Universidade de Aveiro
Departamento de
Electrónica, Telecomunicações e Informática
2017

Miguel Chagas Bilhau Machado

Sistema de Monitorização baseado em Fog Computing
Monitoring System based on Fog Computing

Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática
2017

Miguel Chagas Bilhau Machado

Sistema de Monitorização baseado em Fog Computing
Monitoring System based on Fog Computing

"Permanence, perseverance and persistence in spite of all obstacles, discouragements, and impossibilities: It is this, that in all things distinguishes the strong soul from the weak."

— Thomas Carlyle

Miguel Chagas Bilhau Machado

Sistema de Monitorização baseado em Fog Computing
Monitoring System based on Fog Computing

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requesitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica de António Rui de Oliveira e Silva Borges, Professor do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro e com a colaboração técnica de Eng. António Manuel Silva Oliveira da WithUs, Lda.

o júri / the jury

presidente / president	**Professor Doutor Luís Seabra Lopes** Professor Associado da Universidade de Aveiro (por delegação do Reitor da Universidade de Aveiro)
vogais / examiners committee	**Professor Doutor António Rui de Oliveira e Silva Borges** Professor Associado da Universidade de Aveiro (orientador)
	Doutor Pedro Lopes da Silva Mariano Investigador da Faculdade de Ciências, Universidade de Lisboa(arguente externo)

Agradecimentos

Tudo o que alcancei durante a minha vida devo, em parte, ao apoio incondicional da minha familia e amigos, a eles desejo o melhor. Em particular, queria realçar a importância dos meus pais e avós pelo amor e pela educação que me concederam. Quero ainda agradecer ao meu mentor, o Dr. António Borges e ainda ao Eng. António Oliveira, pelo aconselhamento e visibilidade concedidos durante a realização do meu trabalho. Por último, fico grato pela disponibilidade do Eng. João Reis que me auxiliou na concretização da *prova do conceito* nos protótipos das tomadas inteligentes.

Acknowledgements

The achievements throughout my life could not happen without the support of my family and friends. I wish them the best. I would specially like to thank both my parents and grandparents for their love and guidance. Finally, I would like to thank my supervisor, Dr. António Borges and technical advisor Eng. António Oliveira, for their advisement, knowledge and sponsorship. They have my utmost respect. Ultimately, I would like to express my gratitude towards Eng. João Reis, from WithUs Lda. which helped me deploy the intelligence required to perform the *proof of concept* in the *plug prototypes*.

Keywords fog computing, cloud computing, Internet of Things

Resumo

Este documento apresenta uma arquitectura como solução para o desenvolvimento de uma camada extra de poder computacional entre os serviços na núvem e a Internet das Coisas, denominada de computação no nevoeiro. Esta camada é responsavel pela gestão e recolha de dados provenientes de conjuntos de sensores, geograficamente distribuídos, em níveis inferiores. Assim, o nevoeiro permite servir como ponto de agregação comunicando directamente com a núvem, minimizando a quantidade de tráfego na rede. A solução descreve a camada de nevoeiro como um conjunto de grupos de nós que se agrupam e organizam como um todo, autonomamente. Existem ainda mecanismos auxiliares que permitem a existência de um certo grau de tolerância a falhas de forma a manter o *status quo* do sistema em ambientes ubíquos, lidando com as constantes alterações de contexto. A solução foi testada e validada atravéz de uma prova de conceito onde foram realizados três casos de teste, concebidos de forma a abranger todos os componentes da mesma.

Keywords fog computing, cloud computing, Internet of Things

Abstract This thesis is a contribution of an architectural solution, describing a system that represents an extra layer of computing power, placed between the cloud and sensor networks, acting both as a mediator whose central task is to manage, monitor and collect data from geographically-located groups of sensor nodes and as a communication hub to the cloud with which data is exchanged in a compact and minimalist fashion. The latter is accomplished by designing nodes as autonomous entities, able to organise themselves in smaller groups, within the system. Additionally, these entities possess inherent mechanisms which aim to accomplish fault tolerance within groups of nodes, maintaining the *status quo* of the overall system while performing in an ubiquitous environment, continuously embracing contextual changes. The overall solution was tested in a proof of concept where we conceived three test cases that helped us validate it.

Contents

List of Figures

List of Tables

List of Acronyms

ACID Atomicity Consistency Isolation and Durability.

ADT Abstract Data Type.

ARPA Advanced Research Projects Agency.

ARPANET Advanced Research Projects Agency Network.

CM Communication Manager.

DARPA Defense Advanced Research Projects Agency.

E2E End-to-End.

FIFO First In First Out.

FNC Federal Networking Council.

FSM Finite State Machine.

GPU Graphics Processing Unit.

GUI Graphical User Interface.

HB Heartbeat.

HONet Hybrid Overlay Network.

ID Identifier.

IMP Internet Message Processor.

IoT Internet of Things.

IP Internet Protocol.

IPv6 Internet Protocol version Six.

LAN Local-Area Network.

M2M Machine-to-Machine.

MANET Mobile Adhoc Network.

MTU Maximum Transmission Unit.

P2P Peer-to-Peer.

PLC Power-line Communication.

RFID Radio Frequency Identification.

SaaS Software as a Service.

SSH Secure Shell.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

UI User Interface.

UID Unique Identifier.

URI Uniform Resource Identifiers.

VANET Vehicular Adhoc Network.

WBAN Wireless Body Area Network.

WSAN Wireless Sensor Area Network.

WSN Wireless Sensor Network.

YAPPERS Yet Another Peer-to-Peer System.

Chapter 1

Introduction

The present document is a reflection of the acquired knowledge throughout a year's work. The elaborated solution emerges as a consequence regarding the Internet enhanced by "things". "Things" because we face a new paradigm regarding computational capabilities of a widening range of smart devices. In turn, the internet is acquiring new kinds of end-points which resulted in the Internet of Things (IoT) where computing is regarded to be everywhere, ubiquitously.

Moreover, the increase in quantity regarding these devices, augments the complexity within such environments, thus, new assets will appear as an opportunity to achieve higher quality in services by providing dedicated intelligent controllers closer to these environments. The paradigm which envisions the allocation of processing power at the edge of the network and beyond is commonly addressed as Fog Computing [Open Fog Consoritum (2017)]. In addition, it can be regarded as a system deployed to control the pervasiveness in IoT environments, providing dedicated services.

Furthermore, we direct our approach towards a Fog Computing System capable of monitoring the electric grid by processing and collecting data from such environment. In turn, these can be used as an end-service taking advantage over the real-time *awareness* of Fog Computing. Additionally, we envision a larger system, composed by a Cloud, Fog and IoT. To the former, we delegate the aggregation of data from the Fog nodes, meaning the Cloud can provide *Software as a Service* and *Infrastructure as a Service* from a global scope, regarding the Fog nodes as a whole. Another approach towards the Cloud in respect to the large amounts of storage it can provide, i.e. *databases*. We can take advantage of the Cloud as a huge database while we assign the Fog control over context-oriented intelligence, managing the pervasiveness in IoT environments.

This thesis is a collaborative effort between WithUs[1] and the University of Aveiro[2] where the former presented a problem that could be considered for a master thesis research. This exposure from WithUs takes place due to the constraints in current IoT scenarios and the associated constraints in current designs. The increased complexity and heterogeneity between IoT contexts turns Machine-to-Machine(M2M) communication an hard process. Thus, the growing interest towards the development of a system, under the Fog Computing paradigm, where it is possible to disregard the technological constraints in relation to communication interfaces and dedicated hardware, by increasing the software complexity on each Fog node.

[1] http://www.withus.pt/
[2] http://www.ua.pt/

Ultimately we determine if this solution can be an appealing trade-off by successfully managing itself, as an infrastructure, while communicating over the Internet Protocol version 6 (IPv6) within its core.

1.1 Historical Context

Technological advancements regarding computational systems during the second half of the 20th century evolved to what is considered to be a digital world. The development of devices, capable of performing computations proved to be a huge step towards the future. After the first digital computing systems started to emerge, the commitment regarding resource storage and sharing, began to prove a crucial aspect in relation to these systems. The first notion of a global scale network of computers was envisioned by Licklider [Lickleider and Clark (1962)], as a concept of a widespread network of computing systems. Computers would be interconnected in a way where everyone could access resources with ease, despite their geographical location. This kind of medium would allow individuals to collaborate on different areas contributing by their own to a higher purpose. Although, this was only grasped as a concept, it led to a technological revolution. The first step towards achieving a "Galactic Network" started with the ARPANET [Leiner et al. (2009)], a project whose goal was to interconnect distant computing systems. In regard to its origins, as most leaps in technology throughout the human history, this project was, in its early stages, a military initiative, more specifically, from the Department of Defense of the United States of America. Moreover, an official document from this agency, regarding ARPANET [Defense Communications Agency (1978)], states the following:

> "The ARPANET is an operational, resource sharing inter-computer network linking a wide variety of computers at Defense Advanced Research Projects Agency (DARPA) sponsored research centers and other DoD and non-DoD activities in CONUS, Hawaii, Norway, and England. The ARPANET originated as a purely experimental network in late 1969 under a research and development program sponsored by DARPA to advance the state-of-the-art in computer inter netting. The network was designed to provide efficient communications between heterogeneous computers so that hardware, software, and data resources could be conveniently and economically shared by a wide community of users. As the network successfully attained its initial design goals, additional users were authorized access to the network. Today, the ARPANET provides support for a large number of DoD projects and other non-DoD government projects with an operational network of many nodes and host computers."

ARPANET is a project where different individuals came together aiming to achieve this common goal. Research on different areas of computing provided the knowledge and resources to make it possible.

Upon getting acquainted with the concept expressed in the paper "Intergalatic Computer Network" envisioned by Lickleider, Lawrence G. Roberts developed interest towards the latter which led him to design a conceptual network of computers[Roberts (1967)]. Later on, Roberts and Thomas Merrill made a *proof of concept* where TX-2 computers were connected in mass to distant Q-32 nodes. Additionally, the medium used was a low speed telephonic line. The results shown that the concept regarding the netting of computers was appealing

and doable, although, circuit switched telephone systems proved to be inadequate to perform programmable computations and retrieval of data within the network[Leiner et al. (2003)]. Interestingly enough, this experiment is regarded as the first deployment of a wide-area network. Furthermore an alternative was presented regarding the medium of communication. This was a study published by Leonard Kleinrock at MIT, regarding packet-switching[Kleinrock (1961)]. It suggested a more efficient approach towards resource sharing where communication would be established through the forwarding of packets instead of signals in a circuit. Packets would be split into chunks which in turn would be independently routed between the nodes in the network, from the source to their destination. These breakthroughs suggested that computing systems could interoperate, performing vast amounts of programmable computations, whether running programs or retrieving data. Also, Kleinrock's work proved to be more adequate regarding this network paradigm since the circuit switched telephone system was limited in regards to its physical capabilities. Other breakthroughs were achieved like the Internet Message Processor (IMP), a module used to switch packets between interconnected nodes, this allowed the envisioned concepted by Licklider to become a reality. Granted this foundation, the networking concept started do be used mainly between the military and academic institutions. As the previous quote states, "As the network successfully attained its initial design goals, additional users were authorized access to the network", suggesting that the successful implementation of this kind of network gained visibility in other sectors, the advantages that this technology brought services like finance, government and other private sectors to adopt it. Overall, the various interested entities provided the means to develop new technologies due to the strength of the investments in the computing area. After these early stages, technology advanced in every aspect. From a size-wise perspective, computers, in general, suffered a reduction in production cost and size, becoming available to a wider range of end-clients. Additionally, as the portability increases, so does the processing power, resulting in a self-sustained development phenomenon. The more appealing and widespread technology gets, the more developed it becomes. New sectors bring new contexts into what is regarded as the interoperability of the network. For instance, finance systems began to adopt the Internet as a mean to interconnect different infrastructures, bringing new opportunities and services available for their clients. By interconnecting computational devices, client record and account information could be shared between end-points, facilitating the overall provisioning of the banking services. Even so, not everything regarding technology is an advantage. As the computational systems grow, so does the inherited complexity granted by the *Black box* concept. Furthermore, the advancements in technology, led to the development of new resource-sharing tools. The World Wide Web(WWW) was developed in the last decade of the 20th century, where Tim Berners-Lee was the main protagonist in such achievements. The aim was to meet the needs towards automatic information-sharing between scientific communities. Thus, the goal of the WWW and the following browsers was to present information on-line by making documents and information globally available through the use of Uniform Resource Identifiers (URIs) providing ease of access. Following these developments, the "opportunity pattern" emerges once again. The development of new technologies and ways to distribute information lead to a self-sustainable environment which one can state the following: *The development of new technologies, ultimately leads to the development of better and newer technologies in a near future.* This is only possible because computational systems are designed with abstract layers over abstract layers, i.e. *Black Box*;

What is meant by the latter is that the acquainted interest regarding a technology brings new development opportunities that have not been discovered or met so far. Additionally,

taking the example of the first browsers, by developing new ways of structuring and presenting information enhancements could take place. For instance, the concern over the displaying of information in a browser, which led to the development of Graphical User Interfaces (GUIs). In turn, this generated the need to develop dedicated hardware able to create visually appealing interfaces. Hence, the first Graphical Processing Units (GPUs) originated. Its easy to understand the concept beneath this. Computer science is a self-marshalled, self-fulfilling area where the technological advancements breed new branches for enhancements. This aggregation of technologies now represents what we recognize as "The Internet" nowadays. In 1995 the Federal Networking Council passed on a resolution that stated the following:

> "The Federal Networking Council (FNC) agrees that the following language reflects our definition of the term Internet. Internet refers to the global information system that – (i) is logically linked together by a globally unique address space based on the Internet Protocol (IP) or its subsequent extensions/follow-ons; (ii) is able to support communications using the Transmission Control Protocol/Internet Protocol (TCP/IP) suite or its subsequent extensions/follow-ons, and/or other IP-compatible protocols; and (iii) provides, uses or makes accessible, either publicly or privately, high level services layered on the communications and related infrastructure described herein."

The internet focus is the sharing of resources and services, enabling developers approach technology from an higher abstraction level. In order to provide a strong context regarding the solution within this thesis, one considers the interconnection between the origin of the internet and what it represents in modern society to be deeply connected with current and future solutions. Technology, overall, evolved so rapidly that any hand-held device has more processing power and storage than computing systems which occupied entire buildings when ARPANET was founded. These enhancements made clear the need of new solutions regarding resource access and sharing.

The first distributed systems emerged as a consequence of the Internet's increasing complexity. Some of the early approaches where through grid computing, using parallel computing systems to achieve a common goal. Moreover, utility computing appeared as a solution to provide resources through metered services, but, the rise of software as a service (SaaS) led to subscriptions under applications via the network. Developers can now disregard the means to deploy and maintain the services and can solemnly focus on the development of new services and tools. Ultimately, we face ourselves under a new paradigm, *Cloud Computing*. Different entities, like IBM, Google, Cisco and Amazon, mainly proposed the latter to achieve high-performance computing and resource sharing as an utility, providing software as a service (SaaS). Thus, cloud computing originates as a solution towards a wider range of services. The concern over the infrastructure and service availability can be discarded, meaning that any developer can adapt the dimension of the infrastructure, required to provide a service, as the demand grows or diminishes. Thus, the Cloud can be seen both as an application which delivers a service and also a scalable infrastructure that can be adjusted according to user demands. Many solutions have been deployed, providing services to end-points like laptops, desktops, and other terminals. These, provide dedicated services given the intended purpose of the client, most of these provide a quality of service deemed fit for their clients, until recent years. Moreover, the Cloud is not foreseen as an obsolete system. Instead, we address the Cloud as a fitting solution which can handle huge amounts of information, improving the overall services provided. *Data Warehousing* is one of the main roles the Cloud can represent

in future solutions, bringing different contexts together, exploiting the whole system as a single entity, with smaller, self-managing components. Technology is at a stage where it is possible do mass produce small programmable devices able to process and communicate. The reduction, in size, of devices has moved towards a reality where it is possible to enhance objects with intelligence whether they are moving vehicles, furniture, appliances or any other object, i.e. "thing". In contrast with *cloud computing*, the vast amounts of devices and complexity will turn out to be a huge constraint towards the use of *SaaS* and *Data warehousing*. The same features that make the cloud a suitable candidate for many business applications are, simultaneously, handicaps towards the quantity of processing and data generated beyond the edge of the network. The increase in number of *computational objects*, led to the paradigm of the Internet of Things. It represents the future of the internet as the latter is not only a medium comprised by computers and data centres, instead, there is no assurance in regards to the end-point we communicate to, thus, heterogeneity. It can be anything due to these developments in electronics. It is possible to imagine our surroundings to be enhanced with devices able to perform programmable computations. *Smart cities*, *Smart Homes*, generically speaking, a *smart grid*, resulting in a pervasive computational environment. Hence, the need for a pervasive infrastructure able to control, monitor and ultimately provide better services, will require the deployment of dedicated system in relation to a certain context. At this point, the Internet of Things indicates that ubiquitous computing must be supported by an adjacent structure which handles the complexity of these environments, providing increased quality in services. Hence, we reach Fog Computing as a solution to meet those demands. A new paradigm able to handle the pervasiveness in these scenarios. Fog Computing is seen as a model which can enhance the Cloud-IoT continuum through the allocation of dedicated nodes, i.e. controllers, able to provide diverse and dedicated services at the *edge of the network* and beyond. The distribution of context-oriented nodes, able to filter the collections of data generated by "things" will permit the creation of different levels of abstraction between the Cloud and the IoT. Besides, the development of applications and services can be approached from two different levels. At the level of the Fog, one sees distributed nodes with specific intelligence intended to process the data collected from "things" while from the cloud, one can see the "bigger picture" regarding the whole system, taking in consideration the various clusters existing beneath it. From the cloud perspective most of the decision-making processes are dedicated to the Fogging systems which will allow shifting cloud applications towards a more sheer management over the whole system, due to the gain of contexts below each Fog node.

1.2 Thesis Goals

The main goal is to present a system capable of gathering data while managing the "Things" in the Internet. As such, we propose an architecture inspired on state of the art distributed systems. The goal is to deploy intelligent nodes capable of communicating among them. Nodes are devices that perform programmable computations, cooperating to reach a stable infrastructure in Peer-to-Peer fashion. Additionally, this architectural design was made under the assumption of a real application in order to provide a development context. This was the electric grid monitoring. Even so, the solution itself is not bound to any specific scenario or application as it can be adopted by most, if not all, IoT scenarios. Also, Fog Computing is a paradigm basis which can be used as a conceptual starting point in order to develop a solution.

The enhancement of nodes with fault tolerant mechanisms turn possible the ensurance of the *status quo*. These are capable of handling node *crashes* as well as message *delays* and *losses*.

- **An Architecture** Inspired by current distributed systems, oriented to a peer-to-peer design;
- **An Autonomous Entitiy** The system has entities which form smaller groups and organise themselves autonomously. These are responsible for the delivery of services to the IoT layer.
- **Fault Tolerance** The system can react to *crash* scenarios where nodes become offline. Also, faults over network links cannot jeopardise the operation of nodes.
- **An Operational Protocol between Nodes and Sensors** The former must be capable of receiving data from the latter data as well as maintaining the service, while there is at least one of the former operating.

1.3 Thesis Structure

State of the Art - Chapter 2 We will analyse the State of the Art of the various technologies involved in the development of a system under the Fog paradigm. Firstly we discuss Distributed Systems as a model which can help design the infrastructure over the Fog nodes. We also try to distinguish different models by classifying them and subsequently exploring the concepts behind them. Moreover we explore the networking infrastructure, specifically, P2P systems. These can be seen as an asset towards achieving a network distributed system. We will explore the taxonomies involved and, which advantages and disadvantages can be found towards the different designs. Additionally, we try to extend each classification to the layers involved in the overall solution of this thesis by making comparisons to the P2P designs and what we expect to achieve in the Cloud-Fog-IoT continuum.

Furthermore, we discuss fault tolerance and ways to achieve such as well as intrinsic mechanisms, which can provide the means to deliver a reliable system. At the end of this chapter, we will have a brief discussion over sensor networks which are connected to the design of the IoT environment, where Fog nodes are deployed.

Engineered Solution - Chapter 3 On the chapter that follows, we present and explain the design of the conceived solution. This in turn, is supported by the different States of the art acknowledged previously. We address the solution from a top-down perspective where we will take a glimpse over the Cloud, Fog and IoT interconnection. Even so, the main focus of the thesis is towards the development of a Fog Computing monitoring system, thus, we will direct our efforts towards explaining the infrastructure involved at the Fog layer. Which are the capabilities of each node, the associated intelligence and how the infrastructure itself can be self-maintained without requiring human intervention. After closing the Node behaviour and intelligence, we explore the induced mechanisms that guarantee the stability of the status quo regarding the Fog. This will lead us to discuss the fault tolerance mechanisms introduced within the system providing the means to achieve a reliable infrastructure.

