An Internet of Things network for proximity based distributed processing

Iakovos Koutsoumpakis
Abstract

An Internet of Things network for proximity based distributed processing

Iakovos Koutsoumpakis

The Internet of Things, the interconnection of all computing devices, is a concept that has become very popular nowadays and many companies try to achieve a leading role in shaping its future. Billions of devices are already connected to IoT cloud networks and this number is expected to rapidly increase in the near future. Devices in an IoT cloud network can be producers or consumers of data, while some can be processors. As data often needs processing in order to be transformed from lower to higher conceptual value, before being delivered to the consumers, this processing has to be done in an efficient manner. Ideally processing should take place in the proximity of data producers as opposed to having to transfer large volumes of data over the network in order to reach the processor. For this problem to be solved, scheduling algorithms require additional information that quantifies the "distance" between the different nodes in an IoT cloud network. Consequently, the main focus of this work is the development and the evaluation of an efficient mechanism that uses a heuristic technique to estimate this information, the latency between nodes, greatly reducing to linear the running time complexity that, if every device had to contact every other to calculate it, would be $O(n^2)$. 
## Contents

1 Introduction ........................................................................................................ 7

2 Background ........................................................................................................ 9
   2.1 The Internet of Things .................................................................................. 9
       2.1.1 Definition and explanation .................................................................. 9
       2.1.2 History of the internet of things ......................................................... 11
       2.1.3 The present ......................................................................................... 11
       2.1.4 The future ........................................................................................... 12
       2.1.5 Applications of the IoT ....................................................................... 12
   2.2 Distributed computing .................................................................................. 13
       2.2.1 Definition and characteristics ............................................................. 13
       2.2.2 History of distributed computing ....................................................... 14
       2.2.3 Fogging or fog computing .................................................................... 14
   2.3 Internet of Things data as input to the cloud ............................................ 15
   2.4 The processing in the Internet of Things ................................................... 17
   2.5 Latency estimation ...................................................................................... 17
   2.6 Modelling a distributed processing system .............................................. 17

3 Methodology ....................................................................................................... 20
   3.1 Binning Methodology for Latency Estimation .......................................... 20
   3.2 The actor model .......................................................................................... 21
   3.3 Scala and Akka ........................................................................................... 22
   3.4 Master-Slave Model .................................................................................. 23
   3.5 Difficulties encountered ............................................................................. 24
   3.6 Partial solutions ......................................................................................... 24

4 Implementation .................................................................................................. 26
   4.1 Landmark technique ................................................................................... 26
   4.2 DNS Actors ............................................................................................... 27
   4.3 Latency/Pinging .......................................................................................... 27
   4.4 The final algorithm of our distributed system ........................................ 28
   4.5 Master and Slave nodes ............................................................................. 32
   4.6 Scalability analysis .................................................................................... 32
   4.7 Time complexity analysis ......................................................................... 33

5 Results ............................................................................................................... 34
   5.1 Results ....................................................................................................... 34

6 Conclusion ......................................................................................................... 36
1. Introduction

More and more devices nowadays have the ability to connect to the Internet and share vast amounts of data. Computers, smartphones, gadgets, sensors, cameras, fridges and cars are only some of them. At the same time, more than 30 billion devices are expected to be wirelessly connected to the Internet of Things by 2020. To this direction, the dream of having every device connected to the Internet has coined the term Internet of Things (IoT) [1] back in 1999 [2], while the whole concept of having uniquely identifiable objects connected online dates even further back in time. The applications and benefits of the IoT are innumerable: smart cities, smart cars, home automation, intelligent shopping, industrial control, eHealth and more.

Of all these devices however, some are only producers of data, like sensors, which data often need processing, while others provide the necessary processing capabilities. When considering the potential vastness of this data, it becomes of importance that we process it in an efficient way, bringing the computational tasks as close to the data as possible. The difficulties we face are a lot, stemming from the variety of the devices and the data, their huge amounts, the abstract topology of the cloud, the different processing capabilities of the devices, the computing complexity of the computations, the bandwidth and latency over the network and more.

An example demonstrating the above claim is that of a smart and connected home: the owner might want to be automatically informed about increased traffic before he leaves for work and his alarm clock and coffee machine to adjust to the new departure time. The heater at the house should be disabled automatically as well, when he leaves. In this example it is obvious that a processing device will be required to handle multiple computations, gathering an abundance of data from different inputs. When the problem scales, with more devices and houses, the communication effort increases and billions of such computations are required, the need for handling everything in an efficient, scalable and flexible manner becomes obvious.

Also, considering the variety of all the potential devices, it is obvious that data to be processed will be of different kinds and amounts and thus special treatment is required. Normally in such cases load balancing techniques are required. In our case, however, we will try to provide a mechanism that will decide which processor should do the computation, taking into consideration the proximity of the processor to the input, as well as the proximity to the output nodes.
This report attempts to model such a distributed system with data producing devices and processors that can process the data efficiently and can further simulate the system with an abstract and flexible network topology:

i) nodes (data producers or processors) should be able to locate the network of connected devices and register,

ii) some leader node should handle and monitor the process of adding/deleting other nodes,

iii) nodes should be able to locate and contact each other in a fast and simple manner, thus without a big impact on the overall performance of the system. This means that a node must not be required to maintain massive lists of other nodes or send parallel requests to more nodes in order to locate the desired one,

v) some node should perform task distribution to processors and navigate data from input node to the processors,

vi) processor nodes (called slave nodes) perform computational tasks that are assigned to them by the node that performs the task distribution, thus in a distributed manner,

vii) the system must be scalable.

At the same time, communication latency between data producer and processor is of major importance and is taken into consideration when assigning computational tasks (and thus sending the data) to processors. So it is critical that we introduce methods to keep the average latency between the data input and the processors that will process it as low as possible. However, the smart distribution of the tasks using a scheduling algorithm is part of the thesis but is tackled by Andreas Moregard Haubenwaller in his master thesis[3]. So, the innovation of this work is the introduction of proximity of nodes as the main parameter when performing task distribution. Normally, though, as a pre-work, we would require to get all latency values between all pairs of nodes in the system. This report further describes such a system that, with a use of some heuristic technique can quickly estimate these latency values, greatly reducing the time complexity, which would otherwise be O(n^2) as every node would have to contact every other to measure the latency, to O(n), where n the number of nodes.

