran, A. Susarla, and A. Whinston. P2p networking: An information-sharing alternative. *Computer*, 34:31 – 38, 08 2001.

[61] P. Parwekar. From internet of things towards cloud of things. pages 329–333, 09 2011.

[62] T. Qayyum, A. Malik, M. Khan, O. Khalid, and S. Khan. Fognetsim++: A toolkit for modeling and simulation of distributed fog environment. *IEEE Access*, PP:1–1, 10 2018.

[63] J. Rius, F. Cores, and F. Solsona. A new credit-based incentive mechanism for p2p scheduling with user modeling. In *2009 First International Conference on Advances in P2P Systems*, pages 85–91, 2009.

[64] J. Rius, S. Estrada, F. Cores Prado, and F. Solsona. Incentive mechanism for scheduling jobs in a peer-to-peer computing system. *Simulation Modelling Practice and Theory*, 06 2012.

[65] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In R. Guerraoui, editor, *Middleware 2001*, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[66] O. Scekic. Incentive mechanisms for social computing. In *2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pages 162–167, 2015.

[67] O. Scekic, H.-L. Truong, and S. Dustdar. Incentives and rewarding in social computing. *Communications of the ACM*, 06 2013.

[68] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[69] W. Shi and S. Dustdar. The promise of edge computing. *Computer*, 49:78–81, 05 2016.

[70] C. Sonmez, A. Ozgovde, and C. Ersoy. Edgecloudsim: An environment for performance evaluation of edge computing systems. In *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 39–44, 2017.

[71] J. Spillner, J. Müller, and A. Schill. Creating optimal cloud storage systems. *Future Generation Computer Systems*, 29:1062–1072, 2013.

[72] Statista. Internet of things (iot) revenue worldwide from 2019 to 2030 (in billion u.s. dollars), by vertical. `https://www.statista.com/statistics/1183471/iot-revenue-worldwide-by-vertical/`, December 2020. Last accessed 9. May 2021.

[73] Statista. Number of internet of things (iot) connected devices worldwide from 2019 to 2030. `https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/`, December 2020. Last accessed 9. May 2021.

[74] I. Stojmenovic. Fog computing: A cloud to the ground support for smart things and machine-to-machine networks. *2014 Australasian Telecommunication Networks and Applications Conference, ATNAC 2014*, 01 2015.

[75] M. Syed, E. Fernández, and M. Ilyas. A pattern for fog computing. pages 1–10, 04 2016.

[76] K. Tati. *Exploiting heterogeneity in peer-to-peer systems*. PhD dissertation, UC San Diego, 2006.

[77] N. Tomar and R. Matam. Optimal query-processing-node discovery in iot-fog computing environment. In *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 237–241, 2018.

[78] A. Varga and R. Hornig. An overview of the omnet++ simulation environment. Simutools ’08, Brussels, BEL, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[79] W. S. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *Journal of Finance*, 16:8–37, 1961.

[80] A. Whitmore, A. Agarwal, and L. Xu. The internet of things—a survey of topics and trends. *Information Systems Frontiers*, 17, 04 2014.

[81] Z. Xiaoqing, D. S. Chan, H. Hu, M. S. Prabhu, E. Ganesan, and F. Bonomi. Improving video performance with edge servers in the fog computing architecture. In *Intel Technol. J.*, volume 19, pages 202–224, April 2015.

[82] D. Yaga, P. Mell, N. Roby, and K. Scarfone. Blockchain technology overview. 2018-10-03 2018.

[83] D. Yang, G. Xue, X. Fang, and J. Tang. Incentive mechanisms for crowdsensing: Crowdsourcing with smartphones. *IEEE/ACM Transactions on Networking*, 2016.

[84] S. Yangui, P. Ravindran, O. Bibani, R. H. Glitho, N. Ben Hadj-Alouane, M. J. Morrow, and P. A. Polakos. A platform as-a-service for hybrid cloud/fog environments. In *2016 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–7, 2016.

[85] S. Yi, Z. Hao, Z. Qin, and Q. Li. Fog computing: Platform and applications. 11 2015.

[86] S. Yi, C. Li, and Q. Li. A survey of fog computing: Concepts, applications and issues. In *Proceedings of the 2015 Workshop on Mobile Big Data*, Mobidata '15, page 37–42, New York, NY, USA, 2015. Association for Computing Machinery.

[87] A. Zadgaonkar and R. Dharaskar. Digital forensic investigation challenges in p2p networks. 02 2011.

[88] A. Zaslavsky, C. Perera, and D. Georgakopoulos. Sensing as a service and big data. 07 2012.

[89] M. Zeng, Y. Li, K. Zhang, M. Waqas, and D. Jin. Incentive mechanism design for computation offloading in heterogeneous fog computing: A contract-based approach. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6, 2018.

[90] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: State-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1:7–18, 05 2010.

[91] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and. Technical report, USA, 2001.