Proof of concept - Chapter 4 On the 4th chapter we will present the methodologies and tests cases which served as an experiment towards the proof of concept. We will approach different scenarios to explain the various aspects involved in the node's behaviour. Additionally we try to develop a generic scenario which can be associated to a real environment. The results are presented as a consequence of these scenarios and each test case is supported by the outcome of each simulation.

Conclusion - Chapter 5 We close the present document with a reflection over the achievements and setbacks throughout research and development. Additionally, we also suggest possible areas of improvement or enhancement in regard to the overall solution.

Chapter 2

State of the art

Throughout this chapter we overview the current solutions required to build a distributed system over the Fog Computing context. Besides distributed systems, we will grasp Fog Computing as a design concerning the pervasiveness on IoT environments, while providing services to wireless networks and how both can cooperate, in order to achieve a system capable of providing functional pervasiveness. Additionally, the study of mechanisms that guarantee such characteristic are also fathomed. The state of the art regarding these subjects is supported by different authors and constitutes an insight perspective over what is a first-person understanding over the subjects.

2.1 Distributed systems

The capability to reach a solution is largely determined by how well the concepts and designs are fathomed. As such, in order to provide a most suitable context, the following quote is a categorical description regarding the definition of distributed systems[Tanenbaum and van Steen (2004)]:

> "A distributed system is a collection of independent computers that appears to its users as a single coherent system."

Distributed systems over the years were built aiming to achieve *functional separation*, *reliability*, *scalability* , *transparency* and *economic factors* [Tanenbaum and van Steen (2004)]. These characteristics are crucial if one desires to provide an homogeneous service despite the end user. The scalability of a system is always related to the non-exclusiveness of its nodes, more specifically, a node is exclusive if its functionality is unique in relation to other nodes. It is uncommon a distributed system designing unique roles. Hence, distribution and scalability are inherited through *functional separation*, guaranteed the means to do so. Another crucial point is high availability, a characteristic shared among reliable systems. A system is reliable if there under the occurrence of faults, its services are still being provided correctly (definition is presented on section 2.3 page 19). This, in turn, leads us to *transparency*. A system must be capable of providing a service or set of functionalities without requiring knowledge, regarding its internals, from outside processes. This means that a consumer process, accesses a system interface, provides an input and receives the subsequent output, alike the *black box* concept.

Moreover, Tanenbaum and van Steen [Tanenbaum and van Steen (2004)] consider the taxonomy of distributed systems to be divided in three classes: *distributed computing systems*;

distributed information systems; *distributed pervasive systems*. We will discuss the former because it is a widespread design between business solutions like *cloud computing*. This will help us understand the connection between the Cloud and the Fog. Oppositely, *distributed pervasive systems* influence how *Fog Computing* fits the design of distributed system and how it can overlap the IoT scenario providing ubiquitous services *ad infinitum*. Since a *pervasive* context is always one of the assumptions that must be taken into consideration, the author decided to discuss distributed pervasive systems within the Fog Computing section(see section 2.1.2 on page 11) as a internal characteristic of the Fog. We will not discuss *distributed information systems* because they are not related to the solution we aim to achieve, nonetheless these can influence how the Cloud can operate. If we would consider to design a cloud-based solution, this type of system would need to be fathomed.

- Distributed Computing System, a system comprised by programmable nodes which cooperate to achieve a common goal;
- Distributed Information System, typically comprised by operations based on transactions. These systems perform under *Atomicity*, *Consistency*, *Isolation* and *Durability* (ACID) as a set of properties;
- Distributed Pervasive System, a system where computing is present ubiquitously, originating smart environments.

2.1.1 Distributed Computing Systems

This type of computing systems takes advantage of large groups of nodes by performing operations in parallel or collaboratively. The key is that, the system itself is comprised by nodes which share a common goal. Two specific approaches within this class are *cluster computing* and *grid computing*. The former method of computing consists on distributed nodes over a high-speed cable connection in a local-area network (LAN). A node performs synchronized work with other nodes on a parallel fashion. A design like this also requires a *master node* to manage resource allocation and work distribution. Figure 2.1 [Tanenbaum and van Steen (2004) Sec.:1.3.1 (page 17)] presents an example of the previous description:

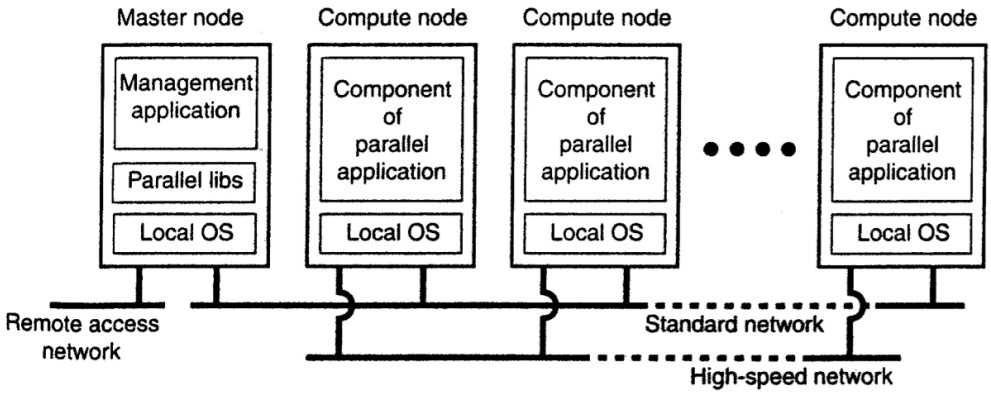

Figure 2.1: *Cluster computing system exaple*

Whilst *cluster computing* systems are typically *homogeneous* in regards to their physical characteristics, in *grid computing* the belonging nodes are quite the opposite.
Grid computing is a model which focuses on assembling different resources, making them available to different end user groups so they can collaborate with each other. The latter is achieved by forming *virtual organizations*[Tanenbaum and van Steen (2004)] in order to control and assign access rights over the assembled resources. Additionally, Foster [Foster et al. (2001) proposed an architecture regarding grid computing systems, figure 2.2 was taken from [Tanenbaum and van Steen (2004) Sec.: 1.3.1 (page 19)].

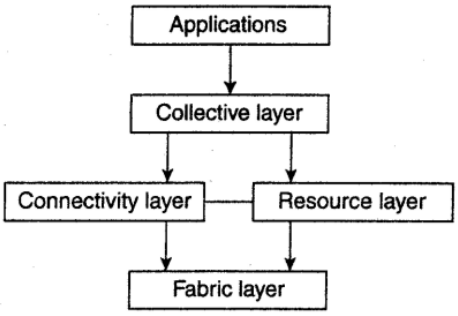

Figure 2.2: *A layered architecture example regarding grid computing*

As Tanenbaum and van Steen enunciate, the fabric layer is responsible basic for operations over resourcesm such as, status and means to find and manage those resources. At a higher stand point, the *Connectivity layer* supports the access and communication required to share different resources and data. This layer is also responsible for security regarding communications and authenticity. Within the same layer, the *resource layer* is responsible for the access of resources. By operating with the *connectivity* and *fabric* layers it provides an access control regarding the data, allowing its configuration and process delegation. Above the latter is the *collective layer*. It is responsible for the of scheduling and concurrency regarding the resources within a *virtual organization*. At the top level, the *application layer* where different processes take place regarding the specific access rights of an *organization*..

2.1.2 Fog Computing

Fog Computing emerged through the founding of the *Open Fog Consortium*, integrated by Cisco, ARM Holding, Dell, Microsoft and Princeton University, among others, in November 2015 [Open Fog Consortium (2017)]. There are earlier references which will also be taken into consideration, although, in order to grow towards a goal, we consider that different organizations must come together and agree on a direction alike the ARPANET. Thus, we try to gain insight on Fog Computing by considering the same scenarios the Consortium made public through their reference document.

Fog Computing is defined by an horizontal architecture, which distributes resources and services throughout the network [OpenFog Consortium (2015)], acting as a middle layer between IoT and the Cloud continuum. Fog Computing requires support towards different layers of application domains, delivering intelligence to businesses and users, allowing an enhancement of the IoT contexts in regard to the Cloud [OpenFog Consortium (2015)]. Although,

the primary feature resides in allocating processing power to the edge of the network, *edge computing* and *Fog Computing* are not the same thing. The reason for such difference resides on the fact that *edge computing*, in its core, provides services to users by pre-provisioning resources at the edge of the network. On the contrary, Fog nodes can be deployed on the edge of the network and beyond providing a dynamic pool of resources between devices [OpenFog Consortium (2017)]. The application of *Fog Computing* can be set according to our surroundings through wired or wireless networks. For instance, radio access networks are a great example which allow functional control over the IoT scenario, allowing the distribution of contexts and operation in overlapped environments. This means that the Fog approach, does not only aim to reduce latency problems through the *in loco* basis but goes further than that. Fog Computing is a mean to sustain basic operations and data collections, by managing small computing devices, giving them purpose. Business intelligence, operational and reliability issues are also important factors in the Fog development by deploying *Fog nodes* throughout the environment. One special characteristic about these nodes is the reduced dependency regarding human control. Thus, operating solemnly on the awareness of the environment.

Another subject shaped by this class of computing is the continuous shift of entities interacting within a network and sub-networks of "things". For example, a car is a moving object, if aware of its environment by connecting to surrounding sensors and vice-versa then, there will be an instant where such object will have to communicate with surrounding networks in order to send/receive and process data. Upon leaving, both must once again detect the contextual change and react accordingly. Moreover, Grimm grasped the requirements over *distributed pervasive systems* [Grimm et al. (2004)] which are the following:

- Embrace Contextual Changes;
- Encourage ad hoc composition;
- Recognize sharing as default.

This leads us to consider these categories as internals regarding a pervasive system.

Embrace contextual changes

The purpose of the cloud, under this paradigm, can be seen as an infrastructure which collects data from the Fog Computing layer and assimilates it. The cloud displays a passive behaviour, meaning that communication will mainly be made through an uplink direction, from the Fog layer. Clearly, this happens because most of the intelligence and data aggregation happens within the Fog. Contextual changes are not usual since pervasiveness exists below the Fog, mainly in IoT environments. At such level, contextual changes are numerous.

Fog Computing must be able to scale accordingly. This means that at any time, devices may connect or disconnect to the Fog without precedent notice. This type of contextual change suggests that within the system, devices must be able to recognize each other and react accordingly. Such behaviour is similar to Peer-to-Peer (P2P) networks where computing systems within a network form groups of peers to achieve a common goal. Moreover peer-to-peer network will be discussed (see section 2.2 on page 14). Embracing contextual changes can only be achieved if:

- There is a network infrastructure able to support real-time processing that this type of pervasiveness requires, typically *ad hoc* wireless networks;

- Sufficient coverage regarding nodes connected to the *Fog* system;

- Nodes must possess sufficient capabilities to guarantee their internal state invariant in order to provide a *correct service* (later on we will define what a *correct service* is).

Regarding the IoT continuum, more specifically, sensor networks, their origin is due to the aggregation of devices where the only guarantee is that there is no restriction in their behaviour. Contextual changes are always happening. Devices may enter and leave each network without notice. Thus, any crucial part of processing and data storage must be transferred to a stable infrastructure. Sensor networks, IoT environements overall, are characterized by the infinite possibles of service provisioning to end-users. Besides the contexts existing presently, the augment of new technologies will generate newer contexts where intelligence must be deployed. Ultimately, we can observe different, overlapping systems, handling resources from different contexts. In turn, these will have to favour *ad hoc* composition in order to co-exist.

Encourage ad hoc composition

Due to the constrains made clear by contextual changes, it is obvious a the service must be available despite such factors. A system with such requirements must be developed taking in consideration that its composition will change at runtime. The interposition between applications and networks must be a simple process in order to easy the correctness of the services. The sole aim here is to take advantage of reusable behaviours in a way that changes within the network and the system would act as a plug and play feature, keeping the overall state stable continuously. A good example of this type of composition is *ad hoc* wireless networks, discussed in section 2.4 on page 24 .

Recognize sharing as the default

In order to achieve a perfectly balanced structure, able to interact with heterogeneous computing systems in a standardized way it is crucial to define interfaces able to carry out required operations. What is meant is that independently of the device specifications, there has to be set of operations able to satisfy interactions between nodes where different requirements take place. Additionally, uniform ways to communicate and homogeneous data formats are needed in order to ensure that communication between distinct devices is made transparently, without the need to take in consideration different methods of delivering a service, through different technologies.

Fog Computing is all about providing context, making use of the IoT scenario. It can also be a solution with much potential towards achieving intelligent environments. On the following section we take a look into sensor networks and how they are deployed. What is the state of the art regarding *ad hoc* networks and how can these, under the IoT Fog scenario, help to build a suitable infrastructure able to provide the services and performance required. Finally, in order to guarantee transparency, communication protocols must be developed according to interfaces in order to provide an uniform service despite system specifications [Tannembaum and van Steen (2004)].

2.2 Peer-to-Peer Systems

So far, we have become familiar with *distributed computing systems* and *distributed pervasive systems* through the Fog context. This brings us to P2P. These systems are commonly used to share resources between different nodes spread across the internet. Many of these resources are located on the *edge* of the network, like the nodes from *Fog Computing*. Hence, P2P can be regarded as an archetype of distributed computing in a sense that it fits the description regarding the categorization of distributed systems. Moreover, algorithmic research regarding search algorithms has been a huge factor towards achieving high resource availability. P2P architectures, specifically decentralized ones, normally share features like self-organization and self-management. Additionally, Coulouris [Coulouris et al. (2012)] stated that P2P systems, despite the taxonomy, share the following characteristics:

- Any node, i.e. peer, can contribute to parts of the system or to the system as whole;
- Nodes must be homogeneous regarding their functionality despite the kinds of resources they may possess;
- The correct operability of a node happens disregarding any input from other entities, thus being autonomous.

A key issue regarding P2P, which determines efficiency, is how accurately data is load balanced accessed throughout these networks. Coulouris discusses Pastry and Tapestry as possible solutions towards *routing* between peers[Coulouris et al. (2012)].

Many P2P solutions focus on resource sharing solutions, e.g. BitTorrent [1] . Although, this is not the only capability of these systems. By taking advantage of different computational systems, a P2P architecture can be enhanced through the use of the different techniques from different designs.

Peer-to-Peer systems have different taxonomies. These can be centralized or decentralized. The latter focuses on collaborative efforts, in a way that a portion of their resources is made available to other components. The former is resource-oriented, this means that centralized systems often do not depend so much on *routing and search*.

2.2.1 Taxonomy

From a top-down perspective, the taxonomy is devided as centralized or decentralized. In centralized architectures there is a central server answering requests in this type of architecture. The aim behind this approach resides in the fact that peers, in order to retrieve information must establish connection to a central server which will then provide a tracker list where the required resources are available. This lacks scalability and robustness mainly due to a single point of failure. Solutions like BOINC [2] and Napster[3] use this type of architecture to provide their services. Besides centralized architectures, decentralized ones have a widened range of implementation possibilities. This type of architecture only grants each peer a partial view over the network. The key part in this type of architecture resides on the different approaches towards building a network with such backbone. Depending on the system purpose, a P2P

[1] `http://www.bittorrent.com/`
[2] `https://boinc.berkeley.edu/`
[3] `http//napster.com/`

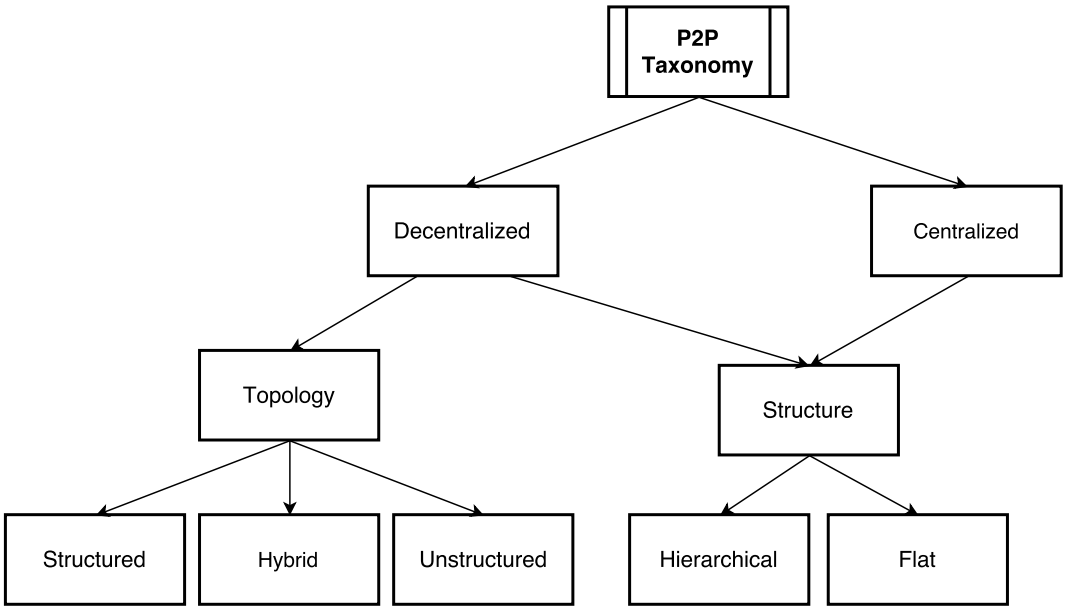

Figure 2.3: *P2P networks taxonomy*

architecture can provide the means to achieve robustness, security, fault tolerance and also avoid single point of failure. Let us discuss the structures and topologies that can be taken into consideration within a decentralized architecture. The topology of a P2P network can be structured or unstructured. The main difference between these approaches is the way queries are forwarded between nodes in the network.

2.2.2 Centralized Systems

As previously mentioned, the centralized P2P model consists on a network of peers with document lookup directed towards a centralized server [Ding et al. (2016)].

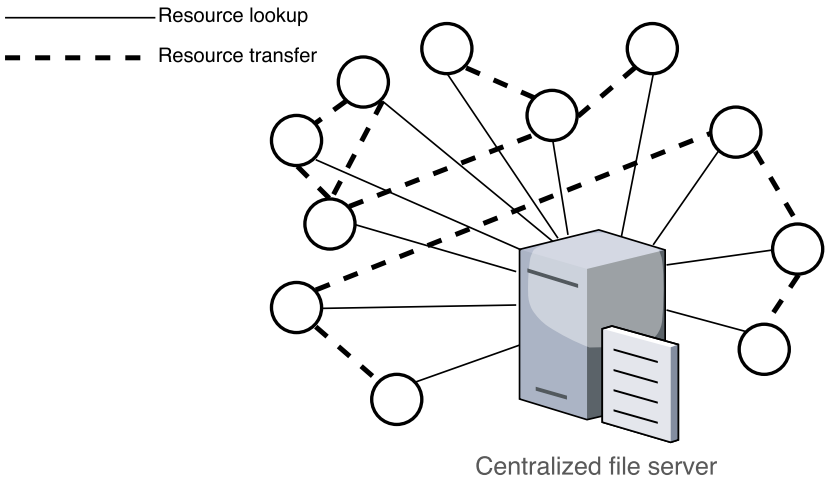

Figure 2.4: *Centralized P2P architecture*

This server will in turn offer a list of peers where the intended resource is located. The main property behind this interaction is that a centralized server would offer such list but his functionality would strictly be so. All subsequent resource transfer is established between peers. This kind of approach can be seen as an encouraging solution towards the structure that must be established between the Fog and the Cloud. By applying this model, the Cloud would manage attached peers in order to oversee operations and data collections. Peers in turn, would communicate with each other and would only recur to the cloud if demanded.

2.2.3 Decentralized Systems

In unstructured topologies every peer has to maintain a dataset with possible neighbours where queries might be sent. These structures are internally hierarchical in a way that, depending on the approach taken regarding the topology, there is always a core within the system that connects distant parts of the system.

Unstructured Network Topology Each peer is also responsible for its own data. The major disadvantages dwell on the fact that: "locating" resources might turn out to be a complex process since there is no direct way of knowing where these are; "uncertainty" regarding response times and the completeness of the answers for the queries that have been sent. This means there is no guarantee that a resource, within the network, will be found.