This document is structured as follows: In chapter 2 the background of the work is described in detail. Then, the methodology is explained in chapter 3, including the model used and the purposes, as well as the techniques for the latency estimations. In chapter 4 we describe the implementation and the programming languages and frameworks used. In chapter 5 we present the testbed, the equipment and the results and in chapter 6 we conclude with some useful results and suggestions.
2. Background

This chapter provides the background one might need in order to understand this thesis work. At first an introduction to the Internet of Things is made, while afterwards there is an explanation of the main concepts, concerns and problems the thesis depends on or attempts to solve, such as distributed computing and latency estimation.

2.1 The Internet of Things

This section provides a definition and study of the Internet of Things, presenting notable milestones from its very first appearance as a concept in the world of technology to current trends and applications, as well as the dreams for the future.

2.1.1 Definition and explanation

According to wikipedia [2], the Internet of Things (IoT, also Cloud of Things or CoT) refers to the interconnection of uniquely identifiable embedded computing-like devices within the existing Internet infrastructure. On a more straightforward definition, the Internet of Things (IoT) is a computing concept that describes a future where everyday physical objects will be connected to the Internet and be able to identify themselves to other devices. The term is closely identified with RFID as the method of communication, although it also may include other sensor technologies, wireless technologies or QR codes. IoT is defined to connect humans, devices and systems.

The dream behind the IoT is building smart houses that open their doors for us, brew coffee for us when we wake up, turn down the lights when we start watching a movie, open the garage door when we arrive at home, etc. Even in old or new movies and series people interact with computers: people communicate with the computer on Star Trek and even more, in Eureka there is SARAH, a smart house of the future which provides numerous services for her residents and guests, from opening and closing the sealed door, to controlling internal temperature, and even providing a variety of drinks. A more ambitious concept is telemedicine and healthcare, which means building systems that can monitor humans with sensors implanted on their bodies that will trigger some alerts if anything is wrong and hospitalization is required.
Figure 2.1. The Internet of Things - photo credits: blog.surveyanalytics.com
2.1.2 History of the internet of things

The general concept of connecting smart devices in a network has been under discussion since at least 1991[4]. In 1994, a concept of "moving small packets of data to a large set of nodes, so as to integrate and automate everything from home appliances to entire factories" was described in IEEE Spectrum by Reza Raji[5].

However we can trace attempts to the direction of Internet of Things even further before that, and more specifically back in 1982, when the Carnegie Mellon University developed the first Internet appliance, the Coke machine, a device connected to the Internet, which programmers could access online and check its status, as well as make it prepare the drinks[6].

Yet 1999 is the real milestone, as it is the year the term Internet of Things is coined by Kevin Ashton [1]. The original interpretation of the Internet of Things required equipping all objects with machine-readable identifiers or minuscule identifying devices so that they can be managed and inventoried by computers. This is the direction the Massachusetts Institute of Technology (MIT) followed by starting, in 1999, to design and propagate a cross-company RFID infrastructure[7]. RFID (Radio-frequency identification), was seen as a prerequisite for building the Internet of Things network.

Soon after, In 2000 LG announced the first Internet refrigerator plan, releasing in 2001 the first of its kind. Internet refrigerators are Web-enabled computer refrigerators that use RFID or barcode technology to handle food stock and expiration monitoring while they can often propose ordering of products.

In the mean time, some other technologies similar to RFID that could also enable tagging of things for near field communication have been implemented, such as QR codes, barcodes and digital watermarking. [8]

Finally, in 2008, the number of things connected to the Internet exceeded the number of people on Earth[9].

2.1.3 The present

The Internet of Things has become very popular nowadays and the numbers are increasing: According to Cisco, there were 8.7 billion in 2012 connected devices in 2012 and more than 10 billions in 2013. This number is expected to increase to 15 billion by 2015. To further boost the trend of connecting devices, April 9 is celebrated as Global Internet of Things Day. Additionally, the Internet of Things World Hackathon, a 30-hour app building contest is held in Palo Alto, Silicon Valley annually, giving developers reasons to produce useful frameworks that will strengthen the networked society.
Despite the increasing rate of connected devices on the network, however, it is estimated that more than 99% of physical objects that may one day join the network are presently still unconnected. Also, it is estimated that there are only about 3 devices connected to the Internet per person in the world (although the real Internet users amount to 40% of the global population).

At the same time, the transition to IPv6 protocol [10] is taking place at a very slow pace. As the IPv4 address space is exhausted, it is important that we switch to IPv6, which offers an increased address space and can accommodate for a massive amount of devices, roughly 100 for every atom on the surface of the planet [11].

2.1.4 The future

Cisco has been calling the future of Internet of Things as the Internet of Everything [12]. According to Cisco, "the Internet of Everything (IoE) brings together people, process, data, and things to make networked connections more relevant and valuable than ever before, turning information into actions that create new capabilities, richer experiences, and unprecedented economic opportunity for businesses, individuals, and countries".

As things add capabilities like context awareness, increased processing power, and energy independence, and as more people and new types of information are connected, IoT becomes an Internet of Everything - a network of networks where billions or even trillions of connections create unprecedented opportunities as well as new risks. Ericsson dreams about 30 billion devices being connected by 2020, while Cisco estimates it them to be 50 billion. [13]

2.1.5 Applications of the IoT

A list of possible applications of the Internet of Things is presented below [2][9]:

- Environmental Monitoring
- Infrastructure Management
- Industrial Applications
- Energy Management
- Medical and Healthcare Systems
- Building and Home Automation
- Transport Systems
- Large Scale Deployments
- Personal and home usage
- Sensor Networks
- Smart Cities
Smart Animal Farming

For example, by implanting RFID sensors in humans we can gather data and send it to physicians, hospitals or other individuals to monitor it, increasing awareness and efficiency in preventing serious incidents, while achieving, at the same time, a reduction of the hospitalizations[14]. Alternatively, by having sensors constantly monitoring equipment such as washing machines or air conditioners we can improve their efficiency. Also, by introducing systems that monitor sensors in critical systems we can prevent hazards and disasters. When it comes to agriculture, we can have sensors and devices automating the crop monitoring.