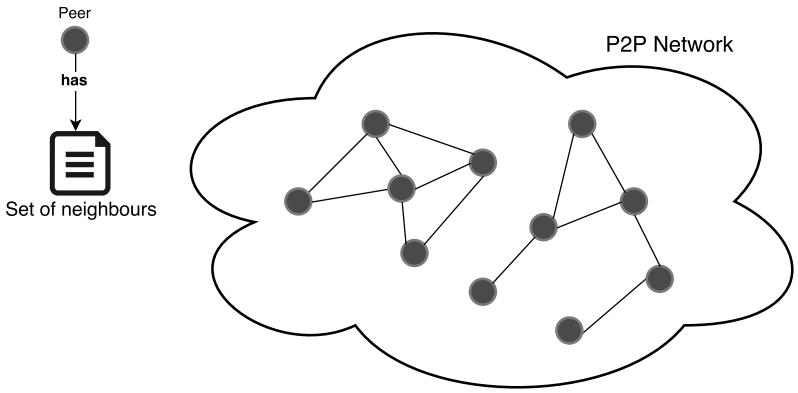

Figure 2.5: *Unstructured P2P network*

A great example of an unstructured decentralized architecture is the Gnutella system. The discovery process, in earlier versions, was based on flood routing strategy. This lead to an increase of traffic throughout the networks. Also, as the network scaled over, obtaining a successful answer shown to be harder. In order to circumvent this problem, a new approach was taken towards the organization of the system itself. A two layer hierarchy was introduced in order to optimize queries between peers. Peers would be categorized as *ultra peer* and *leafs* [Zhonghong Ou (2010)]. The distinction between layers was based on connecting neighbourhoods, i.e. *groups of leafs*, into a *ultra peer*. Queries would be shared between the same neighbourhood and communication between different groups of peers is established only between the *ultra peers*. Note that an implementation like this would require some kind of reactive mechanism in order to

tolerate faults regarding the *ultra peers*. This could be achieved through *leader election mechanisms* which will be discussed later in this chapter (see section 2.3.7 on page 23).

Structured Network Topology By using this architecture as backbone, a system is aware of the location of the resources. Distributed hash tables are commonly used due to their efficiency regarding indexing. This clearly makes queries more responsive and the guarantees towards receiving an answer increases. The trade-off between a structured and unstructured approach resides on the fact that in order to keep track of the data, more storage will be required overall within the system. This type of topology suggests that in a system where pervasiveness is expected, the costs to maintain such variants might cause serious setbacks in managing the system itself. Examples of a structured P2P topology are Chord and Can, both described by Lalitha and Subbarao [Lalitha and Subbarao (2012)].

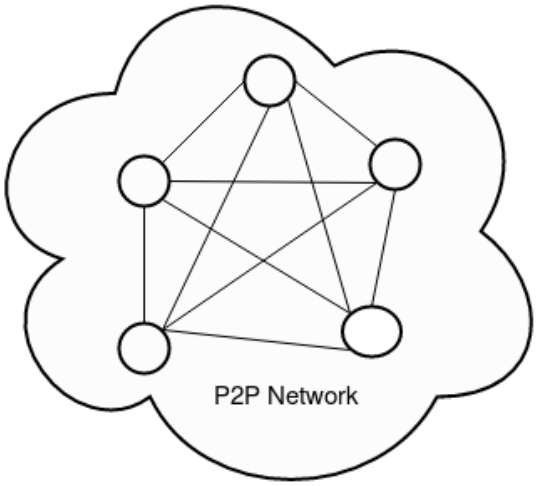

Figure 2.6: *Structured P2P network with a Star topology*

Hybrid Network Topology This model, combines both unstructured and structured topologies aiming to take advantage of the benefits from both approaches. An hybrid scheme, proposed by Ganesan [Ganesan et al. (2003)], introduced a P2P network, *Yet Another Peer-to-Peer System* (YAPPERS). The lookup service of this design consisted on grouping nodes neighbourhoods in what the authors considered to be *immediate neighbours*. Data is similarly spread in a Distributed Hash Table fashion regarding these nodes and if queries between the same neighbourhood would fail, communication would be forwarded to other neighbourhoods, which the authors considered to be *extended neighbourhoods*. The forwarding process happens as long as no successful query returns or every node in the network queried. The key factor in this approach resides in what the authors also considered to be *buckets*. Resources, as mentioned, are stored in a DHT fashion, although, the data set is built considering resources,i.e. *buckets*, of the same kind, thus, queries are sent only to nodes where there is a chance the desired resources might be stored.

Another application worth of studying is the Hybrid Overlay Network (HONet) [Tian et

al. (2005)]. This topology was based on a two-level hierarchy where peers where assigned two different roles and grouped together. Each group has a *gateway peer*, besides the standard functionalities, these peers would also act as a gateway towards other groups of peers, i.e. *gateways*. The remaining peers within a group are common nodes without any other special functionalities. Each group, i.e. *cluster*, has independent sets of identifiers between peers of the same cluster, thus, it is possible to distinguish between peers of the same or different clusters. Additionally, connections are arranged through a *random walk algorithm*. In case of failure, the connection between peers is arranged taking in consideration the hierarchical structure of the *gateways* and adjacent peers. In order to visualize the HONet, take in consideration the following figure:

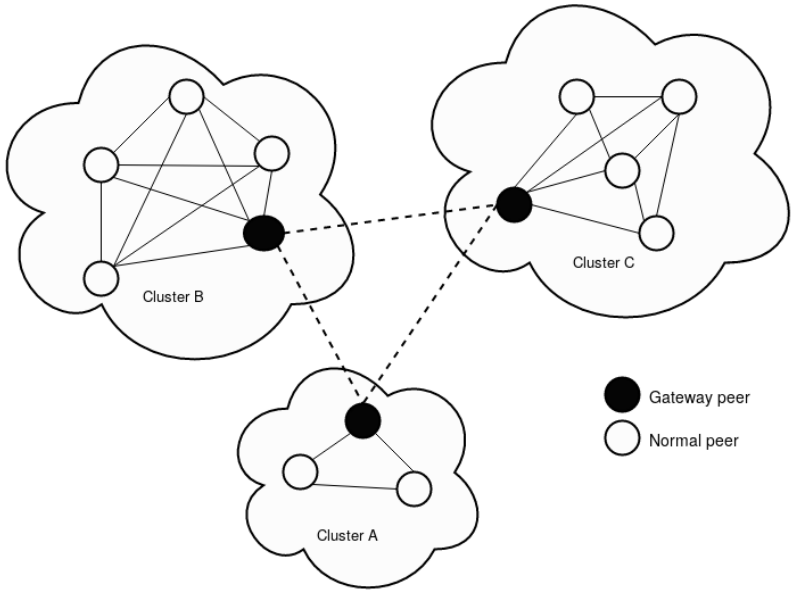

Figure 2.7: *HONet P2P network organization*

This kind of hybrid structure allows the interconnection and cooperation between different groups increasing the redundancy in connections which can be important in cases of major fault, i.e. all peers in a group. The replacement of a service in an infrastructure like this can be made due to the fact that super peers are aware of each other, thus, they can perform in order to stabilise fault situations.

2.2.4 Resource Storage and Sharing

One of the main concerns regarding infrastructures of this kind is on how does storage of data is made in order to become available throughout the network. Specifically, how can technology evolve in order to maintain a structure able to handle the pervasiveness of the generated data. Kubiatowicz [Kubiatowicz et al. (2003)] suggest an architecture, "*Ocean Store*", which aims to persist storage on a global-scale. This architecture is comprised by untrusted servers, thus, additional measure must be taken in order to protect such data. The approach to solve this was through the means of cryptography and redundancy techniques. Although the goal of this thesis is to build a distributed system which can cooperate with

ubiquitous computing, we will not focus on data availability. Nonetheless, Kubiatowicz [Kubiatowicz et al. (2000)] approach the main goals that must be taken into consideration to develop such solution.

2.3 Fault Tolerance

Fault tolerance has a major role in distributed systems. A system that becomes tolerant to critical faults, according to its specification, is a system which will have high availability thanks to its robustness. In order to achieve fault tolerance, redundancy blocks must be introduced within the system. This enables it to replace services without loosing availability, in most cases. Before further discussion, take in consideration the following definitions. A *server* is a computing system that provides a *service*, which is is the operation over an input that according to specification may produce an output. Following this context, a server is *correct* if, in response to its inputs, behaves in a consistent manner regarding its functionality. This means that any expected output, when presented, if incorrect, then a *failure occurred*. Nonetheless, the omission of a result, when expected, is also failure. Thus, there is a need to extend the classification over types of failure.
Furthermore, Kirrmann [Kirrmann (2005)] describes fault tolerance as being composed by two sub-categories. One defined *error masking*, i.e. *fault masking* and another, *error recovery* i.e. *fault recovery*. Although one must recognize the difference between such categories, we will discuss *fault recovery* and *fault masking* through *replication*, system *feedback analysis* and *leader election algorithms*. Besides *recovery* and *masking*, sometimes *awareness* over a process or system is also required, thus, moreover we will approach *heartbeating* as a solution to achieve latter.

2.3.1 Fault Tolerance Taxonomy

The taxonomies considered on this document are inspired by Cristian [Cristian et al. (1991)], which we will summarize in this section.

Fault classification

So far we have captured two situations where a server might be prone to failure. The first regarded the correctness of the output produced by a service. The latter was related to the omission of such answers when expected. Hence, there are different types of faults that can happen within a system, so, in order to fully grasp which faults may occur, but first, we must define a taxonomy able to describe such faults.

- *Omission failure* - A server fails to produce any outputs.
- *Timing Failure* - Happens when the service does not provide an output within a specified real-time interval.
- *Response Failure* - The output produced is incorrect in regard to the expected outcome.

Crash classification

Furthermore, if a server suffers an *omission failure* and subsequently continues to omit results until restarting it is said to suffer a *crash failure*.

- *Amnesia* - Restarts to a predefined stable state that does not depend on previous inputs.
- *Partial amnesia* - System reboots and its state is only partially defaulted.
- *Pause* - After reboot, the server returns to the last previous invariant state before the crash.
- *Halting* - The crash causes the server to never restart.

2.3.2 Fault Assertion

Fault detection mechanisms can detect the *correctness* of a *service* while operating correctly, *but*, they sometimes might lack the correctness of the reason which originated the fault. To make this evident, consider the situation where a fault detector Heartbeat (*HB*) and a subscribed process p communicate through a link l. If p sends his heartbeat periodically, but l becomes lossy and drops packets, then *HB* will detect a fault regarding p. Although, this fault is an *omission fault*, the *omission* itself was caused by the link. Meaning, the cause is not p but instead l. The lack of assertion towards which kind of fault originated the subscript process to stop sending messages is dubious. The problem is not on how *HB* operates. Instead we suggests the following: Accuracy towards asserting the reason of a fault is crucial to how one can deal with the problem. Possibly we add redundancy to the fault detection system, regarding multiple access points, where constraints over links can be avoided. This would improve the success regarding the delivery of messages.

2.3.3 Fault Masking

Failure masking consists on the concealment of misbehaviours to outside entities, due to errors or faults. This means when a fault occurs the system itself must become aware and provide the means for a stable transition between the instance before the *failure* and the moment after it has been captured and handled. In turn, this concealment allows any dependent process to keep its operations, without disturbance. Defensive approaches towards fault tolerance normally consists in *replication* and *status feedback* of the system modules.

2.3.4 Replication

Fault tolerance through replication is achieved, within a system, when integral components have one or more replicas which will act as backup in relation to a primary service(see figure 2.8 on page 21). Another perspective towards replication is homogeneous distribution of roles within different processes or components of a system. Take for instance server groups, where each belonging node is performing specific functionalities. If these functionalities are unique, then if a crash occurs within a node, the system itself will suffer a considerable amount of damage due to the specific functionalities of such node. In order to avoid this, a flat distribution of the nodes, as seen in unstructured P2P networks, can help overcome this problem. By replicating functionalities, i.e. roles, within a group of nodes, a system can ensure that in case of failure, if substitution can take place, it will.

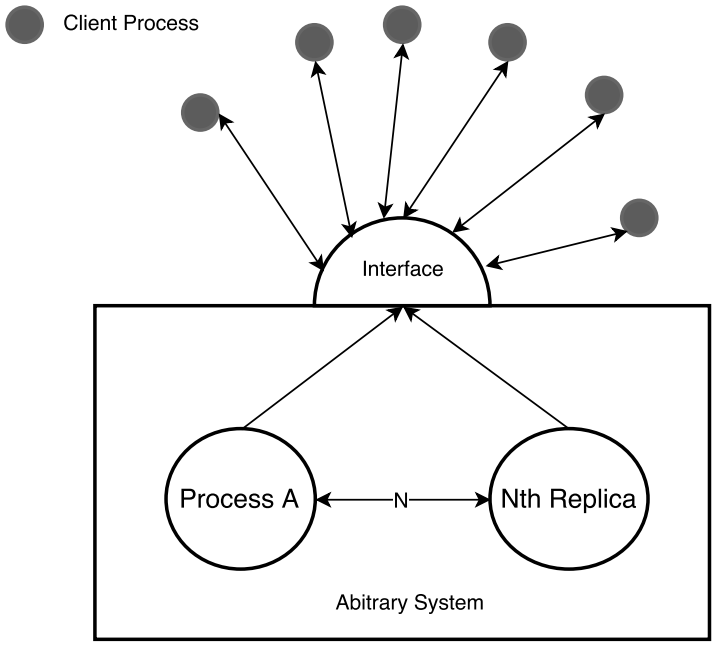

Figure 2.8: *A system with a replicated process*

2.3.5 Preemptive Status Feedback

Another approach to achieve fault tolerance is the use of some kind of feedback provided by a system in order to handle any error that might occur. Under this paradigm, many solutions exist in order to provide a way of handling with fault occurrences. One example of an approach based on feedback is *exception handling* regarding programming languages [Weimer and Necula (2008)]. Moreover, a generic description regarding feedback analysis can be enunciated as such: *A system, is capable of taking action to handle and recover from faults if there is a proper mechanism which provides sufficient information towards the manifestation of such events, thus, making it* ***aware*** of faults;

This approach is commonly obtained through the use of state machines, where the sate and adjacent outputs and inputs must be taken into consideration, allowing the system to provide useful feedback to outside, dependent, systems in order to become *aware* of faults or misbehaviours. This is extremely useful in real-time systems where decision-making must be based mainly on local information due to the fact that This makes clear that *replication* techniques must also use feedback from the system's processes in order to activate backup services. Thus, replication is only obtained if some kind of feedback is granted. Another example of a feedback based mechanism is *heartbeating*, which will be discussed in detail moreover.

Figure 2.9 on page 22 illustrates a process A actively reporting his state to a system, or even another process. The example bellow the latter is the opposite scenario. A client system or process, actively queries another process or system about its internal status. Both approaches can be taken regarding *heartbeating* which we will discuss next.

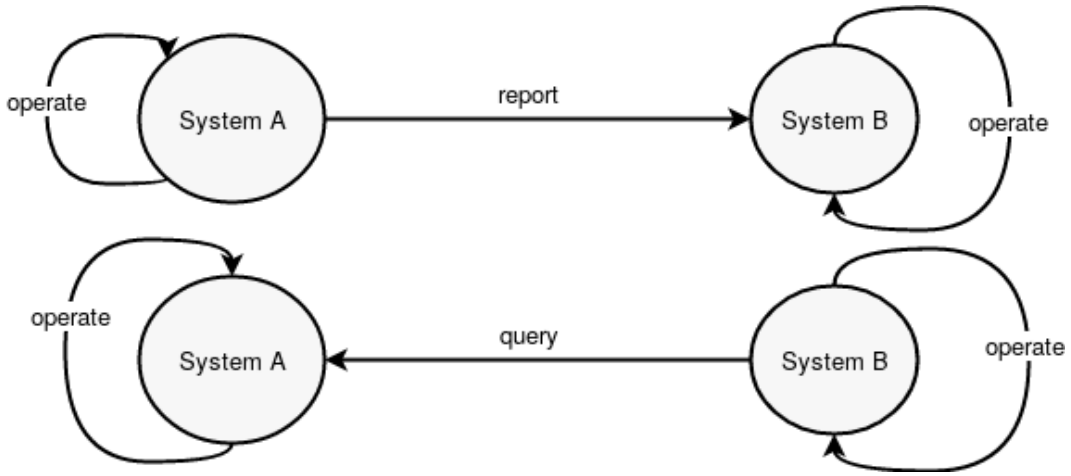

Figure 2.9: *Active/passive feedback supply of system A status*

2.3.6 Heartbeat Mechanisms

Heartbeating aims to guarantee the quiescent reliability regarding communication between message-passing systems. Before further discussion, it is important to grasp two typical situations regarding the integrity of the communication. The communication link may be prone to message loss. This in turn, annihilates any guarantee regarding the delivery of a message. The assumption that if a system a send infinite messages to another system, b, will improve the chances of its reception is completely misleading. A scenario like this only guarantees that communication will fail and there is no such mechanism able to circumvent this problem. Another wrong assumption is that the link is *fair*. This means, if a sends infinite, m, messages through a link, l directly implying that b would receive indefinite messages is obviously this is impractical and efforts have been made to correct such problem [Aguilera et al. (2005)].

The (Non-)quiescent factor

Keep in mind the last scenario. A solution resides on a basic principle: upon reception, b returns another message to a that indicates the *acknowledgement* of m [Aguilera et al. (2005)]; This allows communication to become *quiescent*. Besides losses between communication links, there is also the system *crash* factor. Whilst the quiescent factor is definitely guaranteed assuming that only losses occur in the link, a crash of a process while communicating with another process will invalidate that characteristic. As such, *unreliable failure detectors* have been proposed as a solution to solve this problem. A failure detection mechanism consists on periodically receiving an *alive* message from a *subscribing* process, i.e. *"heartbeat"*. The reception of such message will indicate the "wellness" of the process and the mechanism will increment the value of the counter associated to that *subscriber*. This means there has to be a list where *subscribers* and the associated counter are stored. Problems like consensus, atomic broadcast, group membership are solved with heartbeat mechanisms.

An important thing to note is that the fault detection mechanism itself is not quiescent. The operational protocol dictates that any processes subscribed to the fault detection mechanism must periodically send *heartbeat* messages while alive. However, is it reasonable to use,

non-quiescent, *unreliable failure detection* mechanisms in order to achieve fault tolerance in distributed systems so it becomes *quiescent*? In fact, it is, in order to envision such argument, recall the previous section where we approached fault tolerance, we defined a *server* and *service*. A *service* is correct *if* none of the faults described before occur, resulting in the following requirement: a process is *correct*, if and only if, it will present valid outputs within a realistic time period, according to its inputs. Ultimately,

Operational protocol

Some approaches are based on *bounded* counters whilst others use *unbounded* [Aguilera et al. (2005)]. The difference between *bounded* counters and their counterpart is the fact that usually *bounded* counters act like trigger mechanisms, if a certain bound is reached, a meaningful context arises and the fault detector updates its suspect list, a set of data where the crashed elements are maintained. The counter process may increment when receives an *heartbeat* messages, but, it may also use decrements upon sending one, depending on the approach. The *unbounded* counter is a much simpler mechanism. It will increment the values nonetheless, but due to the *unbounded* characteristic of its counter, the value never reaches a *bound*. Instead, it simply outputs the counter values and client applications will give meaning depending on their purpose.

2.3.7 Leader Election Algorithms

This subset of algorithms aim to solve another problem that was not mentioned so far. In systems where nodes interact continuously and maintain a data structure regarding their organization, a fault that occurs in a node leads to inconsistencies on that same data set. This type of failure led to the development of election mechanisms with intent over re-establishing a faulty scenario like such. Consider a P2P system where peers with shared purposes, form small clusters within the network. In order to maintain the stability and interconnection with new outbound requests, a leader must be elected. In the rest of this section we will discuss what we consider to be the most prominent algorithms. Also, in order to achieve a fully understanding over the algorithms, assume that the failure event always occurs in the leader of that group.

> **Bully algorithm [Garcia-Molina, 1982]** In a group comprised by N nodes, if a certain node K detects a failure, an *election* message is sent by multi-cast to all the nodes whose *Unique Identifier* (UID) is higher than the UID of K. This means K is holding an election. If the nodes whose UID is higher are alive, they will reply to K with *ok* and they will hold an election themselves. This means an *election* message is sent according to the highest UID "rule". This process continues until only one node is remaining, which is the node whose UID is the highest. This node will then announce its victory and will become the new leader by broadcasting a *coordinator* message to all remaining nodes. If the previous, crashed, leader comes back online, it will send a *coordinator* message to his group in order to assume control over the group. Figure 2.10 was taken from the internet, thus we provide its source[4] .

An important aspect regarding this algorithmic approach is the fact that it guarantees the termination of the election process in a finite set of steps. This characteristic is crucial due to the fact that there functionalities attached to group mechanisms. Whether it is resource

[4] `https://www.andrew.cmu.edu/course/15-440-sp09/applications/ln/bullyex.jpg`

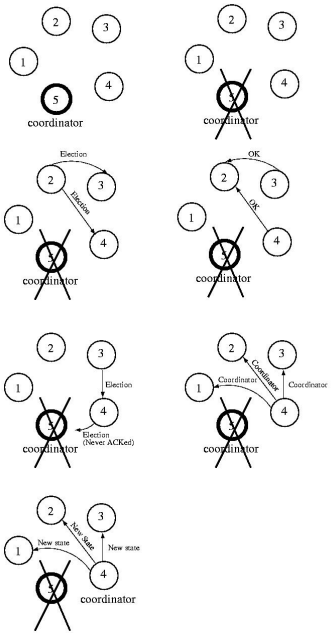

Figure 2.10: *Bully algorithm election process, left to right, top-down.*

sharing, parallel processing or other. Thus, it is crucial to guarantee that there is no *indefinite* delay regarding a process determinant towards achieving group stability.