Smart parking, smart cities, smart roads are also just few more of the several applications the Internet of Things can have, with innumerable benefits for the users.

2.2 Distributed computing

In this section we attempt to provide a background on distributed computing, which is a fundamental part of system we implement.

2.2.1 Definition and characteristics

As defined by Wikipedia, "Distributed computing refers to the use of distributed systems to solve computational problems. In Distributed computing a problem is divided into many tasks, each of which is solved by one or more computers, which communicate with each other by message passing". [15]

A distributed computer system consists of multiple several autonomous computational entities, usually called nodes. These are software components running on multiple computers, they all have their own memory and they all collaborate and run as a single system. The nodes in a distributed system can be of any kind of possible configurations, such as mainframes or supercomputers, personal computers, workstations, minicomputers or even mobile devices with processing capabilities and so on. All these machines may be physically close together, connected in a Local Area Network, or they can be in different geographic locations, connected by a Wide Area Network. They communicate by passing messages and coordinate their actions in order to achieve the common goal. If seen from the outside, they act as a single computer.

The structure of the system(meaning the number of computers, topology of the network and latency) is not known in advance and it might change during
the execution process. Additionally, every node only has a limited view of the system and will probably get only one part of the input.

Fault tolerance assumes that if an individual node fails, then only that specific one will be readdressed and not others.

2.2.2 History of distributed computing

Distributed computing is a fairly old concept, as concurrent processes communicating by message passing were present in operating system architectures introduced and studied in the 1960s[16]. In the 1970s the first widespread distributed systems appeared, mostly having Ethernet connectivity as a standard, thus being connected in local area networks. Moreover, ARPANET, the predecessor of the Internet, was developed in late 1960s, while the e-mail was developed on it only a few years later. E-mail soon got widespread use, becoming the earliest example of a successful large-scale distributed application.

Other than the ARPANET and the Internet, Usenet[17] and FidoNet[18] were introduced in 1980s, both supporting distributed discussion systems(emails, forum discussions etc).

As of the late 1970s distributed computing is regarded as a branch of computer science on its own, while the first conferences appeared at the same time, namely the Symposium on Principles of Distributed Computing (PODC) in 1982[19] and International Symposium on Distributed Computing (DISC) in 1985[20].

In the last two decades, several distributed computing projects like Folding@ home[21] use the processing resources of thousands of personal computers or gaming consoles for scientific research[22].

2.2.3 Fogging or fog computing

Fog Computing is considered as an extension to Cloud computing and its services. It is a paradigm that, just as Cloud computing, provides data, computations, storage and applications to the users[23]. However, fog computing takes these services closer to the users, hosting them at the network edge or even further at set-top-boxes and access points at the outer points of the network. This requires that a fog network supports for mobility. The final network has a dense geographical distribution offering low latencies and thus better Quality of Service[24].
These features and characteristics bring the Internet of Everything closer, as they can support the demands of applications for real-time latency (e.g. in actuators, industrial automation or transportation).

2.3 Internet of Things data as input to the cloud

As mentioned in chapter 1, Internet of Things data can be simple sensor data, for example smoke detector values or speedometers, thermometers, seismometers and so on. Or it can be more complicated data like webcam photos or car error logs, traffic information etc. Once all these devices are connected with each other, they enable more and more smart processes and services that support our basic needs, economies, environment and health. Such enormous number of devices connected to internet provides many kinds of services and produces, though, huge amounts of data and information, which needs processing.

Examples of computational tasks on these Internet of Things data can be monitoring for trigger events like hazard indicators, sensor warnings, sensor malfunctions, alarms going off, Another kind of computations can be the creation of virtual streams, i.e. aggregations (average/min/max etc.) of one or more existing streams of sensor data, e.g. average temperature CO\textsubscript{2} emissions in some parts of a city. Moreover, other tasks can be image processing and pattern recognition on images or sound clips or traffic camera streams, feature extraction on logs, etc.

A concrete example of an Internet of Things system is a smart house. In a smart house there can be many types of sensors that produce data: Several motion sensors report possible movement, surveillance cameras send video feed, thermometers report the temperature, fire alarms monitor the house etc. Computational tasks would be turning on the lights when movement is detected by a motion sensor or camera in the room, triggering or alerting the police in case of unexpected activity, unlocking the doors and informing the fire department in case of fire, turning on the heating/cooling when the temperature is outside the optimal levels, automatically switching the TV to show feed from surveillance camera when motion is detected at the entrance, switching on the lights and heating when a car enters the garage. Also, the smart house can monitor the energy consumption at any time and report losses, automatically monitor and feed the pets according to a schedule, create mood lighting for any occasion etc.

Combining all this information from the different sensors and taking such decisions requires lot of processing to be done. Additionally, some central processor or server is also desired, in order to make the house accessible to the authorized user from the outside e.g via a smartphone.
Figure 2.2. Smart home - photo credits: http://smarthomeenergy.co.uk
2.4 The processing in the Internet of Things

When having IoT data as input, it is obvious that there will be numerous sources of data instead of one. To this direction, when processing this data, we have the option to either move the data to the nodes that will do the processing, or move the actual computations closer to the data input within the cloud. Moving the data might be a big burden for the network bandwidth and the overhead can dominate the total time required. So it is more logical to move the computations to nodes as close to the input as possible. However, since we also need to consider the processing capabilities of the nodes, we have to relate between network overhead and nodes’ capabilities carefully in order to plan the task distribution in a smart way and get the optimal result.