2.4 Sensor Networks

Sensor networks are huge clusters of small embedded devices that may have other capabilities besides perceiving external stimulus from their environment. These sensors are limited regarding their processing power and energy supply as well as communication capabilities. The reason for such specification is due to the fact that these devices are built on the principle that must be small and energy-wise. Moreover, from a system perspective, these networks may be seen as small distributed databases due to the fact that many of these are deployed in applications whose purpose is to collect information.

Tanenbaum discuss [Tanenbaum and van Steen (2004)] that these sensor networks can be designed under two approaches. The first (1) requires sensors to send all their data through the network. The data is collected by the operator in the site and processes or stores it. The second (2) approach consists on a query-response model where the operator sends queries to sensors. The latter will, in turn, compute and answer according to its internal state and reply to it to the operator. An operator is a device which stores the overall information regarding the sensor network. Ultimately the coordinator maintains a dataset with the aggregated answers received. On (1) the single point failure is clear, data is solemnly sent to the operator. Regarding approach(2), the nodes comprising the sensor network might require additional capabilities in order to achieve their functionality and the operator only stores the answers[Tannembaum and van Steen (2004)]. Although, these solutions are feasible. Both lack the requirements to maintain balance within the network. They lack the constitution required for a system to be

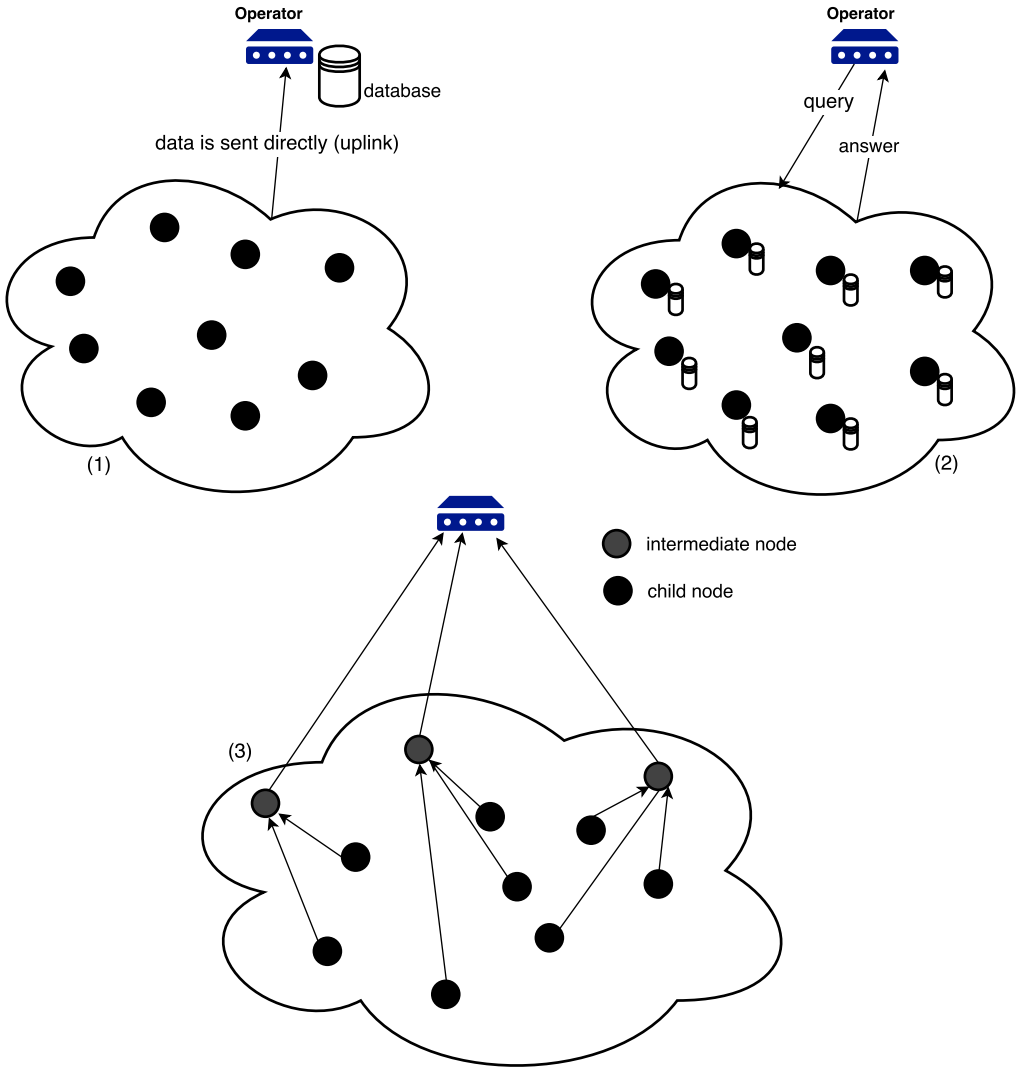

Figure 2.11: *Three different approaches regarding sensor networks*

---

able to scale. These disadvantages have been approached by TinyDB(3) [Madden et al. (2005)] partially solving them through the use of a declarative database interface in wireless sensor networks. It requires a tree-based routing algorithm to optimize the aggregation of data. This protocol is an hierarchy regarding the collected data, where higher layers, closer to the root of the tree, will have more aggregated data than the ones below. Still, some problems persist: there is no guarantee in relation to fault tolerance and in a scenario where IoT takes place it is not possible to determine *a priori* which of these sensors belong to which hierarchical layer.

In figure 2.11, (1) and (2) are suggested by Tanenbaum[5] as possible organizations towards sensor nets.

[5]Tanenbaum et al. 2004 - Distributed Systems: Principles and Paradigms Sec.: 1.3.3 (Page 30)

2.4.1 Internet of Things Role in the Fog

The internet of things, as a concept, dictates that any embedded device, with minimal specifications can contribute to a system as a whole, by gathering information from its environment. It is easy to realize that the possibilities are endless since modern society is already built on top of a digital world. The cause for such uprising in the technology is due to the reduction, mostly in the last decade, of the size and production cost of small computing systems. Take for instance smartphones, a top of the line nowadays is much more powerful than a desktop system from a decade ago. When the interacting entities within our surroundings become able to process and provide information, under a certain context, intelligent environments become a reality leading to the development of useful services which at a certain point will become intrinsic. The interconnection of the IoT devices can take huge advantage of dedicated computing systems, deployed to provide a committed service according to their contextual requirements. With such an increase of wireless devices the need for a scalable networking infrastructure [Bonomi et al. (2012)] able to support IoT as a system as become clear. This reason as lead researchers and manufactures, in a continuum space time, to consider ad hoc composition as an important element towards the development of wireless sensor networks. Ad hoc wireless networks are very useful to create small wireless areas where communication can be established between nodes.

2.4.2 Ad hoc Wireless Networks

The shift of the networking paradigm happened when computer systems started to communicate through wireless channels. By adopting new paradigms, computer science has redirected its focus, regarding networking, to envision devices as inter-connected nodes in a homogeneous fashion, enabling the development of networks associated to a context. There are several wireless technologies that allow ad hoc nets to become a reality, being RFID [Herschel et al. (2012)], Bluetooth [Bisdikian (2002)] and Zigbee [Hillman] some of them. Although, how are these networks structured and how they fit the requirements imposed by the IoT structure? The following classes of networks are a reflection over some of the existing ad hoc solutions, thoroughly discussed by Reina et al. [Reina et al. (2013)]. These will help us understand what is the state of the art regarding the subject, bringing a deepened perspective over the matter.

- **Mobile Ad hoc Networks(MANETs)** are characterized by being self-organized networks that do not require the need of a backbone. The purpose for the development of such networks lies in their main features categorized by self-healing, self-maintaining, self-configuring and self-repairing [Reina et al. 2013]. Thus, very suitable within a mobile context. MANETs are built according to the mobile paradigm, where pervasiveness is the main characteristic. In relation to IoT, MANETs are comprised by entities. These entities can be any computational system able to communicate through a wireless channel, disregarding the size. They also act as a router meaning that any entity is also a router within the network. The mechanisms mentioned before are achieved through a series of routing protocols. These protocols maintain the entity's routing table integrity. In order to do so, protocols use broadcasting and multi-hoping. This originates a lot of packet redundancy. There are some examples of solutions that tried to solve this problem by reducing the number of packets (Multipoint Relay and Connected Dominant Sets). Another crucial aspect about these routing techniques is the fact that they also help the entity to become aware of services and resources. A simple approach

towards achieving connections is the use of two different IPs. One is responsible for communicating through the MANET, the other establishes connection to the Internet. Although, due to the mobility paradigm, a gateway target may change thus invalidating the current IP configuration. There are other approaches, but all have drawbacks which will require further research in order to achieve a practical solution. Reina [Reina et al. (2013)] also analyse how service and resource discovery within such nets can be achieved. A deeper research was also conducted by Karagiannis [Karagiannis et al. (2011)]. Appliances in relation to vehicular networks (VANETs) can be taken into consideration being the following some of them: *Intersection collision warning; Lane Change assistance; Cooperative forward collision warning;* [Reina et al. (2013)]

- **Wireless Sensor Network (WSN)** unlike MANETs, this type of networks is designed based on efficient power consumption. The most common topologies implemented are tree or star-based. In order to achieve full efficiency, the nodes within a WSN typically send their data to a central device. Whilst in a start topology, nodes are at a distance of one hop, in a tree or mesh topology communication requires multiple hops between nodes. In most cases, this kind of network is comprised by static nodes meaning that the topology itself does not suffer changes, simplifying the organizational process. Since the common principle between IoT and Fog Computing is context, this type of network may be seen by two different approaches. Every node within a WSN is an entity, thus is distinguishable from other nodes which may have a different context, or, the WSN as a whole is seen as an entity, thus, the overall functionality provides the context. An enhanced approach within this class of networking is the Wireless Body Area Networks (WBAN) discussed in extent by Tanenbaum [Tanenbaum and van Steen (2004)] and Reina [Reina et al. (2013)]. Additionally, in the health care sector, Ko [Ko et al. (2010)] explore applications such as *Wireless Sensor Platforms* and *Medical Sensing*. Furthermore, the possibility to consider different contexts within the same network, although in order to do so there has to exist intelligence on higher layers that is aware of such differences.

- **Radio Frequency Identification (RFID)** has proven to be a technology capable of transmitting small amounts of data within short distances. The principle behind RFID consists in having two devices. One holds information regarding some context, i.e. tag, while the other is able to access it and read it, i.e. tag reader [Kaur et al. (2011)]. Regarding tags, there are two types. The first is called passive tag, the other, active tag. The former does not possess any kind of power supply, instead it relies on the energy transmitted by the radio signal sent by a tag reader. The latter, active tags, are proactive due to having a power supply. This technology is typically based on close range communication. Nonetheless, given the tags (both passive and active) and a tag reader, it is possible to build business intelligence by integrating a middle layer connecting to this technology. Different designs have been developed, called *near-field* and *far-field RFID*, also discussed in extent by Kaur [Kaur et al. (2011)].

Chapter 3

Engineered Solution

In comparison to other technologies, the electric grid has taken a toll due to the stagnation of advancements. Measuring devices and adjacent infrastructures have become outdated. These, continue to be largely dependent on the human factor in order to provide and maintain the services. Subsequently, hazardous situations endure due to the lack of fault tolerance and detection mechanisms within the grid. Factors like aging equipment and an obsolete system layout are deeply tied to the deregulated services the industry provides. This as been a reality thus far. However, augmented concerns over these facts have led manufactures and developers to explore new ways to shift the electric power supply delivery towards more sophisticated models. New solutions provide a higher cultural value according to the present *ethos*. Entities have already proposed *smart meters*, like WithUs [1] and Siemens[2]. These can enhance or even replace the current infrastructure of the electrical grid with a new distributed approach. Allowing these intermediate devices to communicate with other end-points clearly enhancing the quality of the services being provided by delivering dynamic services like adaptive flows of electricity according to the real-time consumption, hazard detection in the electrical grid. Additionally, the human factor is largely removed in relation to the management and maintainability of the electrical grid. This means a shift towards the approach of the services can occur by focusing on deploying dedicated nodes with intelligence able to continuously monitor the grid itself and gradually increase the services provided . Moreover, the dynamic allocation of electrical power supply also enables the energy provider to adequately adapt, the tariffs and associated power supply, to the client's needs. Clearly this eliminates the need for interrupting the service in case of exceeding the supplied power.

By taking advantage of currently developed solutions the aim of the design is to adapt the Fog into this context. Nodes will be deployed to collect information for surrounding sensors. This implies the need for a procedure where nodes are self-configurable. Additionally, each node is supposed to autonomously become aware of surrounding sensors and establish connection in order to collect data. These sensors proactively read information from electrical plugs, meaning that each sensor is in fact a *smart plug* which is *plugged in* into the *electrical* one. Thus, this brings us to how can one establish communication between devices. Current technologies are based on a master-slave topology. For instance, within a residence there has to be a master able to detect a set of these *smart plugs*. Node that the constraints of the visibility

[1] http://www.withus.pt/

[2] http://w3.siemens.com/smartgrid/global/en/products-systems-solutions/smart-metering/components/Pages/overview.aspx

between *master-slave* devices are dictated by the communication interface. The proof of concept used *smart energy gateway* [WithUs] with PLC, Zigbee and ethernet communication. Although, the interface connecting the sensors to the gateway was *radio-based* which restricts the range of the visibility, depending on the aperture of the antenna. Note that we were bound to the specifications of the equipment. Hence the discussion regarding *ad-hoc networks*.

These connections are based on the same principles of these networks. Devices with a radio interface (e.g. Zigbee) rely on ad-hoc composition to establish communication between these devices. Thus, it is granted that there are suitable solutions which allow low-powered devices to communicate. On the other hand, communication between a *master* device and the Internet is typically done through a different interface. For instance, in our proof of concept we used a *physical interface*, i.e. ethernet, in order to connect to a router. Additionally, regarding the IoT scenario, every device is considered to be a slave, requiring a connection to a *master* device in order to deliver the overall service. Take into consideration that the *controllers*, i.e. master devices, are responsible for the composition and management of *ad-hoc* networks which function as the bridge between the sensors and the controllers themselves.

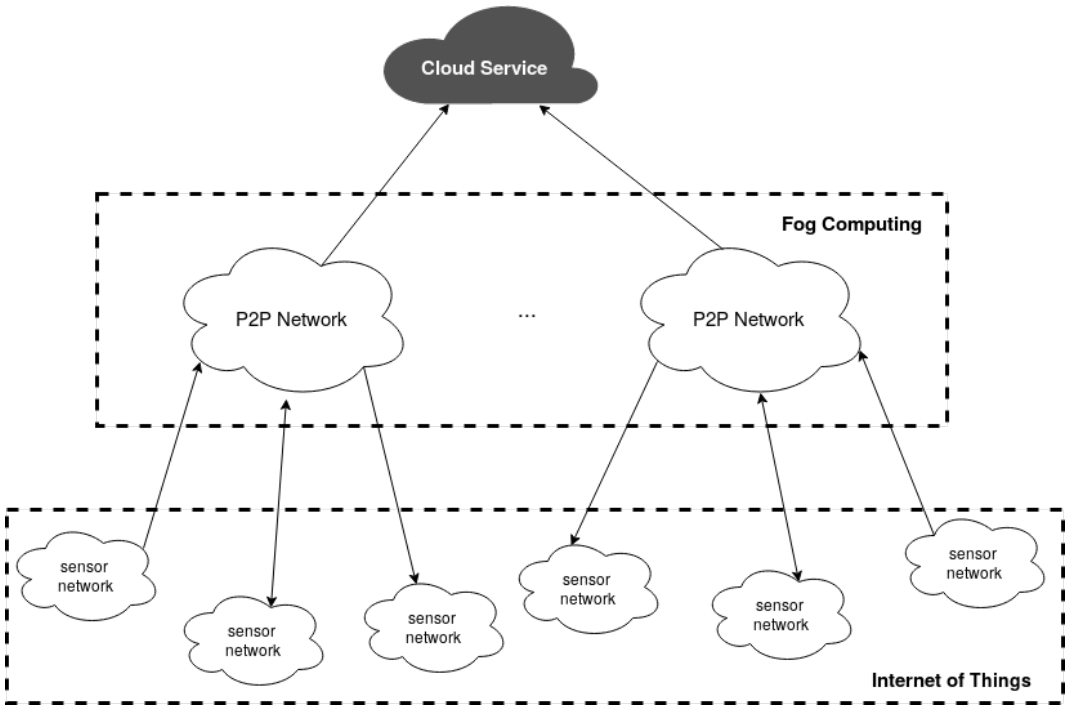

Figure 3.1: *Typical structure composed by the cloud, Fog and IoT*

On a global scale, the hierarchy existing between the Cloud, Fog and IoT, according to the taxonomies discussed about P2P, is structured and centralized at the top level. Every layer has its distinct set of functionalities and there is a abstraction gain in the Cloud-IoT continuum. A cloud solution can focus on managing the electrical grid in terms of aggregated data from the Fog. In turn, it allows the introduction of geographic awareness and the possibility to develop reactive mechanisms destined to handle faults in residential areas like power outages. Figure 3.1 depicts the global concept involved in this solution, connecting the Cloud, Fog and subsequently IoT.

3.1 The Architecture

Whilst globally a full-stack solution is seen as a promising design, we will shift focus to the Fog due to the objectives of this thesis. In this section we will explore the decisions taken towards the design of a distributed model. The architectural concept intends to achieve a stable infrastructure which is capable of being autonomous eliminating the human factor. Recalling the two classes of distributed systems discussed previously, one might add that this system is a *distributed computing system* aiming to support the *distributed pervasive systems* deployed in our surroundings. Although we presented *grid computing* and *cluster computing* main designs in regards to *computing systems* the architecture of the Fog takes advantage of both concepts. The design towards the grid is not specifically the one discussed as *grid computing*. The grid itself is regarded as geographically distributed nodes, enhancing the view of the cloud. The computational capabilities of *grid computing* are applied to Fog Computing nonetheless. There are different available resources throughout the system. Nodes are resource-devices which allow computation to take place. These, in turn, collect data from residential areas which is also made available throughout the system if requested. On the other hand, *cluster computing* fits in this system in a way that the grid itself is comprised by clusters of devices, performing cooperative operations in order to maintain stability and ensure service availability. Thus, the nodes are mainly homogeneous considering their behaviour in order to achieve such. This lead to designing an infrastructure which is *decentralized* like in P2P. The reasons are implicit, we must avoid single-points of failure, thus, decentralizing the system is the obvious approach. Additionally, by being homogeneous, nodes will be *structured* in order to maintain flexibility in handling *faults*. If a networking structure, which provides a service *fails*, then another structured group must replace their functionalities in the pervasive system. The requirements enunciated in the state of the art highly suggest that in order to support *pervasive* environments the design must take into consideration *embrace contextual changes*, *favour ad-hoc composition* and consider *sharing as a default behaviour*.

This Fog Computing system is based on existing P2P designs. Another characteristic of this P2P model is that within the Fog layer, each node is flatly distributed mainly due to the homogeneity of their functionalities. The aggregation of nodes is bound to the set of addresses supplied thus we can discuss groups of nodes as also being homogeneous *neighbourhoods* which can be simply replicated. Thus, the whole picture is regarded as a network of nodes, deployed according to a certain geographical context. These nodes will form conceptual sub-networks which will interoperate cooperatively to maintain the stability of the environment where these are deployed. Hence, we face a design which is completely *flat* and *structured* in small clusters turning the infrastructure into a decentralized one. These groups of nodes may or may not be connected between each other. There is no restriction whatsoever. The sole focus is to find a solution where nodes can autonomously form a structured environment which allows information sharing.

One should emphasize that this concept allows the distribution of nodes in an *ad hoc* fashion. More specifically, it could grant a composition based on the purpose of a real application. For instance, in the electrical grid, a group of nodes could be deployed according to the city topology, whether it is a city block, avenues or any other spatial concept. Thus, the overview of the nodes, from the Cloud, has the capability to be given a meaning according to the application purpose. For instance, one could achieve hazard detection in a grid if we consider nodes distributed over parishes. Thus, there is a segregation of groups which gives meaning to the city structure itself.