2.5 Latency estimation

As we have described, the latency and bandwidth required within the cloud network are two important factors we have to consider when performing the task distribution, as they impose serious limitations on the performance of the network. However, it is not always easy to estimate any of them. In a large scale system we have a massive amount of nodes and we can not maintain latency times or bandwidth throughput information about any combination of them. Also, it is impossible or inefficient to maintain large amounts of such information in every node or centrally. Yet we need to know the latency for any possible combination of the present nodes when performing the task distribution. Furthermore, even when we add new nodes in the system, we can not assume that we will start inspecting the connection with the majority of the existing nodes, as this would require a lot of time. This necessitates techniques that will give us useful information in low cost and impact. And while there is plenty of available research material on this field, the problem is still open. So, the best to be found is some heuristic techniques, that can efficiently estimate latency values between nodes. Searching for such techniques, we realized that the bibliography on this topic is poor, as we only found one method. This technique is used here and will be described later[25].

2.6 Modelling a distributed processing system

So, the model we attempt to develop will consist of nodes that can reach each other via the network. The network can grow at any time, meaning that new nodes can discover it and register at any moment. Some leader node(s) will perform task scheduling and assign computational tasks to any of the other
nodes. The main parameter for the scheduling should be the latency between the node that will handle the computational task on some (Internet of Things) data and the node that inserts the data. Thus, the system must utilize some efficient way to locate some processor nodes with small latency towards the one that pushes the data, without having to measure numerous inter-node latencies. Additionally, since the network can grow to huge numbers of nodes, some balancing techniques should be implemented in order to avoid bottlenecks, mostly from the schedulers. Finally, for simplicity in the implementation we can assume that all the nodes in our model can be either producers of Internet of Things data, or processors. The nodes can communicate with message passing.

Further extending and analyzing the aspects of our system description, desirable characteristics of the model are the following:

- scalability
- platform independence
- code migration
- easy partitioning of tasks
- simplicity

The first and most important factor that comes to mind when we try to define the necessary points for our model of the distributed system is the scalability. Since we are going to have a potentially massive Internet of Things network, with millions or billions of connected devices, it must be able to handle them without a problem. Each node should have a relatively small and well defined role and the communication cost needs to be kept as low as possible. Ideally, no nodes should be allowed to act as key nodes that millions of others depend on. More clearly, we want all nodes to have only small computational tasks.

Additionally, as the different devices can be running different software or operating systems, we need to emphasize on platform independence. This way we will provide the ability to support for compatibility with more types of devices and make sure task distribution will be easier.

Code migration is another desired thing, as it allows for a smart distribution of the tasks and an easy reconfiguration of the task distribution and execution.

Moreover, we want to support an easy partitioning of tasks into small sub-tasks. As we want every node of the network to be handling small computational tasks, so should the tasks themselves be clear and easy to break into smaller, independent ones. A distributed processing model that by definition uses small tasks should be preferred.

Simplicity means that the implementation of the nodes ought to be simple. As every node should be handling small computational tasks, so it is also that the implementation of the node and its introduction to the system has to be
simple, while accessing and re-addressing its tasks must be handled without a lot of effort or time required.

To sum up, the challenges we encounter include the need to support heterogeneity, to have a good network topology, simplicity in adding/removing nodes and, of course, scalability.
3. Methodology

3.1 Binning Methodology for Latency Estimation

As we mentioned earlier, we need to know the latencies between any pair of nodes in the network when we perform the task distribution. We emphasize the fact that in our system we will need to know the latencies between the node that inputs the data and all possible processor nodes beforehand, as it is not acceptable to calculate them just when we want to do the task distribution.

A naive way of calculating these latencies would be by asking every single node in the network to send a ping request to every other node. This of course would be extremely slow and terrible for the scalability of the system, having a $O(n^2)$ complexity, for an n number of nodes. So, we should better make use of a technique to efficiently estimate latencies than measure them. To this direction we can consider that, heuristically, if two nodes are both very close to a third one, then they are relatively close to each other[26](likely to have small latency between them). So, with the use of this third node that we call a landmark node, we can safely assume that the latency between them will be relatively small and generally smaller than with the other nodes in the network.

![Diagram of nodes and landmarks](image)

*Figure 3.1.* The slave node $s_5$ that is close to landmark node $L_3$ is more likely to be closer to $s_3$ than to any of the slaves registered on the other landmarks.
Figure 3.1 demonstrates the statement above: the slave node $s_5$ is, in the average case, closer to $s_3$ than to any slave node registered on the other two landmarks. The strong point of the method is that with only $n$ latency measurements for $n$ slave nodes we can safely come to the assumption that all the nodes that register in the same landmark have small latencies between them, without having to measure latencies between pairs of slaves.

Keeping some strict latency thresholds for the distance between each of these two and the landmark one, then we can normally assume that we will also get an acceptable latency between them. In the case that there is no existing landmark node in the vicinity, a new node could declare itself a new landmark node.

This procedure is called binning methodology or landmark technique[27] and it is state of the art in the field.

The adaptation of the binning methodology in our system is described in 4.1.

### 3.2 The actor model

The Actor Model is a mathematical model of concurrent computation first proposed by Carl Hewitt in 1973[28]. It proposes a new approach to concurrency, aiming at avoiding common problems the thread model of concurrency faces, due to the use of locks. In this model, there is no shared state and the processors communicate with each other by message passing. Individual processes (actors) are allowed to send messages asynchronously to each other[29]. Also, the execution of the program is split up into concurrently running tasks, though these must be small and simple.

In the actor model, each object is an actor, an entity that has a mailbox and a behaviour. The messages that are exchanged between actors can be buffered in their mailbox. When an actor receives a message, its behaviour will be triggered, upon which the actor can send a number of messages to other actors, create some more actors or designate some new behaviour for when the next message will be received.

What is important in this model is that all communications are performed asynchronously. So, there are no blocking calls and the sender will not wait for a message to be received upon sending one. It will immediately continue with the task it was executing. Also, an actor can receive messages from any sender. There is no guarantee, though, that the messages an actor sends will be delivered to the recipient in one particular order, though.
Another important property is that, as mentioned earlier, message passing is the exclusive method of communication, which means that there is no shared state between actors. If an actor wishes to obtain information about the internal state of another actor, it will have to use messages to request this information. Thus, access to an actor’s state is easily controlled, and this helps avoiding problems like the lost-update problem, which happens when two or more operations try to access and alternate the same value at the same time. In this problem, as the two operations are not atomic, it is possible that their execution gets interleaved and the final value is incorrect. Also, the actor model does not risk from deadlock issues that could result from the use of locks, the method thread-based concurrency utilizes to ensure mutual exclusion and bypass the lost-update problem.

3.3 Scala and Akka

Scala is a general purpose programming language designed at Ecole Polytechnique Federale de Lausanne. Its design started in 2001 and was publicly released in 2004 and is slowly getting momentum. Overall, Scala bears many resemblances to Java, especially in syntax, where it inherits a lot from Java. It follows a multi-paradigm, supporting for both functional and object-oriented programming. Scala source code is compiled to Java bytecode and is thus executable on a Java virtual machine. Also, Scala code can use Java libraries or extend Java classes. Everything is a variable, e.g. a value, an expression, a function or even a method can be given as a value in a variable and variables are treated as objects. Data types and behaviours of objects are described by classes and traits, which are the equivalents of Java’s interfaces. Scala has a strong static type system and while it does not enforce the use of immutable objects, it encourages a distinction between mutable and immutable objects.

Overall, Scala code is more concise and readable than Java. This mainly stems from the functional characteristics of the language, as in functional programming functions are less complex, only mapping an input value to an output, \( f(x) = y \), which means they never alternate any variable values. This, along with the extensive use of immutable objects reduces the risks of bugs in the code, as well as common concurrency issues like the ones described in section 3.3.

Finally, another advantage of Scala that is of major interest for this thesis, is that it offers parallelism and concurrency at a high level of abstraction, with the latter featured by the actor model. To this direction, Akka was implemented in Scala, constituting an open source framework that provides actor-based concurrency. The first public release of Akka dates back in Jan-
uary 2010, under version 0.5. Akka supports multiple programming models for concurrency, however it emphasizes actor-based concurrency. It supports for writing highly concurrent event-driven (more specifically message-driven) applications.

One of Akka’s main strengths is location transparency. Since actors are created in a strictly hierarchical fashion, there is path which is a unique sequence of actor names given by recursively following the supervision links between child and parent down towards the root of the actor system. Using these actor paths, we can reach the actor. During message passing, sender reference is always included in the messages sent as well. These actor paths can either be purely local references or can be remote references, which support networking functions. This brings us to another strong feature of Akka, the remote interaction.

Akka remoting feature is designed for communication in a peer-to-peer fashion using tcp protocol. With remoting we can lookup a remote actor or create actors with remote deployment.

Remote actor lookup is the process of getting the ActorSelection in order to be able to interact with it in the same way as with a local actor. In order to remotely deploy an actor, though, the classloader of the actor systems has to have a JAR containing the class. This can be done, for example, by having the jar file of the code there. When sending messages to remote actors, serialization of the message is required.

Finally, Akka users can utilize the cluster capabilities to join actors in clusters easily, with seed nodes that are considered initial contact points for discovering the network. Nodes in a cluster share the same actor system. In a cluster set-up, nodes can subscribe to cluster events and get notified about changes in the cluster’s structure and nodes’ availability.

3.4 Master-Slave Model

To serve our need of having schedulers to distribute tasks and some nodes to process them, the Master-Slave model of communication between nodes comes in handy. According to Wikipedia, Master/Slave is a model of communication where one device or process has unidirectional control over one or more other devices. With this model some nodes will handle the task distribution, while the rest are responsible for the actual execution of these tasks. The first ones are called master nodes and the second ones are the slaves. The number of slave nodes is normally much greater than the number of master nodes. In such systems the master is often elected from a group of eligible devices, while the other devices act as slaves. The master node can be of dif-
ferent architecture or a more powerful processor or can just be a random one of the available devices.

3.5 Difficulties encountered

While modeling the system we encountered some problems which limited the implementation and required modifications of the methodology proposed. The first difficulty was simulating a scaled-up system for extracting results. In order to get realistic results, we would require the use of hundreds or thousands of devices, in order to reach the master and DNS nodes’ actual processing capabilities and their ability to handle multiple simultaneous requests and connections, because, as explained earlier, the cost of managing the lists of registered slaves is trivial. The task distribution itself is a considerably heavier task, however even the task distribution for 50-100 devices would be easily handled by an average PC. Still this is not within the scope of this thesis but the one of Andreas Moregard Haubenwaller.

Furthermore, if we use just a few devices and spawn a very big number of nodes there, the bottleneck will be on other resources of the system and not the expected ones on a real system: Then the latency values will not be accurate, as the increased workload for the numerous master nodes running on their processors, or perhaps some blocking calls, will distort the results. This happens because operating systems and processors of devices struggle when they have to handle tenths or hundreds of nodes’ instances running on them.

In addition, all these devices would need to have a solid range of latency times with each other over the network, in order to demonstrate the strength of and necessity for latency estimation techniques that we use for achieving small latencies. This is not possible to acquire in a small Local Area Network, where the latency times are small and consistent. Thus, a bigger network with a more random and wide distribution of latency values would serve better but is not easy to get.

Finally, not having a system or implementation to compare our implementation against is another concern, when it comes to the evaluation part.

3.6 Partial solutions

In order to tackle some of the problems explained above, some modifications were required:
We will eventually use only one computer for the tests but we will not consider the real latency times we get as an indication of the performance of the system. They will only be used for the registration of the slave nodes and for the latencyThreshold. Since they cannot be genuine, we will only use fake ones and consider the number of splits that happen as an indication for the performance instead. In order then to get various latency values without having a real network of connected devices, we introduced some artificial latency increase using a uniform distribution.

We should also have in mind that asking the DNS actor to remotely deploy a master node is a costly part of the code, being slow itself and also requiring a blocking call, which means that some resources remain idle.

Finally, we conclude that splitting a master into two is the heaviest task as it requires asking the DNS actor to deploy an actor remotely with a blocking call, then all the registered slaves are contacted to ping the new master actor and additionally some of them will have to be deleted and send their information to the new master. So we aim at reducing the number of splits that happen, as the time these require is dominant.
4. Implementation

In this chapter we explain the tools used to model the system, such as the programming language and framework, we explain how we applied the landmark technique and how we tackled problems such as the discovery of the network and the latency measurement. Then we present the our algorithm and evaluate it in terms of scalability and complexity.