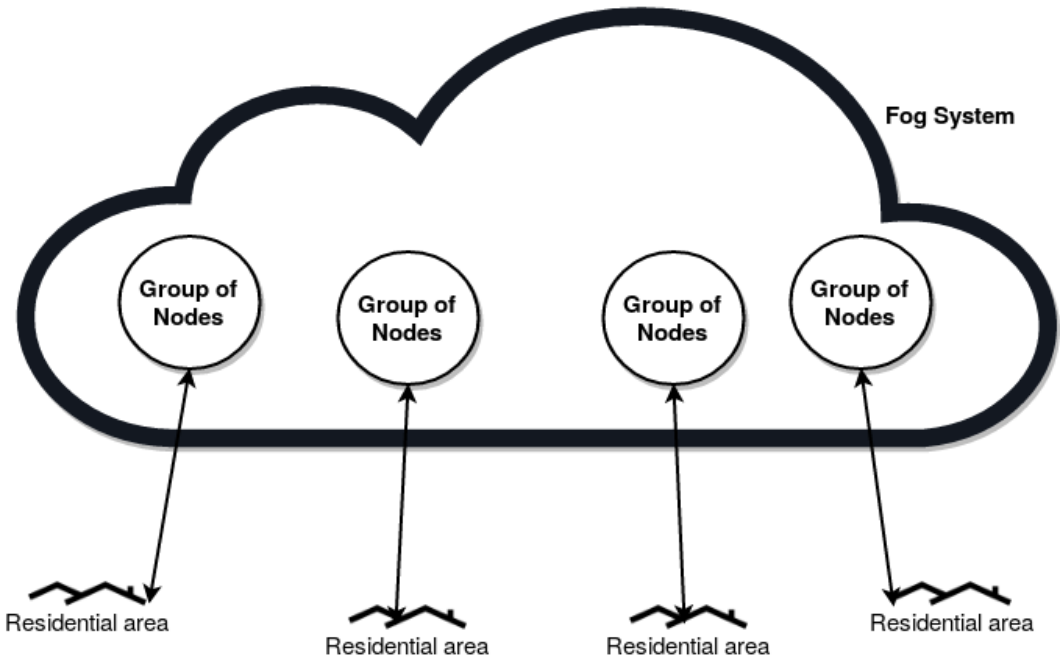

Figure 3.2: *Fog Computing system comprised by groups of nodes, i.e. neighbourhoods*

3.1.1 Node Groups

The design conceptualizes nodes as groups. By predefining the set of *neighbours* a node, we can disregard the need for a *peer discovery* mechanism to be developed. Thus, by trying to establish communication with a set of addresses, the visibility of that node is asserted directly by a response or its absence. In order to make group aggregation possible between homogeneous nodes we must consider which information must be kept within each node in order to maintain the group structure. Each node is defined by an *Unique Identifier* (UID). There are other alternatives to approach distinction between homogeneous entities, although, simplicity is always a crucial aspect to take into consideration regarding the design. There are no restrictions regarding the attribution of UIDs besides their *uniqueness*. Hence, these can be *pseudo-random* numbers generated in the manufacturing process or can be assigned *per deployment purpose*, the number itself is irrelevant. Given the UID, there is the need to share additional information between nodes. Besides the identifier of each node, there is also the need to keep track of each node's *IP address* associated to each entry. Additionally, each entry should possess two flags regarding the *aliveness* of each node and whether a certain node *is a leader*. Recalling *leader election mechanisms*, the flag, *is Leader*, useful in order to maintain its hierarchy.

Note that a consistent group table can only have one *is Leader* entry *flagged true* in each group. Figure 3.1 is an illustration of a *group table* formed by the different nodes in the group.

	Ip Address	is Active	is Leader
'1'	ABCD::1	true	false
'2'	ABCD::2	true	true
'3'	ABCD::3	true	false
'4'	ABCD::4	false	false

Table 3.1: *Group table*

Topology

As described before, groups are structured and *flat* regarding their topology. Although, in order to achieve a stable environment, the design towards nodes organization is based on a centralized, hierarchical structure. What this means is that within each group there is a node which acts as a central point towards communications. This node is addressed as *group leader* because it possess additional functionalities in relation to other nodes within the group. Additionally, the remaining nodes within a group are called *members*. The group leader is responsible for the incoming connections from nodes outside the group which belong to the *set of neighbours*. Additionally, the leader is also in charge of updates regarding the data structure containing information about the belonging nodes. This structure helps minimize connections inside the group whilst maintaining consistency regarding the global information in each node. Moreover, any newcomer node will direct its communication to a leader given it will be the only one which will reply to invitational requests. Furthermore we will approach this structure in regards to the node's internal behaviour designed as a state machine. Group leader maintains a connection with the active members. Thus, we can fathom that within the Fog Computing system, there are different topologies involved.

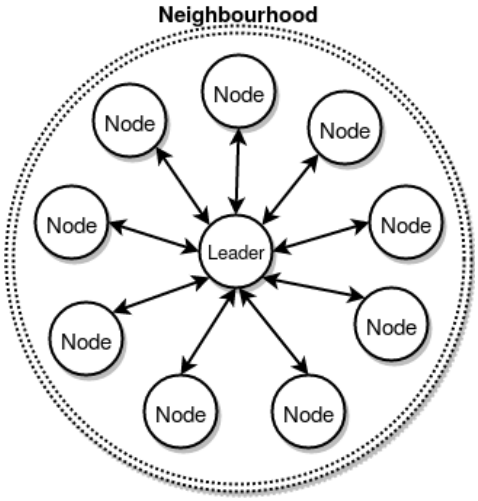

Figure 3.3: *Communication topology within a group*

3.2 Node

A node in the Fog layer is a programmed entity with different modules, concurrently operating to perform the tasks imposed by its behaviour. Moreover, each node can be seen as a

finitie state machine (FSM) where the internal modules operate to produce inputs which result in *state transistions* at any given time. The goal behind this is to perform a predetermined sequence of actions, depending on the *awareness* acquired throughout the node's *aliveness* timespan. Moreover, this model is defined by a list of states related to the status of the node. The conditions that make transitions happen rely on three different and independent factors. First a node can transitions in-between states if there is meta-stability. Additionally nodes can also change their state due to the occurrence of an output timeout. Ultimately, nodes can also perform a state transition under external stimulus. This means modules possess one or more threads independent from the state machine execution. Hence, within each node there is a multithread environment where different processes take place to accomplish the designed FSM behaviour. Although it is hard to illustrate multithreaded behaviours of application in general, we will try to justify the overall design with a thorough discussion over the FSM. In order to maximize the approximation of each explanation to the designs, take into consideration the object-oriented pardigm while describing the internal modules. An object-oriented perspective enhances the description of the capabilities of each module due to the degree of abstraction granted by objects and their properties. Before analysing in detail the composition of the node internals it is best to firstly describe the FSM behaviour, which in turn, will lead us to the modules. Besides, its easier to grasp the most abstract view first and consequently dissect its properties.

3.2.1 Finite State Machine

In order to understand the reasoning behind delivering autonomous capabilities to each node, a solipsistic perspective must be undertaken to fully grasp the interactions taking place in the Fog. Thus, if only one's mind is sure to exist, the lack of knowledge implies that the first proceeding of each note is to become aware of the availability of its *neighbours*. These always aim to find *equilibrium* within the system by ultimately creating a *group* or joining one. Hence, upon reaching one of the latter, the node can be considered to be in a stable state. Local information is the only resource available. Consequently, throughout the node *correct* operation it cannot make assumptions in regards to the past, present or future. As a solipsistic node, knowledge of anything outside one's own mind is taken as uncertain. For now take for granted there is a message passing system which handles communication, later on we will explore such module. Throughout the discussion and explanations over the FSM we will use Figures in order to help the reader visualize what is being described. Additionally, there are four messages that will be exchanged between nodes:

- Who is there (broadcast);
- Answer (unicast);
- Acknowledge (unicast);
- Info (broadcast).

Let us consider an empty universe where these nodes may suddenly emerge. Additionally, messages are guaranteed to be delivered between nodes. The occurrence of faults must also be disregarded.

Eventually, the first node gains existence, finding itself in its *initial state, i.e. Detached.*

Figure 3.4: *Initial state in the FSM*

In order to become familiar with the visual design keep in mind transitions between two different states in the FSM are represented by an unidirectional arrow. Subsequently, every arrow has a legend attached. The purpose for the latter is to represent the received messages, *trigger*, that make the transition happen and which message is sent while transitioning between states, *action*. These legends are represented as *Received message / Sent message*.

Granted the node only knows the *address* of other possible nodes, he begins to assert if any of the latter are *alive*. Hence, a primary *broadcast* message, *who is there*, is sent. Conceptually, the node started *looking for group*. Note that the *Detached* state besides being the *initial state* is also a *transition state*, this means it does not require any trigger to shift to another state. Also, each node attaches its UID to the *who is there* message. Furthermore we will explain such reason.

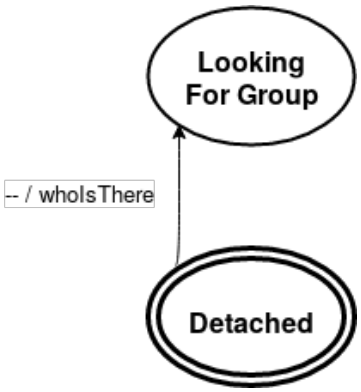

Figure 3.5: *Transitional state detached triggers the transition independently.*

After the node sent a message it has to wait for an answer. However, such reply might never be received which is the case, the node is the first in the universe. Thus, a *timeout* has to be applied in case of *omission*. Although there was no reply, the node, being solipsistic, is sure of one's existence. Consequently, the *timeout* triggers a transition to a new state, *Group leader* which indicates that the node has taken initiative towards forming a *group* and will wait for new nodes to appear. Note that the transition between *Detached* and *Looking for group* is characterized by the transitional properties of the initial state. Oppositely, the transition from *Looking for Group* to *Group Leader* is triggered by a *local* input which is the *timeout*.

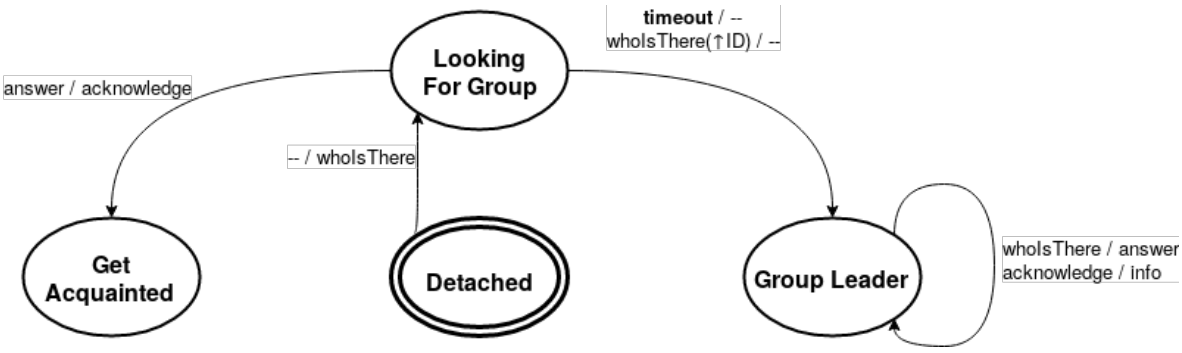

Figure 3.7: *Blocking state get acquainted triggers the transition independently.*

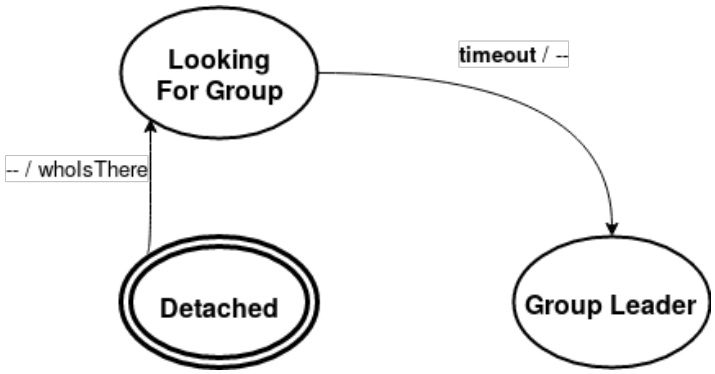

Figure 3.6: *Timeout event leads to group leader transition.*

While in *Group leader* the node will remain as such and wait for incoming messages from other nodes. Under these conditions, this state can be considered a *stable state* where the node will remain.

Taking in consideration the scenario described thus far, we will now introduce a new node in this universe. Like any other node, it starts as *Detached* and immediately transitions to *Looking for group*. At this point, the scenario changes for there are two nodes and one is going to reply to the *who is there* message sent by the second emerging node. Upon receiving the message, the first node, as a group leader is responsible for replies to newcomer nodes. Maintaining its state, it replies with an *answer* message. In turn, the second node receives the latter which activates a transition to a new state, *Get Acquainted* where it will send an *acknowledge* signalising the *Group Leader* it is ready to join the group and receive the information regarding the latter. If, after a brief period of time, no answer is received, the node in *Get acquainted* will resend the *acknowledge*. This process repeats until the number of *retries* is exceeded. If so, this node will return to *Detached* where it will restart the whole group lookup process. The preference towards opting for reset is highly supported by the constraint towards decisions based on *local information*.

Upon receiving the *acknowledge* the first node knows the newcomer is waiting for the *handshake* process to be finalized. Thus, it finally sends a message, *info*, whose payload contains its *group table* information. Alike the *who is there* message, every *info* message is also a *broadcast* to the set of neighbours the node has predefined. Furthermore we will explain why.

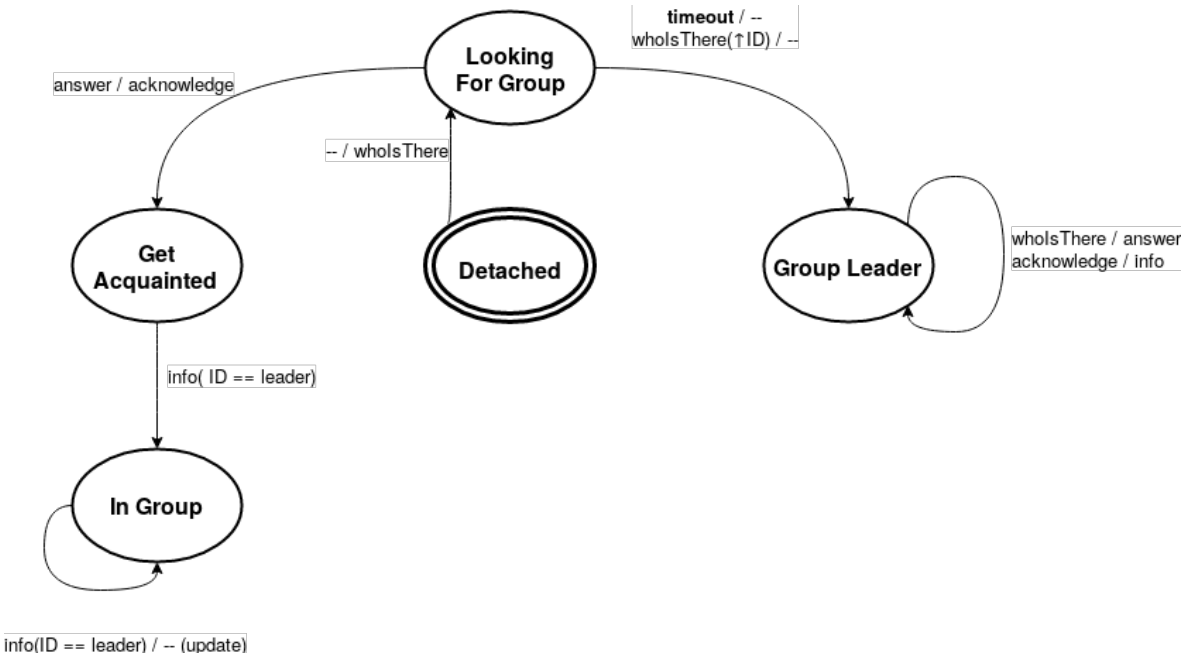

Figure 3.8: *Blocking state in group triggers the transition independently.*

The process terminates when the newcomer node receives the latter and makes a transition to *In Group* which is the second *stable state* in the FSM. At this point the node will remain in such state and will listen to future *updates*, regarding its group elements status, from the group leader. Moreover, any newcomer node will follow the second behaviour described previously.

The finite state machine backbone is presented in Figure **??**. The final version has more transitions due to the increased reliability which implied an augment of redundancy. Before we move onward to the description of the modules that make this FSM work, let us summarize what as been fathomed.

Each node upon gaining existence immediately searches for other active nodes. This will have two possible outcomes which will terminate as the node belonging to the group as a *member* or as a *leader*. Additionally, there is a *finite* set of messages, exchanged between nodes, that make node aggregation possible. The messages enunciated thus far are divided in two types, *unicast* and *broadcast*. The former mechanism is unique message sent towards one destination node while the latter is a message sent to a pre-defined set of addresses, i.e. *set of neighbours*. Overall, there is a finite set of states that allow a node to transitively reach a stable state where it can operate *correctly*.

3.2.2 Communication

Communication at the *Transport layer* is based on User Datagram Protocol(UDP) protocol. Due to its simplicity, lack of retransmission delays and ultimately being *stateless*. All of these are imperative characteristics in real-time systems. Additionally, UDP is widely used on host-to-host communication, thus, the communication mechanism uses UDP as the transport protocol.

Each node requires atleast two UDP *sockets*, one which supplies incoming connections and another which deals with outgoing ones. The design of the transmitted resources must also take

into consideration the maximum size of an *UDP Datagram* and the *Maximum transmission unit* (MTU).

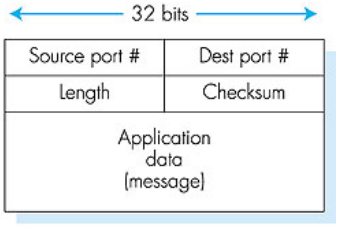

Figure 3.9: *UDP segment structure*

The source of figure 3.9 is provided below[3].

Given the protocol which enables the communication to take place in order to share resources, we will now approach the design of the functionalities made available to client processes. Generally speaking, messages must be *sent* and *received*. Hence, like the postal office system, one can approach the solution by providing a way to:

- *Put* a message;
- *Poll* received messages;

These operations satisfy the conditions where the receiving process can be differentiated from the sending one. In fact, the internals of *communication* between nodes are abstracted by this approach. This leads us to how this module was designed in order to support the previous operations. We will discuss its internals regarding the objects developed to support such structure and how a multithread technique can turn this module self-sustainable.

Design

In order to ease the *naming* and referencing, we will address this module as *Communication Manager* (CM). This entity is responsible for the abstraction over the communication process. Thus, a set o predefined requisites must be established. Additionally, the API must be bound to a certain interface providing a finite set of methods that can satisfy the client process needs towards communication in both directions, i.e. *incoming and outgoing*. Hence, we achieve the following list as a set of requirements:

- Send a message to multiple addresses, i.e. *broadcast*;
- Send a message to an unique address, i.e. *unicast*;
- Be able to store messages and maintain the order they are sent or received.
- Upon request, from the API, retrieve a message, by its order.
- Abstraction over connections.

Furthermore the API will suffice these needs by providing the following signatures:

[3] `http://netlab.ulusofona.pt/rc/book/3-transport/3_03/03-07.jpg`

- *void sendBroadcast(Message m);*
  Broadcast method that takes the input parameter and sends, according to the set of neighbours, the message to the destination. The message must have a valid source address and payload. Since there is a neighbour list, the destination headers are guaranteed to be set accordingly.

- *void sendUnicast(Message m);*
  Unicast method also requires a message an input parameter. Both header and payload must contain valid information in order to guarantee its delivery. The method receives the object and stores it in the appropriate queue.

- *Message retrieveNextMessage();*
  This method is a *blocking* method since the invoking thread on this method is constrained by the monitor protecting the inbox queue. Additionally, another method is provided (isEmpty) in order to avoid *blocking* if there is no message available.

- *boolean isEmpty();*
  The standard method that indicates through a boolean return value if the associated inbox is empty or not. This helps threads blocking on the communication manager if expecting a message. Instead they can invoke isEmpty in order to skip the *blocking* imposed by the monitor on the queue.

Due to the fact that we are discussing an autonomous entity, the notion of *time* regarding each message is different from the client perspective. In order to circumvent this problem one is able to guarantee the order of messages by using *Queues* as an Abstract Data Type. Queues are structures where elements are inserted and removed in a FIFO fashion, thus, maintaining the overall order of the elements while storing them. The CM contains two independent threads, which we will name as *Receiver* and *Sender*. The reason behind two threads is the fact that there are also two queues, one for the incoming messages and another for the messages to be sent. Additionally, the difference between a *send* operation and a *receive* is sufficiently distinct to justify such separation.

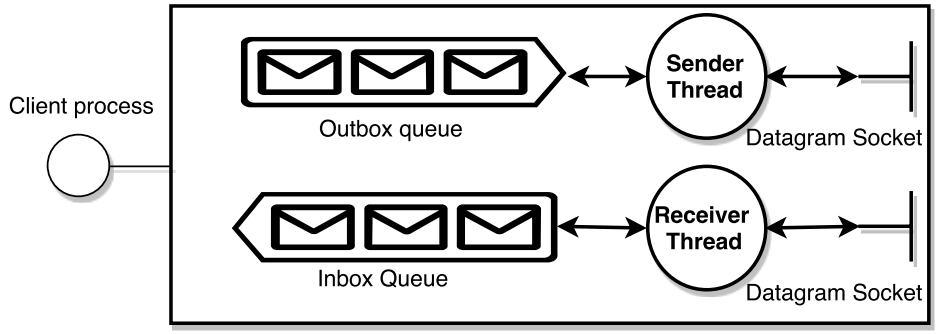

Figure 3.10: *Communication module*

Receiver thread

This thread is responsible for listening to a certain *datagram socket*, i.e. port. It handles the validation of the data and keeps an open connection to outside nodes in order to continuously receive messages. Upon receiving a message, the receiver does the *unmarshalling* it and stores

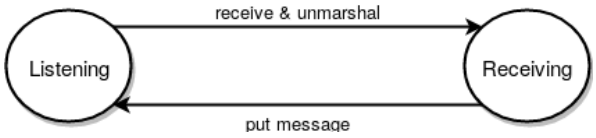

Figure 3.11: *Receiver thread behaviour*

---

it in the respective queue. The concurrency factor happens when invoking clients attempt to retrieve messages while the receiver is operating over the queue. Hence, this *shared region* has to be protected by a *monitor* or a *semaphore* mechanism. Moreover, its behaviour can be described as a state machine with two states. The *initial state*, *Listening*, is also a *blocking state* where the thread is listening in a given port for the arrival of packets. Upon receiving one, it proceeds to the de-serialization of the latter and marshals it into a message object. This procedure originates a transition to the second state in the FSM, *Receiving*. This state is a *transition state* which triggers the thread to store the unmarshalled object in the *inbox* queue ultimately making it available for retrieval. This operations originates a transition back to the *Listening* state where the node will repeat the procedure.

Additionally, in order to understand the procedure itself it is presented bellow the algorithm associated to the receiver.

```
do {
    DatagramPacket p = socket.receive();
    Message m = unmarshal(p);
    inbox.put(m);
} while(true);
```

Sender thread

On the other hand, the sender thread is synchronously retrieving messages from the queue and *marshalls* them in order to be placed in the *UDP Datagram*. Similarly to the inbox queue, the outbox has to be protected due to the concurrent environment from the client, which inserts messages, and the sender, which polls them in order to be sent. This thread's behaviour is also defined by two states. Similarly to the receiver, its *initial state* is also a *blocking state*. The thread remains in this state in two situations: there is no message in the queue to be sent; the monitor has concurrently synchronised the thread so it has to wait for the shared region to become available. Upon concurrently *pulling* a message, the thread transitions to another state, *Sending*. Alike *Receiving* this state is also a *transitional state* meaning after acquiring the message it begins to marshal it and sends it through a open socket towards the destination. This operation makes the thread transition to the *initial state* where it will repeat its behaviour.

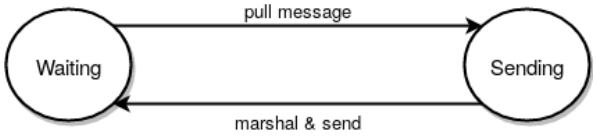

Figure 3.12: *Sender thread behaviour*

```
do {
    Message m = outbox.poll();
    DatagramPacket p = marshal(m);
    socket.send(p);
} while(true);
```

Message structure

A well defined structure is crucial to support the necessity towards sharing resources. Since the communication module was developed as an object we also designed messages to be handled in such composition by providing them an object oriented structure. Thus, messages are objects sent over the network which hold information shared between distributed nodes. Their structure was designed as illustrated in Figure 3.13.

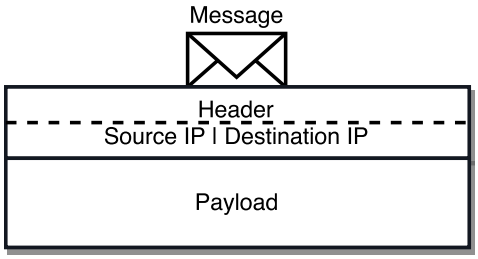

Figure 3.13: *Conceptual message object*

3.3 Fault Tolerance

Achieving a fault tolerant system while considering the the node properties thus far it is required to assert which scenarios may cause disruptions in their overall behaviour. Before, we considered packets to be always received. If one refrains from such characteristic, then message-passing is prone to the following failures:

- Delays;
- Losses;
- Duplicates.

Another aspect is node availability. A message *loss* cannot be distinguished from a *crash*. Thus, it is required to enhance the system with an *Heartbeating* mechanism which grants *awareness* between nodes. A crash of a node will originate two outcomes: Inconsistency in the *group table*; Disruption in the hierarchy structure in case of the node being a *leader*. This results in the enhancement of a node's behaviour with a *leader election algorithm* which allows, within a finite timespan, group self-organization by electing a new leader. Additionally, it is only possible to distinguish a message *loss* from a node *crash* if there is a guarantee towards the end-to-end (E2E) availability. In turn, *delays* and *duplicates* can be handled through the induction of *redundancy* within the FSM.

Ultimately one aims to achieve *fault tolerance* regarding those scenarios. By being successful, it is possible to grant high-availability related to depending infrastructures at the IoT level. Additionally, nodes can replace other node's functionality by detecting and handling *crashes*.

3.3.1 Heartbeat System

From the node perspective, the *Heartbeat* mechanism is an independent module. The operational procedure relies on connecting to other heartbeating mechanisms in order to receive and send *heartbeats* throughout the network. Moreover, there is a specific purpose behind the communication between heartbeats of different nodes. As enunciated before, *heartbeating* can help maintain the consistency regarding *group tables* and the group hierarchy. Whenever a new entry is added to the group table, at the node level, the associated address on that entry must be supplied from the node to the heartbeat module in order to start sending *heartbeats* to the respective node. The idea behind this concept requires two threads to run simultaneously. Since every node is homogeneous, one takes advantage of the same property regarding the HB module. One thread, *Heartbeat Generator*, is responsible for the *broadcast* of periodic *heartbeats* to a set of addresses. The second, *Heartbeat Listener*, functionality is to receive heartbeats from other nodes.

Finally, we achieve *awareness* through the use of a *shared region* which is a *bounded counter* formed by *(key, value)* pairs of entries, the *keys* are a set of addresses gradually supplied by the owner and *values* the corresponding *heartbeat count*. Every periodic *heartbeat broadcast* decrements an unit from each *(key, pair)* of entries. Additionally, the *Listener* waits for heartbeats and increments the *value* from the associated *address*. The use of a *bounded counter* structure enables the triggering of events such as sending a *localhost* message to the owner, noticing the *offline* status of another node. The procedure that handles such event is left to the node itself.

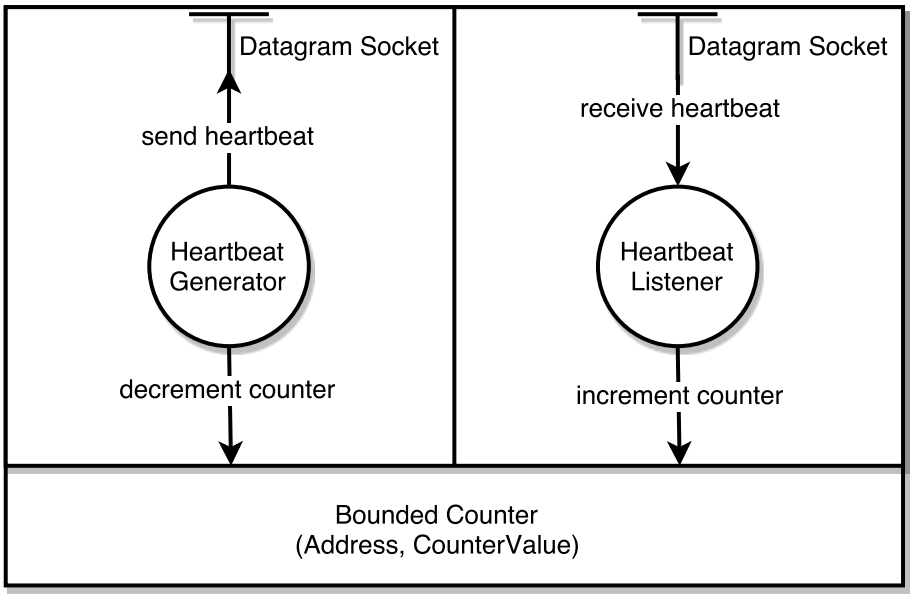

Figure 3.14: *Conceptual Heartbeat module with two independent threads*

Let us take a look over the algorithms associated to each thread in the Heartbeat module. Firstly we will discuss the *heartbeat generator* logic. After, it is presented the algorithm, related to the *heartbeat listener*.

Heartbeat generator algorithm

```
do {
    for (Entry e: boundedCounter.entries()) {
        sendHeartbeatMessage(e.address);
        boundedCounter.decrement(e.address);

        if(boundedCounter.get(e.address) <= 0) {
            notifyNode(e.address);
            boundedCounter.remove(e.address);
        }
    }
    sleep(T);
} while(true);
```

Where T is an arbitrary, constant, amount of time.

Disregarding the communication process, the thread starts by iterating over the *set* of entries in the *bounded counter*. Recall that each entry is composed by a *(key, value) = (address, counter value);* Hence, on each iteration the thread sends an *heartbeat* message to the address associated to the key. The thread then decrements a unit from the *counter* and proceeds to the next instruction. Before reaching the next entry, the thread verifies if the counter value, regarding that entry, is *below or equal* to zero. If so, it means that there was a succession of heartbeats with no response which led to the *threshold* limit. Thus, the generator detects a *crash* and notifies the node of such. The thread continues to iterate over the entries until none remains. The process finishes with the listener *waiting* for a period of time before repeating the process.

Heartbeat listener algorithm

```
do {
    Message m = receiveNextHeartbeat();
    if( boundedCounter.contains(m.getAddress) )
        boundedCounter.increment(m.getAddress);
    else
        boundedCounter.add(V);
} while(true);
```

Where V is an arbitrary, constant, initial counter value. Note this value determines the amount of sequential *omissions* before triggering the *crash detection.*

As its observable, the algorithm behind the *listener* and the *generator* are simple. Given that the former thread *periodically* sends *heartbeats* then, the listener will receive them *periodically.* The first regards to the retrieval of an *heartbeat* message. Upon receiving such, the listener ensures that the associated address is tracked by the counter. If so, the listener has to increment a unit on the counter value of that same address. Else, the address is stored in order to start tracking the respective node. The reason behind this distinction lies in the fact that nodes do not become aware of each other simultaneously. Thus, it is likely that an heartbeat can be received before the node signalises the heartbeat to track such address.

3.3.2 Group Leader Election

The election algorithm is a *simplified version* of the *bully algorithm* given that we consider each node to be solipcistic. Additionally, each node favours the node with the *lowest UID*, following the *crashed leader*, to be the *best candidate*. Similarly to the *bully* approach, a node upon becoming *aware*, through the *heartbeat*, of a *leader crash* it sends a message, *start election*. Oppositely, this message is sent to the *set of neighbours* instead of following the *highest ID* rule in the *bully algorithm*. Every node, upon receiving the election message will remove the *crashed leader* from their tables and elect the best candidate with the *lowest ID*. The process terminates with the leader sending a *broadcast* message with the updated *group table* indicating the process is finished. If the node that detect the *leader crash* is also the best candidate the election process is the same. Additionally, if the leader considers to leave the group, it will send a broadcast to the set of neighbours in order to start an election.

Ultimately, this algorithm can be seen from the FSM perspective as a set of transitions. Moreover, due to the fact that *start election* is a broadcast, a node can receive it in any state. We will cover the latter after the figure 3.15 which will help us understand the implied *redundancy*. One also opted to omit the other messages and associated transitions, achieving simplicity throughout the explanation.

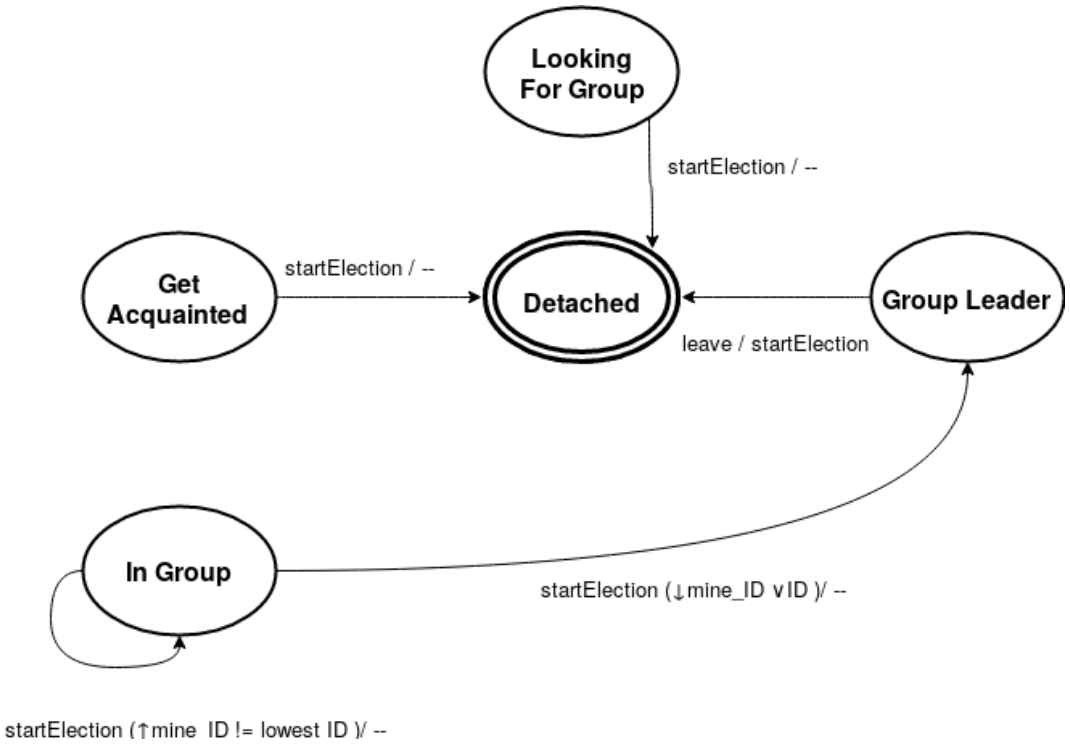

Figure 3.15: *State transition upon receiving start election*

The first thing to notice is the fact that one can disregard the *Detached* state because its a *transitional* state where the node describes a strict behaviour. Additionally, a node will never receive a *broadcast* message while being on *group leader* state because it represents the *crashed node*.

Once on *looking for group* the node can receive a *start election* message. The approach

taken in this situation dictates that the node will rewind to the previous state. Since an election is being held, the node would get no response from a leader regarding the *who is there* (see figure 3.5 on page 35). Thus, it will repeat the process in order to allow the new leader to respond to this message ultimatey normalizing the behaviour. Furthermore, a node in *get acquainted* would be under the same situation originating a transition to *detached* which will restart the looking for group process.

The last state remaining is *in group*, whilst being here, a node belongs to an active group. Thus, it will join the election process asserting the best candidate. From the latter, two possible outcomes can happen:

- The node has the *lowest UID* making it the best candidate;
- The node does not have the *lowest UID* resulting in the loop over *in group*.

Upon receiving a notification from the heartbeat, the node must assert the role of the *crashed entity* within the group. Thus, different steps are taken regarding the election process. Furthermore, the behaviour taken by each node can be described, as pseudo-code, through the following algorithm:

The process starts with a broadcast of a message *start election* to the set of neighbours. Given that communication has already been discussed we can disregard the properties of sending this specific message resulting in the method *broadcastStartElection()*. Following, the node removes the leader from its *group table* and proceeds to find the best candidate to be elected, the latter being a node with the *lowest uid*. The group table, as a data structure, provides methods to satisfy these operations. The node then turns the *is leader* flag regarding the assessed candidate.

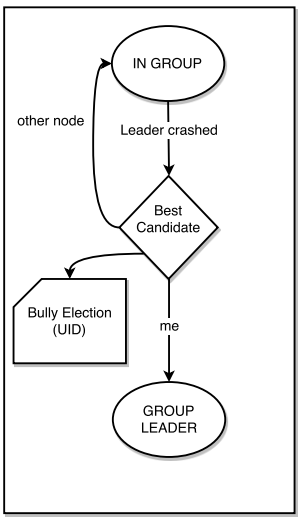

Figure 3.16: *Decision process imposed on leader election mechanism.*

3.3.3 Additional Redundancy in the Finite State Machine

In a real scenario, nodes can start at any given period of times. Hence we should consider *simultaneous* start of nodes in order to increase the reliability of the FSM. Additionally, we only covered the transitions where the node would receive the expected input, thus far. In order to enhance the reliability of the state machine we will introduce redundancy related to messages that might be received outside the expected behaviours. First of all, *broadcast* messages to the set of neighbours must be considered in any state. Those are *who is there* and *info* messages. Moreover, we will have to address *answer/acknowledge unicast* messages due to messages *omissions*. Let us start by introducing the *who is there* message in the FSM states. The decision process, when it comes to ID comparison, always favours the node with the *lowest UID*.

The *Detached* state is disregarded because it is a transitional state, thus, the node does not check the received messages. Furthermore, the first state where one should analyse redundancy is *looking for group*. Recalling *simultaneous starts*, two or more nodes will send *who is there* messages which will be received by one another whilst being in *looking for group*. Although these are not the expected *answer* from a *group leader* it indicates, from each node's perspective, that other entities are also trying to find a group. Consequently, there is no group, which implies that a new scenario emerges: Simultaneous start of a group of nodes, in absence of a *stable group leader*, requires that the nodes become aware leading them to form a group amongst them.

Here the decision is simple. Assuming every node receives *who is there* from all the other nodes, it is possible to iterate over each received message aiming to assert if one owns the *lowest UID*. If so, the *lowest UID* node will transition to *Group Leader* where it will form the group.

Oppositely, all the remaining nodes with an higher UIDs will remain in *looking for group*, waiting for the *answer* message from the former. If the node *does not* crash while transitioning to group leader, he will answer to the *who is there* messages received in *looking for group*, inviting the remaining nodes. In case of a *crash*, while transitioning to *Group leader*, a **timeout** will occur in the nodes waiting for the *answer* in *looking for group*. This happens because nodes assume the best case where an *answer* message is lost. In turn, these will transition to *Get Acquainted*, here they will send an *acknowledge* in order to signal the leader they are ready to join the group. Since the leader is *crashed* they will retry after a certain time but eventually return to *Detached* due to the limit imposed on the retry process. Maintaining the focus towards this state, let us now analyse what should be done in in regards to *info* messages. The meaning behind this message is that a *group leader* updated its *neighbours* by completing an handshake, although, there is no direct implication over the node which is in *looking for group*. Consequently the node will ignore such message and peek over the next in the queue.

In sum, there are three new "triggering" situations in the FSM. One implies a return to the *Detached* state, in case of the only value in the information received is in regards to other nodes, with *lower UIDs*, also looking for group.

Another transition, between *Looking for group* and *Group leader* is oppositely triggered when there are only nodes with *higher UIDs* looking for group. Ultimately, the reception of an *info* message, no matter the *UID*, is discarded due to the lack of *interest* towards it.

Moreover, a node in *Group Leader* can receive *info* messages. Here, there are two divergent situations. If a node receives an info message, from a *neighbour* in its set, this means that in

the meantime between its start up and its own group formation, there was a delay or losses in the transmission. Additionally, due to the fact each node cannot make any assumptions towards time, one must deal solemnly with *local information*. Thus, if such a message is received the *group leader* must assess if the associated UID of the other node is *lower* or *higher*. By figuring such, one is able to determine the conditions which led to this scenario:

If a node (B) receives a message *info* from another node (A), then, this means A just finished an *handshake* process with a third node(C) which joined A's group.

- If A has a *lower* UID: Then B should disband the group because there is a better *candidate* in another group. Thus, B transitions to *Detached* as a consequence of becoming aware of such information. This can happen due to the *delay* or *loss* of any of the messages that originate a transition to *Get Acquainted* or *Detached* while B was in *looking for group*. By restarting the FSM to *Detached* B will repeat the group lookup process and if no messages are delayed or lost it will join A's group.
- If B has the *lower* UID: Since B is the node which received the *info*, then he must ignore it. B is in fact the best candidate to be a group leader because it possesses the *lowest identifier*. Any *info broadcast* made by B in the future will result in A become aware of the previous enumerated situation, which will make A reset to *Detached* consequently joining B's group.