4.1 Landmark technique

In our implementation of the cloud network, we will make use of the previously binning methodology (landmark technique) for latency estimation which was introduced in section 3.1. As explained there, with this technique we divide the nodes to some subgroups (or clusters), which can be described as bins (buckets), as each subgroup could be a bin of nodes. Each bin only contains nodes that have small latency between them. When a new node arrives and wants to choose a bin, it will only contact a representative of each bin, which is called landmark node. If two nodes have low latency with the landmark one, then heuristically they have small latency between them as well. This technique was originally intended for estimating latencies between random nodes on the network. However, in our work, we will extend its use to support our purposes for task distribution.

The master nodes will serve as landmark nodes and the bin they form will consist of the slave nodes that belong to them. Thus, when a new slave arrives and wants to register in the network, it will ping all master nodes and measure the latency. Then, it will ask to register in the closest one, i.e. the one with the lowest latency. The master-landmark nodes maintain lists of slave nodes registered there.

Additionally, in order to guarantee that at the end no latency values will be very big, we will introduce a threshold that we will call latencyThreshold after which no bin is considered as an acceptable solution. When a slave node wants to register but the latency to the closest landmark node exceeds this limit, it will become a new master node itself.

In some other implementation, slave nodes could also serve as landmark nodes as well, however this would result in a more complicated implementation with no obvious benefits.
4.2 DNS Actors

As the new nodes that arrive will need to get information about how to locate the network, its nodes and their IP addresses, it is obvious that we need to have a starting point, i.e. (at least) one fixed address that we can contact. One solution is by introducing a reference node, hereby called a DNS node, which has the responsibility of maintaining a list of all master nodes’ addresses in the network. The DNS node is then responsible for keeping this list up-to-date and sharing it with nodes that want to register. The DNS node’s IP address will be considered as publicly known and every node that wants to register will contact this first.

Thereafter, as we are using the landmark technique with the bins of nodes, if we get the list of master nodes in the network we can have access to every single node present in the cloud, as those maintain lists of slave nodes registered in them. So, it is only a matter of prompting the DNS nodes to provide the list of the master nodes. Additionally, in our implementation with Akka, the DNS Actor is the one that deploys the master actors, using Akka’s feature for deployment of actors on a remote location. To sum up, the tasks of the DNS actor are as follows: the DNS Actor receives a registration request and registers a node as a master node, also remotely deploying a master actor there. Additionally, it accepts requests from slave nodes that ask a master actor to be deployed there, in the case that there is no master node within the acceptable latency threshold. In this case it creates a master actor there, while of course updating the masters list.

4.3 Latency/Pinging

The simplest way of measuring latencies is by measuring the time difference between the moment we ping and the moment get a response. So, we need every master node to be able to respond to ping requests. Since in the actor model we want every actor to have simple and clear tasks, it is not a good practice to assign this task to the master actor. Additionally, as a master node might often be in a blocking state, the latency result would not always be accurate. To sort this problem out, we make use of some Ping Actors, one for each master node, which are initiated the moment a master actor is initiated on a node and have as the only task to respond to ping requests for measuring the latency.
4.4 The final algorithm of our distributed system

This section presents the algorithm that implements the actors that compose our Internet of Things framework for distributed processing, supporting for discovery of the network and registration, balancing of the network and achievement of low average latencies, based on the models and techniques presented in Implementation chapter. There is a DNSActor, which holds a list of the master nodes that exist in the network. This DNSActor initiates and creates a master/landmark node in the current node. Initially: The DNSActor is deployed and it creates the first master node in the same device. Then slave nodes start registering in the network:

1. When a new slave node arrives and asks to register, it will first contact the DNSActor and request the list of master nodes. It then waits until the request is fulfilled.

2. The DNSActor responds with the list of master actors in the cloud.

3. The slave node makes ping requests to every master node. The requests are done concurrently. Since we are only interested in finding the closest master, we can continue when the first ping request is completed. We also utilize a threshold variable, namely latencyThreshold. If the new node is too far from every other master node in the cloud, then it should not join any of them but act as a new master itself. Since this threshold limits us, we can timeout the ping requests if we exceed it. If no master responds in time, then we proceed to step 4, otherwise to step 5.

4. The current node will act as a master node itself: It contacts the DNSActor and asks to be registered as master node. The DNSActor then remotely deploys a master actor in that node.

5. The node registers itself to the closest master node and maintains a pair of variables: (masterActorReference, latency).

6. The master actor checks the number of nodes registered in it. If it exceeds a threshold, namely the slavesThreshold, then it must split into two master actors, to make sure scalability will be guaranteed. In this case we pick a slave node from its list to act as a master and we ask the DNSActor to deploy a master actor there remotely. The slave actor now gets this new actor as its master, as it runs on the same device and it is probably closer. Also, the old master sends concurrent requests to its registered slaves to decide if they will move to it or not. We aim at sending about half of the nodes to the new master, in order to achieve good load balance, while we also want every slave node to end up in the master closest to it. However, there is a trade off between an optimal distribution and the load for the network. Even so, we ask all the registered nodes to ping the new master and decide if they will move to that (if it is closer) or not. Then we remove those that have moved, while they register...
in the new master node. This method does not guarantee an even distribution of nodes among the two masters, however it guarantees that we will get the smallest possible latencies. Also, the slave node running in the same device as the new master will now

On an attempt to make the algorithm more comprehensible, we present the two main procedures, registration of a slave node and split of a master node in pseudo-code:

Slave node asking to register - Algorithm running on slave node

initialize.timer;
for each masterActorReference in mastersList concurrently do
    send ping request and wait;
    if timer > latencyThreshold then
        timeout = true;
        break;
    end
end
if fulfilled then
    latency = now - timer;
    timeout = false;
    break // (algorithm continues when the first ping reply received – no need to wait for the rest as this is the closest master);
end
if timeout then
    send request to DNSActor: "deploy MasterActor on myActorReference";
    send request to DNSActor: "mastersList.add(myActorReference)";
else
    closest master is the first that replied: closestMaster = (masterActorReference, latency);
end
Master node accepting registration (and possibly splitting) - Algorithm running on master node

mySlavesList.add(newSlave);
if mySlavesList.size > slavesThreshold then
    send request to DNSActor: "deploy MasterActor on mySlavesList(1)"
    and wait for reply(which is newMasterReference);
    for each slave in mySlavesList do
        send request to slave: "ping the newMasterReference and move
    there if newLatency < latency" and wait for reply willMove;
    end
    if willMove == true then
        mySlavesList.delete(slave) (slave node will call the register
    method towards newSlave);
    end
end

We can also have information about whether a slave node’s device can support a master running there, however it is not implemented and it is left as future work. In this case, we can simply avoid asking to create a master in a
Furthermore, we can easily support a different slavesThreshold for each master node, according to its capabilities. It is not implemented, so it is left as future work.