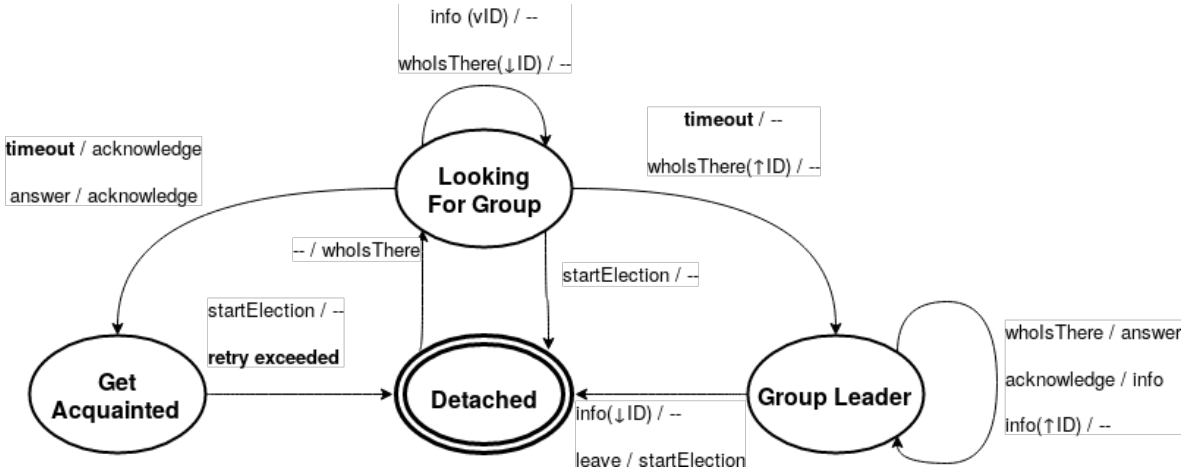

Figure 3.17: *Detached, Looking for Group and Group Leader states with expected trigger messages and additional redundancy.*

We have covered *Looking for group* and *Group leader*, thus, there are two more states remaining in the FSM where we must analyse which measures must be taken in regards to the possible set of messages that might be unexpected but nonetheless received.

A node in *Get Acquainted* can receive both *who is there* and *info* messages. The former only indicates the node that there are other nodes searching for a group. Thus, every message of this type can be ignored. Moreover, *info* messages are expected while a node is in this state. Although, the only covered instance was when the *info* is received from the *group leader* who sent the *answer* message. This means that when a node receives a message whose UID is different from the one belonging to the expected group leader different information emerges, resulting in the following scenarios:

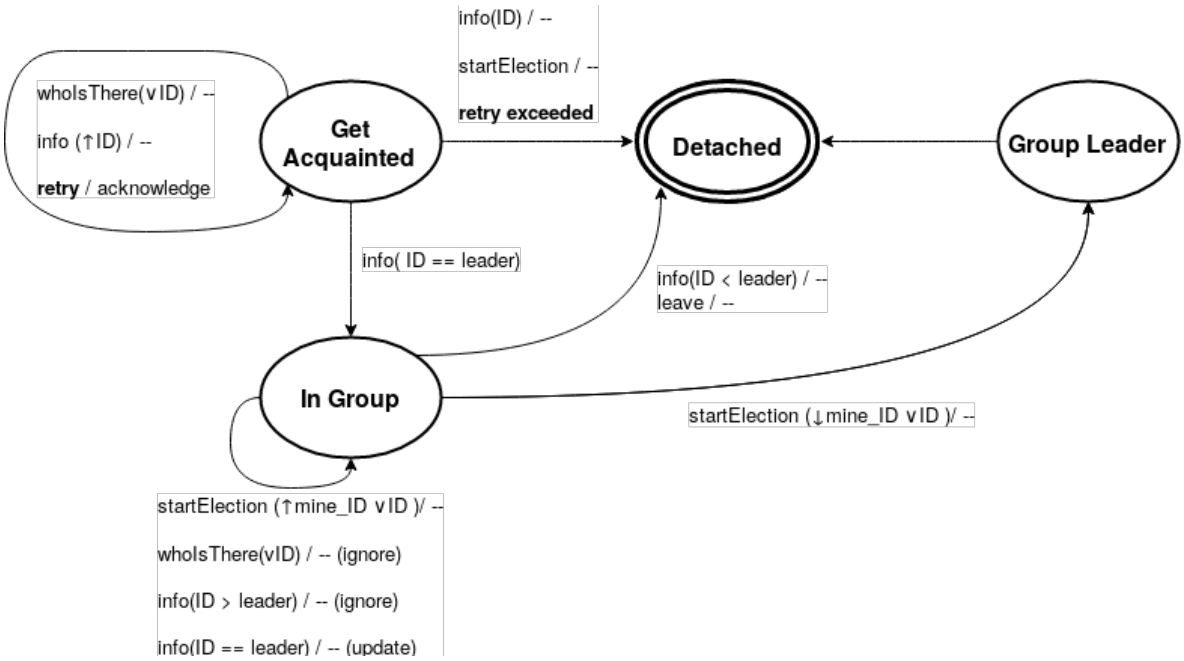

Figure 3.18: *Detached, Looking for Group, Group Leader, Get Acquainted and In Group states with extended triggering messages.*

- Identifier in *info* is *higher* than the corresponding group leader: This means there is another group formed, but the group leader is a worse candidate than the one who replied to the *who is there* message. Thus, this message should be ignored. Note that every node in *group leader* upon receiving an *info* message with a smaller UID immediately returns to *Detached*.

- Identifier is *lower* than the the one expected in *info*: In this case, there is a better candidate which the node was not aware of. Consequently, the node should formalize the process with that candidate instead. The outcome from such *awareness* leads the node to go back to the *Detached* state in order to do so.

Moreover, granted the *correct* behaviour in a node while in *Get acquainted*, it will transition to *In Group*. A node whose state is *In Group* expects *info* messages, as a normal behaviour, from its group leader. Although, we previously discussed that there are two additional situations where info can diverge from the standard behaviour. When a node receives an *info* message with a *higher* UID than the current leader, it will ignore it. Another message which is ignored is *who is there* messages. It is the group leader's responsibility to answer such messages. The remaining case is when the *UID* is *lower* than the current group leader. Here, the node will have opt to leave the group by returning to *Detached* where it will join the best candidate group.

Figure 3.19 represents the final design of the *finite state machine*. We have approached all the outcomes in regards to the actions that should be done upon receiving a certain message from another node. We took advantage of the information they represent turning *unexpected messages* into an asset capable of increasing the reliability of each node's behaviour. By extending the decision making to all the messages that can be received one can reach a *stable environment* in abnormal conditions. At this stage is possible to conceptualize each node as

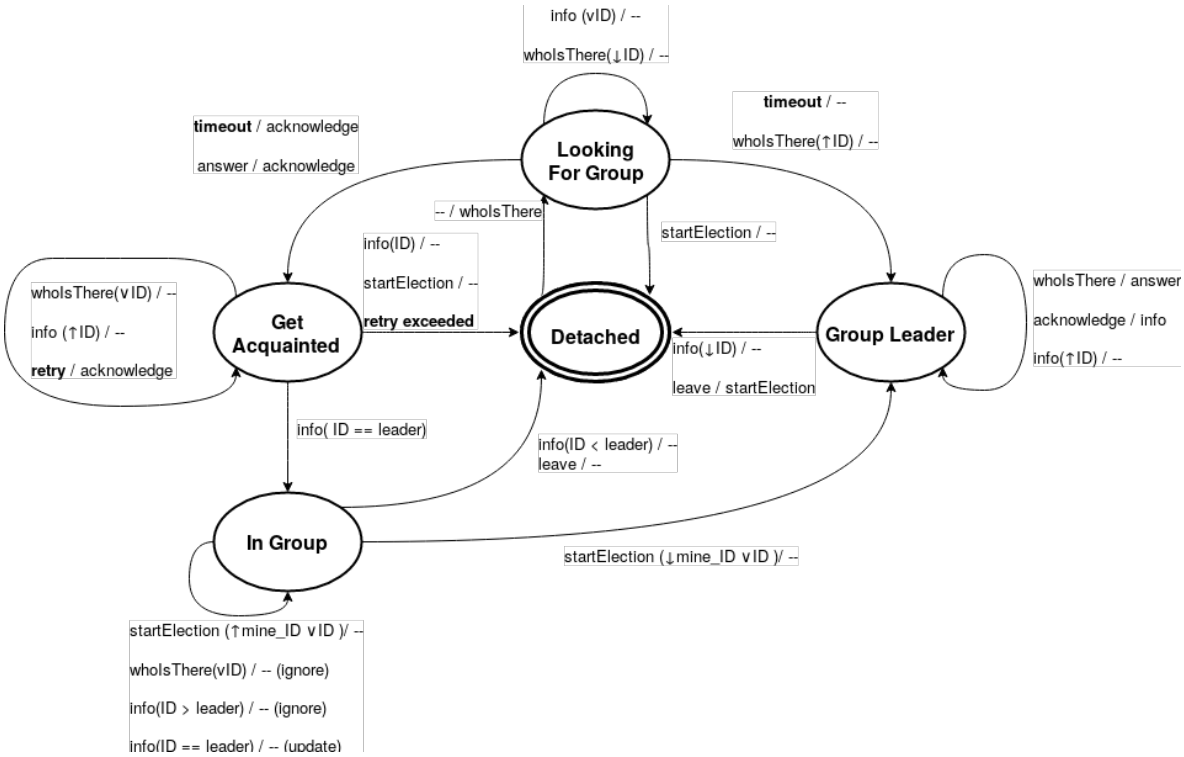

Figure 3.19: *FSM overall functionality enhanced with intelligence by the induction of redundancy.*

an entity which will autonomously attach itself to other nodes. Additionally, if such action is not possible, the node will form a group where he will accept future invitation requests from other emerging nodes. The discussed mechanisms grant additional reliability in regards to the continuous effort towards maintaining a stable organization between nodes. Achieving this design allows us to further discuss the application between these nodes and the pervasive environment in IoT. *Disclaimer:* After the development and deployment of this system, one grasped that there is a different approach in regards to the redundancy triggers in *looking for group*. Thus, we will approach the subject as further developments (see section 5.2 on page 78).

3.4 Fog-IoT Continuum

The bridge between the Fog and the IoT environments is built taking in consideration some characteristics of the latter. First and foremost, IoT devices typically organize themselves in sensor networks (see section 2.4 on page 24). These usually communicate through wireless channels, which in turn, imply a reduced visibility towards their surroundings. What is meant by this is that a sensor, in order to connect to a node in the Fog, has to be within the node's range in order to be able to communicate. Nonetheless, we aim to achieve a cooperative protocol between IoT devices and Fog nodes. In order to understand the behaviour implied in the resource sharing procedure, one has to acknowledge that a device has to search for existing *ad hoc networks* within its surroundings. Additionally, the device connects to a network which it relates to, defined by its design. Throughout the development of the procedure one must had to consider that these devices can be very limited in regards to their computational capabilities.

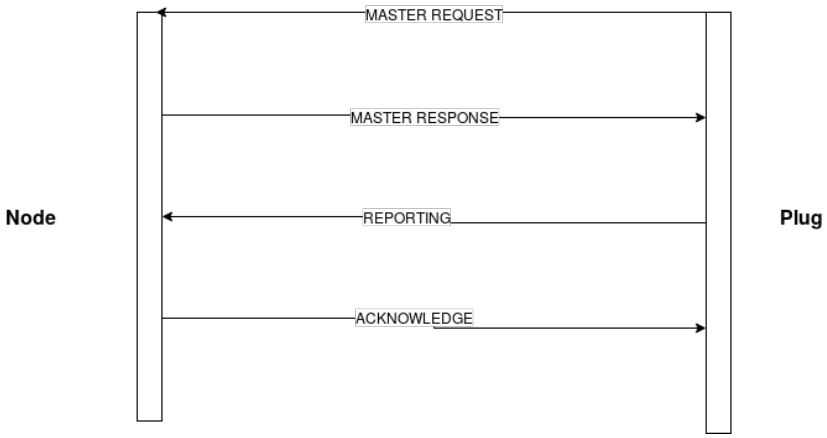

Figure 3.20: *Interoperability between a plug and a node*

This implies that the protocol has to be simple and resource-efficient. The procedure must be done in a finite number of steps enabling the IoT device to share its data with a node, as soon as possible.

3.4.1 Resource Sharing Procedure

The protocol establishes a set of messages that support the interaction between *nodes* and *devices*. The considerations about *local information* are also present throughout the communication, thus, we have acquired four steps that enable resource-sharing to take place (see figure 3.20). The initiative towards establishing communication must come from each device. These, will search for nodes and upon getting attached to one, will start exchanging the messages defined by the protocol. First, the device sends a *master request* to its node, the device then waits for an answer within a limited time period, if such is not received, he will retry the procedure described thus far.

In turn. the node will receive the message and register the device to a data structure that maintains attached devices information. Additionally, the node answer with a *master response* towards the device, indicating it is able to receive that device's resources.

Moreover, the device receives the previous message and becomes aware that it now is attached to a node and proceeds with the forwarding of its collected data to its *master*. Now, assess that depending on the periodicity factor within the need to receive real-time values, one can adjust the interval of time between reports of the device to the node, hence, *report* message. The device also checks of incoming *acknowledge* from its master indicating the node is receiving the data.

A deeper analysis in this behaviour suggests that there is an unreliable behaviour in the resource sharing transmission. Thus, the design of the communication process was enhanced upon considering faults in nodes, thus the *acknowledge*. If a node stops responding to the *report* messages, then after a certain amount of omissions the device will reset its behaviour, restarting the search for a new node in the vicinity. An important factor about this procedure is the lack of *load balancing* of sensors between nodes. Partially due to their network range constraint but also because the main focus was towards achieving a stable system at the Fog.

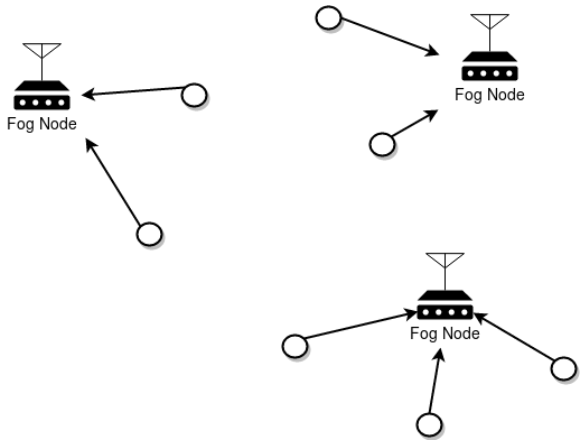

Figure 3.21: *Conceptual node-device scenario*

3.4.2 Device Response Under Node Crash

So far, we have acquired the steps required to develop a fault tolerant environment between nodes. Although, how should a device proceed if the node it is attached to *crashes* is also an important concern towards the design of a behaviour able to prevail under stress situations. In sum, this section can be regarded as a feature of the behaviour. Given the *ad hoc* connection between these two entities and the previous protocol a device describes the same behaviour as an *heartbeat* when an answer in regards to their *master* is not received. Thus, upon reaching a certain limit of *retries*, the devices begins to send a new *master request* to its surroundings aiming to guarantee a new connection.

Regarding figure 3.21, it illustrates a group of nodes and attached sensors. This scenario is a typical stable scenario where nodes are aggregating data from their devices. This in turn is available to the cloud for analysis and displaying to end-users. Even so, the scenario changes as soon as the devices attached to the *crashed node* reach their retry limit. Ultimately, each device has to search for a new network(figure 3.22). Keep in mind these nodes are "broadcast" by nodes intended to provide an environment for sensors to connect to. The protocol is then reset and the steps described before take place, stabilising the scenario. There is no effort to maintain a pervasive database which guarantees the redudancy of lost data in a crashed node. Replication regarding this resource is not a focus of this thesis, although it is mentioned as a further improvement (see section 5.2 on page 78).

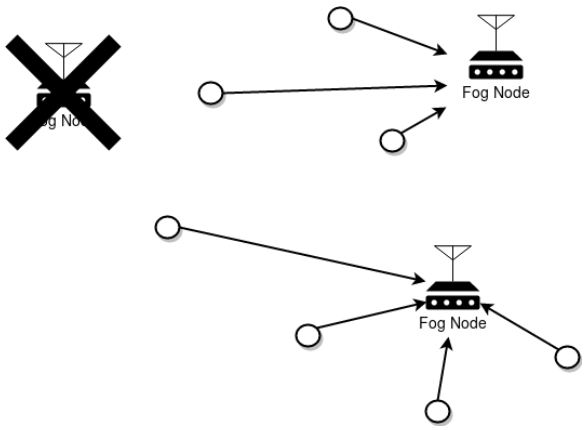

Figure 3.22: *Crash and subsequent self-alignment of the network.*

Chapter 4

Proof of concept

In order to realize the ideas proposed we will present in the following section the methodologies and test cases that we regard as being adequate to simulate a realistic environment while assessing the validation of the inherent mechanisms in each node. The demonstration principle behind the various proofs is made by deploying programmed software in order to apply the behaviour to each node.

This prototype has two different interacting entities. Nodes and sensors. In turn, these have different roles within each simulation. Whilst sensors are subject to a *proof of concept* in regards to aggregation, nodes need different simulation environments to validate the different mechanisms they possess.

4.1 Methodology

The test cases were performed using five prototype nodes, *smart energy gateways*, and twenty-one prototype sensors, *smart plugs*. The design behind the proof of concept lies on the fact that the Fog must handle itself properly under faulty situations. Thus, we recreated three test case scenarios where the assessment over the node behaviour, as a solipsistic entity, interacting with other entities outside one's mind, is taken into shape as crashes. And real life behaviours may occur. The idea behind these scenarios is simple: *if a node performs under stress conditions, it will work smoothly under normal ones.*

Moreover we present two figures associated to the equipment previously described. First it is possible to see in figure 4.1 a *smart energy gateway prototype* where the software belonging to the Fog was deployed. Additionally, on figure 4.2 one can observe two plugs with the standard electrical component but also a power button that can turn on and off the plug.

The first example is a *simultaneous start*. In a realistic scenario, nodes can emerge at any time. Nonetheless, they must be capable of cooperating in order to achieve a *stable* organization. Moreover, we also consider the situation when a group is *stable* but a sudden crash happens on a group leader and another random node. This will help validate both the heartbeat mechanism and the leader election algorithm. If a group of nodes is able to detect both crashes and recover from them, eventually stabilizing the group, then we can assume both approaches to be valid. Furthermore, one last test case is when sensors attach to nodes. Thus, we randomly placed sensors which autonomously connect to nodes in their range. This can be supported through visual confirmation of each plug data, present in each associated node.

Figure 4.1: *A gateway prototype used in the proof of concept, provided by WithUs Lda.*

Figure 4.2: *Two smart plug prototypes used in the proof of concept, also provided by WithUs Lda.*

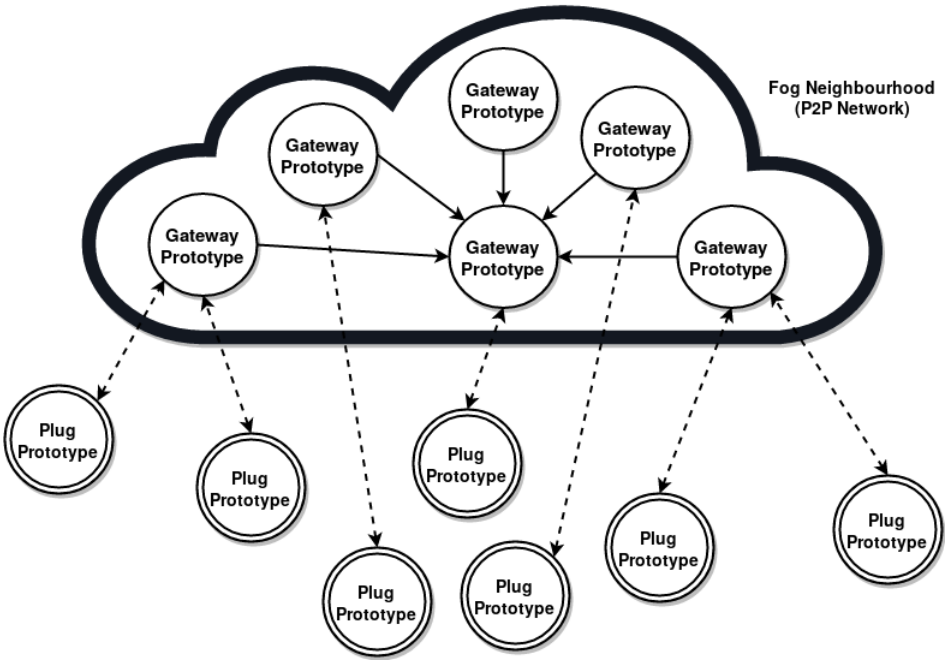

Figure 4.3: *Proof of concept topology, in comparison to figure 3.1.*

The procedure associated to the *simultaneous start* tries to start nodes at close instants, allowing an approximation towards *simultaneity*. Besides this consideration, every procedure and behaviour is autonomous. The *crash* test case is based on the procedure where there is an induced *crash* on a machine. This means while operating correctly, we shutdown the device causing the same behaviour when a typical crash occurs. This procedure will help the detection over other entities, granting awareness on each machine. Also, *leader election* mechanisms are a consequence of such *awareness* with a goal of achieving *stability*.