The figures 4.1, 4.2 and 4.3 respectively depict the process and message sequence for a slave registering on a master, a slave declaring itself as master if there is no existing master with an acceptable latency and a split of a master when its number of slaves exceeds the slavesThreshold.

![Diagram of slave registration process](image)

*Figure 4.1.* The slave actor has requested the list of master nodes from the DNS Actor. He then sends a ping message to each master node(1,2,3). They reply back(4,5,6) and as soon as he receives the first reply he acknowledges that node as his master(7).
Figure 4.2. The slave actor has requested the list of master nodes from the DNS Actor. Since he does not receive any reply within latencyThreshold, he declares himself as master and informs the DNS Actor to deploy a Master Actor there(7). The DNS Actor deploys(8) the Master actor and receives the actorReference as a reply(9).

Figure 4.3. If the number of slaves registered on a master exceeds slavesThreshold, the master will split: It picks the first slave on its list and asks(1) the DNS Actor to deploy a master actor there. The DNS Actor will deploy it(2,3) and return the Actor reference(4). The master will then ask all the slaves to ping the new master(5,6) and move there if their latency is smaller than the one with this current master(11).
4.5 Master and Slave nodes

In our algorithm, we have a master-slave model where the master node(s) will do the scheduling and the slaves will handle the computational tasks. The master nodes also have to maintain the list of slave nodes registered there. Our implementation assumes, for simplicity, that every node has similar architecture and capabilities and thus could pick any of the two roles, master or slave.

One question that arises is what the optimal ratio of master and slave nodes is. There is no perfect answer to this question, however it is obvious that the more slaves a master has to handle, the higher the load it will have to handle. This load refers to both the scheduling task, as well as the communication handling and message passing for the distribution of tasks itself. So, although we generally need as many nodes to be slaves, as it is those that will process the actual Internet of Things tasks, we do not want to risk having the master slaves reach their computational limits. Another parameter we should consider is that the master slaves might not have the same computational capabilities with each other, so they cannot handle the same number of slave nodes. However in our implementation, as a matter of simplification, we consider all the nodes to have the same processing capabilities and we have a generic threshold for the slave nodes a master can handle.

4.6 Scalability analysis

The master nodes are safe when we scale up the system. There will always be a limited number of slaves they will have to handle (always below the slavesThreshold), so task distribution and communication costs will be easy to tackle. So, when new slaves arrive, we simply add more master actors. However, in this implementation, as a slave node wanting to register will ping all the master nodes present in the system, every master node will have to serve a big number of simultaneous ping requests which could be a problem in scaled-up systems, where massive amounts of new nodes come to register. Although the cost of receiving and responding to a ping request is very small, still if huge amounts of simultaneous requests appear, the master node will have to strive to respond. This problem could be tackled by only providing the slave nodes with a limited amount of master nodes to ping for the registration.

On the other hand, DNS actors are not scalable. All nodes asking to register will have to ping a DNS actor. So, a DNS actor might be asked to serve huge amounts of simultaneous connections, which will also require an equal amount of accesses to its data structures to read or update. One possible solution would be to have many DNS actors, however the problem is that we will need to
synchronise them so they will all have an up-to-date list (otherwise we would have more than one networks, with few or none nodes joining them). This will be left as future work.

4.7 Time complexity analysis

We estimate the time complexity of the different parts of the algorithm, namely adding a node and splitting a master node.

When a new node comes to register, it will send one message to each of the m masters to measure latency, and they will send one message back. Then the new node will send one message to the master where it will register. Thus, the complexity for adding one slave node is $O(m+m+1)=O(n)$ as the number m of masters is always close to n, the overall number of nodes, or $m > n / \text{slavesThreshold}$.

When splitting a master node into two, he will pick a slave from his list of t ($\leq \text{slavesThreshold}$) registered nodes to serve as a new master and then he will contact that one node (1 message) that becomes a master and all the other slaves as well (t-1), to ask them to ping this new master (t-1 requests for latency measurement + t-1 replies). So we have $O(1 + (t-1) + (t-1) + (t-1)) = O(t)$ messages. This t can be either close to the overall number of nodes in the system (n) (if there are few master nodes) or very small compared to n (if there are many master nodes), or in the average case relatively smaller. Thus in the average case $O(t) = O(n)$.

<table>
<thead>
<tr>
<th></th>
<th>worst case</th>
<th>average case</th>
<th>best case</th>
</tr>
</thead>
<tbody>
<tr>
<td>register slave</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>split master</td>
<td>O(n)</td>
<td>O(t)</td>
<td>O(t)</td>
</tr>
</tbody>
</table>
5. Results

5.1 Results

As described above, we do not have a system to compare ours with. Thus, the best we can do is execute different scenarios and compare the results, in order to find the optimal selection of parameters introduced in the final algorithm, namely the `slavesThreshold` and `latencyThreshold`, for different numbers of nodes. The most demanding and time consuming part of the algorithm is, as explained earlier, the split of a master node into two, triggered on a master which, after a new slave registers, now exceeds the slavesThreshold nodes registered there. Thus we want to avoid the blocking calls and additional message exchanges the splitting process involves. So, better results will be considered the ones with the minimum splits. Yet excessively increasing the slavesThreshold is not an option because, as mentioned in 4.5, the master node would struggle when tackling the task distribution and when maintaining all those slaves registered on him.

Additionally, the overall number of messages sent is the main indication of the cost of the algorithm, as these messages are all overhead. Also, we correlate the number of messages with the number of slave nodes.

As the latencies are randomly generated, the number of splits for the same number of slave nodes will not always be the same.

As we can see in table 5.1, the number of messages sent is generally proportional to the number of splits we have. This is more obvious in the table 5.2, where we have a fixed number of slaves but different numbers of splits.

<table>
<thead>
<tr>
<th># of slaves</th>
<th># of splits</th>
<th># of messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
<td>40</td>
</tr>
<tr>
<td>30</td>
<td>6</td>
<td>516</td>
</tr>
<tr>
<td>60</td>
<td>10</td>
<td>1271</td>
</tr>
<tr>
<td>90</td>
<td>16</td>
<td>2431</td>
</tr>
<tr>
<td>120</td>
<td>21</td>
<td>3813</td>
</tr>
<tr>
<td>150</td>
<td>25</td>
<td>5565</td>
</tr>
<tr>
<td>250</td>
<td>45</td>
<td>14650</td>
</tr>
</tbody>
</table>

Table 5.1. Number of splits and messages for different numbers of slaves
<table>
<thead>
<tr>
<th># of slaves</th>
<th># of splits</th>
<th># of messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>10</td>
<td>1271</td>
</tr>
<tr>
<td>60</td>
<td>11</td>
<td>1356</td>
</tr>
<tr>
<td>60</td>
<td>17</td>
<td>1442</td>
</tr>
<tr>
<td>60</td>
<td>20</td>
<td>1501</td>
</tr>
</tbody>
</table>

**Table 5.2. How the number of splits affects the number of messages for a given number of slaves**

Thus, in order to get the optimal performance from our system we need to have a big slavesThreshold so that we minimize the number of splits. However we should be careful so that no master node has to handle task distribution for an excessive amount of slave nodes, as this would cause problems. Thus a cost function that calculates the slavesThreshold should consider that, if it is too small, then we have a big number of splits, which adds a lot of overhead, while a big slavesThreshold value means that the masters might struggle when handling the slaves registered there and perform bad or become bottlenecks. In a real system we should choose the slavesThreshold parameter by carefully considering the capabilities and limitations of the device that runs as a master in handling the computational and communication tasks required for handling the slave nodes. In our testing system we got a glimpse of such a problem when, for a big number of slave modes, approximately 300, the testing machine almost ran out of memory and the messages exchanged were being stacking in the queue and delayed due to the difficulty to serve their increasing number.

Were we not using this heuristic technique to reduce the communication cost, a total number of over $n^2$ messages would be required to be sent, where $n$ the number of nodes, as every node would have to contact every other node. Now, every node will only ping the master nodes instead. Thus, we have a big gain as the number of messages required is greatly reduced. However, for $n$ nodes registering, the complexity remains $O(mn)=O(n^2)$ as $m > n$ / slavesThreshold, and the system might not scale well when $n \to \infty$. 

35
To conclude, in this thesis a cloud network that can process Internet of Things data in an efficient manner is described. The key points are the use of the actor model to build it, as well as the use of a heuristic technique for latency estimation between any pair of nodes in the network. Akka and the actor model in general proved to be a strong tool in modelling the system, as they serve the requirements for a platform independent solution that supports for code migration and simple tasks. Also, the feature of remote deployment of actors came in handy with several applications in this thesis’ code. All the same, the average latencies between nodes that will provide the data and the devices that will process them are guaranteed to be remain low, while little effort is required to accumulate information about latencies between any such pairs nodes. So, we can easily pick suitable processors without wasting time in calculating latency values between numerous pairs of devices.

Furthermore, despite the simplification in our assumption that all nodes can potentially serve as masters and that they all have the same processing capabilities, this can be altered in future implementations without much impact on the system description. However, a weak point of the implementation is the scalability of the DNS actor, where normally redundancy is required, plus more DNS actors to better handle the big numbers of requests. Yet another comment is that on the problem of latency estimation there is little bibliography available.

Overall, though, our system is guaranteed to provide small average latencies and small bandwidth costs with low overhead in preprocessing, and, combined with a smart scheduling algorithm, can constitute a solid framework solution for processing Internet of Things data.
There are many aspects of the problem and the solution that could be considered when defining some future work. To start with, an optimization for the scalability and fault tolerance of the DNS actors would be necessary. Not only is it a single point of failure, as a fault of the DNS Actor will make the system unable to accept new nodes, it is also the only part that will not scale well, since in the case that numerous new nodes ask to register, the DNS actor will fail to support the multiple connections and requests for the lists of master nodes or for remote deployment of master actors.

Support for removing nodes from the network should also be added. Although removing nodes was outside the scope of our thesis, it should be supported in order to make the implementation more solid. And while removing a slave node would only require terminating its actor and updating its master’s list, deleting a master actor would require asking its slave nodes to move elsewhere and also notifying the DNS Actor to remove it from the masters list.

Another improvement would be returning only an optimized amount of master nodes of the Masters list as a response from the DNS node to a slave node that wants to register. We are currently returning the complete list, which means that a slave actor will have to do numerous ping requests, which, although done concurrently, still mean increased strain for the network and the master nodes. However we should think it thoroughly before assigning work strain to the DNS actor, such as calculating an optimized output of the Masters list.

Fault tolerance should also be introduced in order to make the system more solid. Redundancy of DNS actors and master nodes should be available, while synchronization in such solutions should be handled carefully. In the case that the Master node fails to deliver the task distribution schedule, then we should arrange for a backup master node or reassignment of the process so that we will definitely get some result.

Finally, a more improved version of the system should be taking latency fluctuations into consideration. The latency between two nodes could change for any reason, hence reorganizing the network by readdressing the master-slave relationships would be necessary in any case.
References

http://dl.acm.org/citation.cfm?id=1985625.1985645
http://www.cisco.com/web/about/ac79/innov/IoE.html
http://www.forbes.com/sites/quora/2013/01/07/
Aug. 2014.