This way, it is possible to relate figure 4.3 to figure 3.1[1] in a sense that P2P networks are formed by groups of nodes, i.e. *smart energy gateways*. Bellow the Fog layer we can consider sensor networks to be represented by *smart plugs*.

Nodes where placed randomly through different rooms within a building floor (see figures 4.4 and 4.5). Additionally, sensors were scattered across these rooms in order to be in range of the node's networks. The population sample is sufficient to cover all the test cases proposed as a proof of concept. Additionally, the sampling technique lies on the output received by the interface provided on each node. The latter provides local information regarding some of its internal properties like state, group table, collected sensor data and the tracked addresses by the heartbeat. In turn, these will help observe the behaviour of each node according to the test cases.

The gateway prototypes are small embedded Linux systems capable of running a Java environment. Thus, the deployment of the intelligence in each node is done homogeneously. These devices have two communication interfaces. One aims to provide a medium to form WSAN while the other establishes connection to the Internet through an access point or edge router. In turn, the sensors represent small embedded devices, running an OS built to

[1]Pag. 30

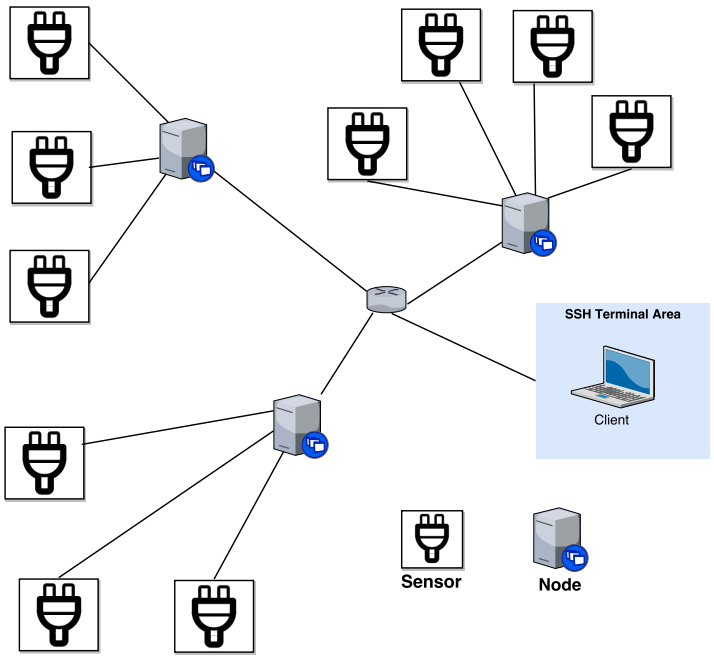

Figure 4.4: *The topology used as a concept in the proof of concept.*

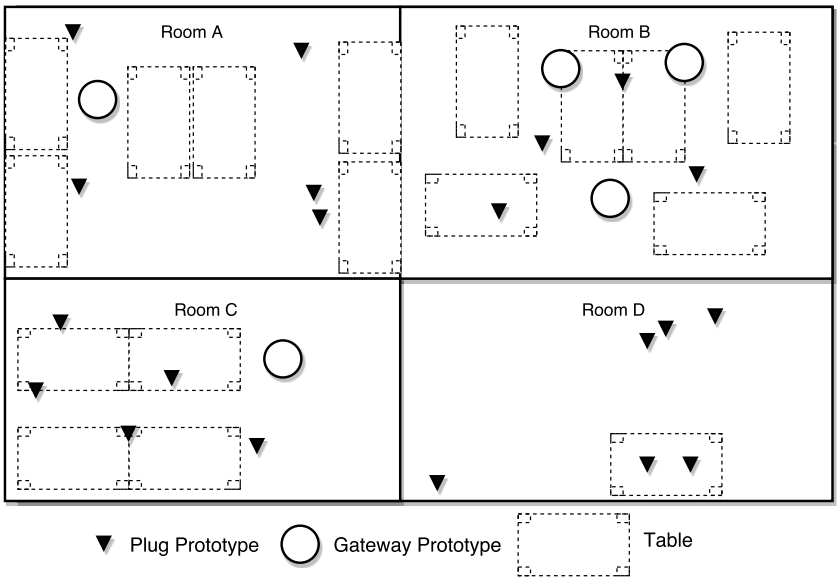

Figure 4.5: *The actual distribution used regarding plugs and nodes.*

provide a development platform capable of performing programmable computing. Additionally, these devices connect to the Wireless Sensor Area Network(WSAN) generated by each node. Ultimately, it is possible to distinguish the communication interfaces on the *gateway prototype* by assessing their purpose. Hence, the wireless interface is used to deploy the WSAN allowing plugs to establish a connection to the interface's node. On the contrary, the physical Ethernet interface is used to connect to an edge router or access point in order to communicate with other gateways. Globally its possible to grasp these two interfaces as being *ad hoc* or Internet oriented, respectively.

The distribution of sensors and nodes, as mentioned previously, took into consideration the physical constraints regarding the propagation medium of signals. Generically, solid objects like furniture and walls degrade the transmitted signals being that it increases proportionally to the distance between sensors and nodes. In this case, radio devices could communicate at distances between fifty and one hundred meters. Although, indoor environments directly imply a decrease in the transmission range. This way, we ensure that this variable, regarding the devices capabilities, does not influence the tests cases, ultimately considering the distribution of devices to be made within a twenty meters radius between *plugs* and *gateways*.

The IP protocol used throughout the experiment was IPv6, which helps us to validate the connection between E2E nodes. Moreover, a crash is simulated by forcing a machine to halt, completely shutting down its functionalities.

Message losses are a particularity of fault tolerance towards crashes. Ultimately, the redundancy provided to the FSM guarantees that message *duplicates* are handled through the *uniqueness* of the trigger associated to each message. Furthermore, *delays* are disregarded because the FSM operates over *local information* and even the absence of sufficient information is enough for the node to take action. The connection to each node is made through Secure Shells(SSH) via a laptop. The collected statistics and logs display the same data observed in real-time on each test case.

4.2 Test Cases

The test cases where designed along the software milestones. As the complexity and inherited mechanisms increased within the node's intelligence, so did the scenarios and stress situations. The test cases described thus far are sufficient to ensure the correct behaviour of the system overall. Thus, we also consider these three cases to be capable of ensuring the latter whilst future developments take place.

On the following subsections we will discuss these scenarios and what was the obtained behaviour regarding nodes interoperability.

4.2.1 Simultaneous Start

The simulation of this environment takes in consideration what is required for one to consider as an *instant* regarding some time reference. A real-life situation can be achieved by starting different nodes almost instantaneously through the use of a central machine, capable of running a start-up script engaging, at the operation level, sequentially with the remote machines. If we achieve the desired outcome, the validation of sequential node aggregation, on different instants can also be taken for granted since it is an adjacent process in regards to a *simultaneous start*. Each node performs, starting on *Detached*, following the *broadcast* of *who is there* which leads to a transition to *Looking for group*. Here, each node will check its *inbox*

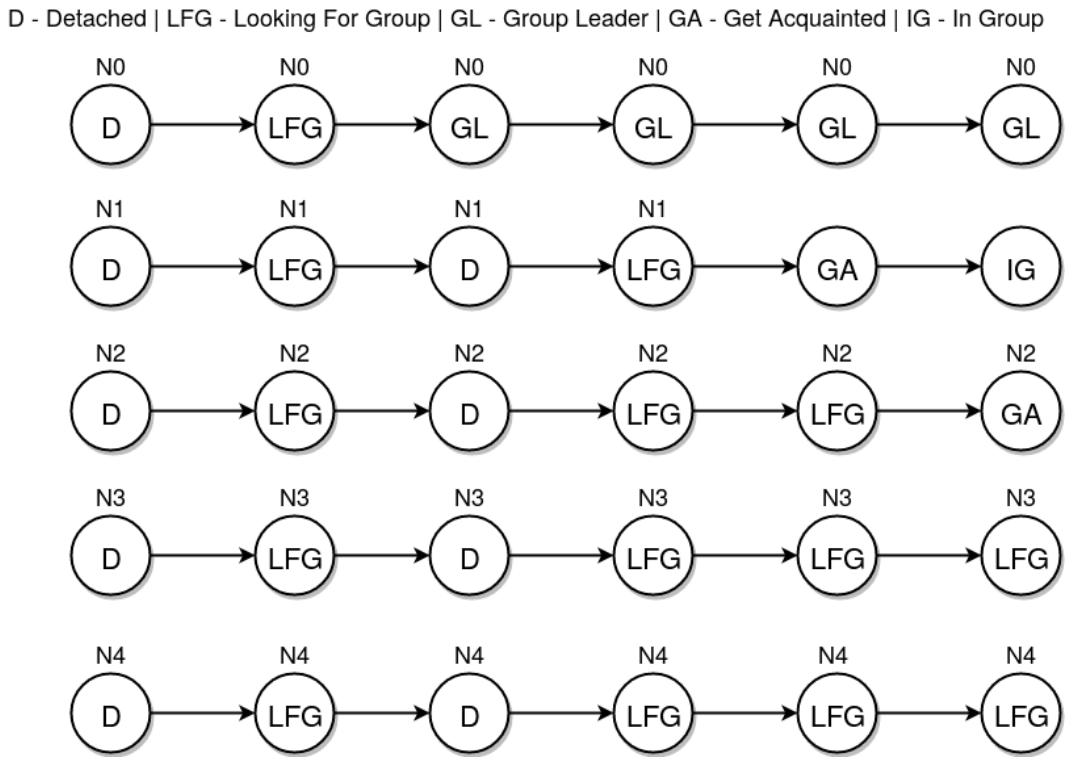

Figure 4.6: *Simultaneous start of five nodes and their states*

---

subsequently analysing the received messages. The node with the lowest UID will transition to *Group leader* while the remaining nodes go back to *Detached*, restarting the group lookup process.

Nodes eventually organize themselves and share information between each other. Given this achievement, *sequential start* is also granted. Additionally, one has to consider the simultaneous start when a group is already formed, thus, upon validating the latter, one also experimented a simultaneous burst of nodes emerging in the network. These are expected to join the newly formed group ultimately finding *equilibrium* within their set of neighbours. It is important to note that the order of the node's, regarding their transitions, towards joining the group is conceptual. There is no mechanism that guarantees a specific order according to their UID. Thus, in the experiments it is expected that the order where nodes join the group leader is not necessarily the one described in the last picture. Instead, the purpose is to achieve a stable organization in a finite number of steps. The results associated with this test case are later on presented in the results section (see section 4.3 on page 60).

4.2.2 Crashing Nodes

In this test case scenario we take advantage of the *proof* validated in the previous section. Upon reaching a stable group, some nodes may crash. Furthermore, crashes can occur at two different hierarchical levels in each group: on *members* and *group leaders*. Thus, we will crash a *group leader* and a *member*. Additionally, we chose specifically the two nodes whose UIDs are the lowest. Recalling *leader election*, the best candidate is the node with the lowest

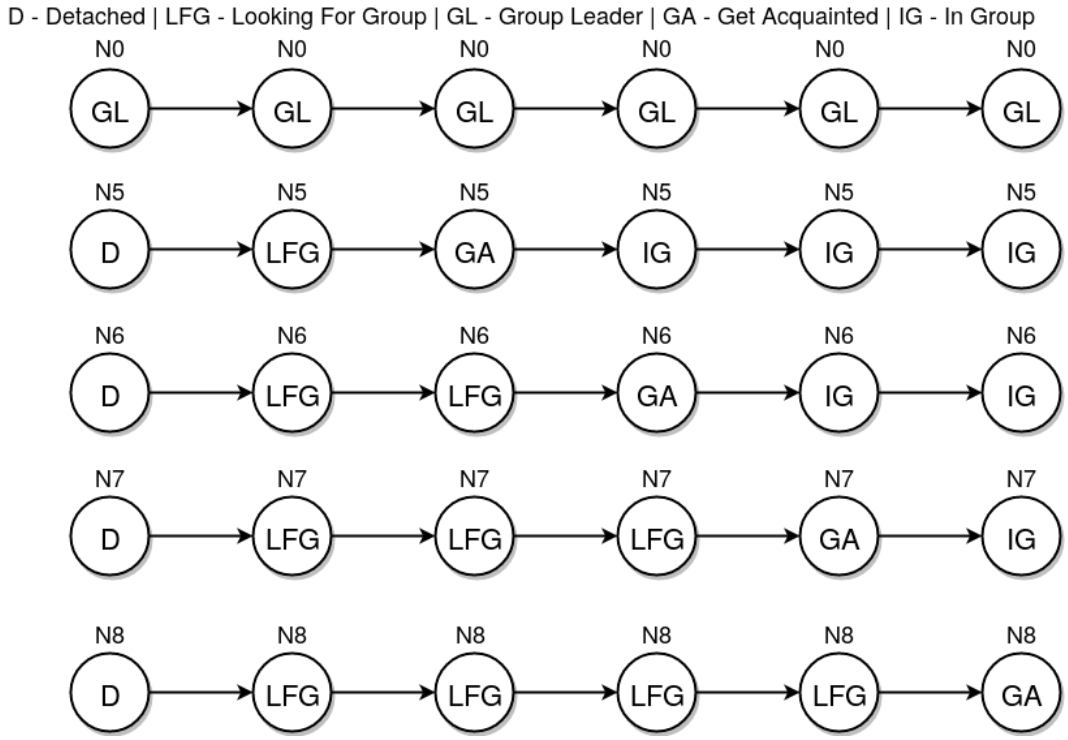

Figure 4.7: *Simultaneous burst of nodes connecting to node zero*

UID, following the *crashed leader*. Hence, Heartbeat, FSM redundancy and Group leader election mechanisms are tested because of such factor. Depending on the detection order, by the heartbeat, a group member may detect the member crash prior to the leader's. If so, it will remove the member. Later on, when the detection of a *leader crash* is made, the node will start the election. Since both crashed nodes have been removed from the group, individually by each node, election can take place on the set of active nodes in the group table. On the other hand, if a node detects a leader crash first, there will be inconsistent data during the election. This can happen when a crash occurs, besides the leader, and the crashed member is the best candidate on the election. If nodes detect a leader crash, they will assume an invalid node to be the best candidate. Even so, the overall stability of the group is ensured by the *heartbeat* towards the crashed candidate. Resulting in another election.

4.2.3 Sensor Aggregation

Sensor aggregation was conceived in an aspect where one deploys the *smart plugs* randomly spread around nodes. They boot and initiate the procedure of looking for a master to start reporting their measurements. Upon completing the protocol described in section 3.4 on page 49 the node's UI additionally shows the attached sensors. These display the output regarding the structure that maintains data related to these sensors. The information shown in table 4.1 is the conceptual table with the *attached plugs* information. Moreover, we will induce crashes in nodes in order to show the re-establishment of the environment where devices search for a new node to be connected.

Furthermore, every measurement collected from sensors represents real data transmitted

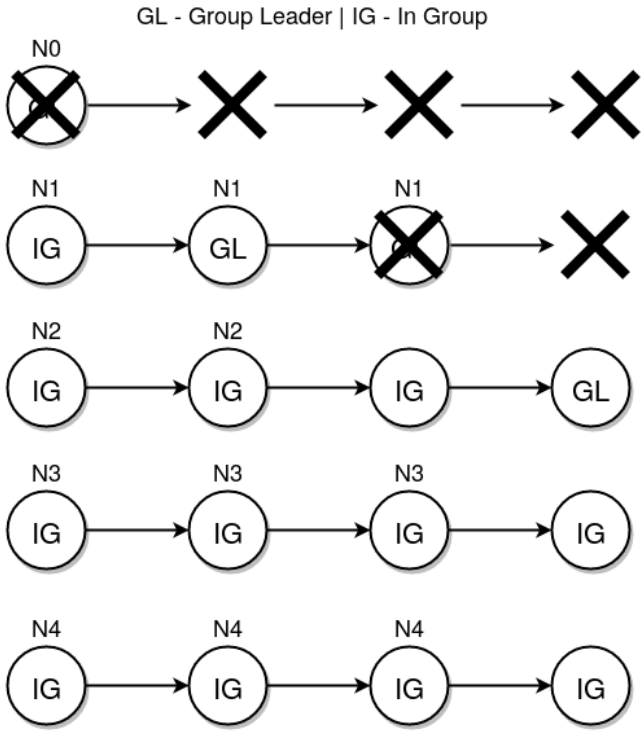

Figure 4.8: *Crash occurrence in the two lowest UID nodes*

Plug IP Address	Relay Status	Wattage	Voltage	Amperage
aaaa:0:0:0:212:4b00:3cd:7087	On/Off	W	V	A

Table 4.1: *Plug data table*

in real time. Additionally, there are unspecified cases where the plugs are attached in a piggybacked fashion, generating different electric measurements from the environment. Even so, we did not define a specific layout on plug distribution over their electrical counterparts. There are no complementary procedures which artificially introduce data to make the *proof of concept* possible. Every detail displayed on the figures shown throughout the results represent statistics, collected in a real simulation.

4.3 Results

The following sections is composed by a series of snapshots taken during the simulation of the test cases. These snapshots are supported by the information they expose regarding each nodes current state in the universe of nodes.

Firstly, every node starts in *Detached* and follows its behaviour according to the FSM. Although, we only display the moments where nodes find themselves in a stable state giving meaning to what has happened. Besides the written explanations, the modifications while a node is self-organizing only become noticeable through their state which has been already explained.

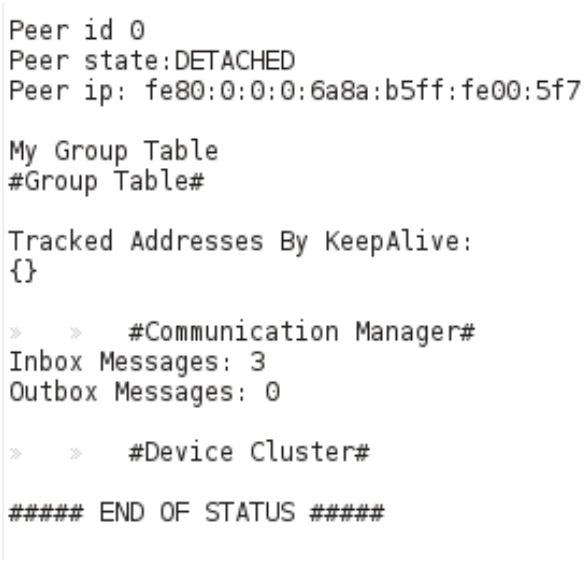

Figure 4.9: *Initial information displayed by each node, as Detached, on the connected terminal.*

In order to become familiar with a node's UI, take into consideration figure 4.9 which is a snapshot of the latter.

The information shows the Peer's, i.e. node, **ID**, **State**, **IP**, **Group table** information, **Heartbeat tracked addresses**, the **incoming** and **outgoing messages** on the **Communication manager** and ultimately, the **devices attached to the node**.

4.3.1 Simultaneous start

It is expected that during the test case of a *simultaneous start* the nodes self-organize forming a group. Additionally we also know that the node with the *lowest UID* is the one supposed to form the group while the others will have to join it. Consequently the nodes start simultaneously by broadcasting a command through a script. By being connected to the node via a terminal on a laptop, the following figures represent the UIs from the nodes in each test case:

In fact, node zero did start the group and accepted nodes one, two and three to join the group. At this stage there is one missing node which did not join, yet.

Additionally, throughout this simultaneous start, we captured the instant before and after node four joined the group, resulting later on in the broadcast of that *info* message, resulting in the information shown on figure 4.11:

Moreover, from the fourth node perspective it has acquainted with the group leader and received the information regarding its new group. The node interface displays such information upon reaching the stable state *In group* resulting shown in figure 4.12:

Due to the constraints regarding the amount of physical nodes one had to shift the deployment to a *virtual* environment. This means the tests done with more than five nodes were carried on in the same machine by launching different Java environments which communicated through a *loopback* interface. Thus, from the sixth node perspective, the final scenario could be seen as:

```
Peer id 0
Peer state:GROUP_LEADER
Peer ip: fe80:0:0:0:6a8a:b5ff:fe00:5f7

My Group Table
#Group Table#
ID: 0   | IP: fe80:0:0:0:6a8a:b5ff:fe00:5f7| isActive: true    | isLeader: true
ID: 1   | IP: fe80:0:0:0:6a8a:b5ff:fe00:22af| isActive: true    | isLeader: false
ID: 2   | IP: fe80:0:0:0:6a8b:b5ff:fe00:849| isActive: true    | isLeader: false
ID: 3   | IP: fe80:0:0:0:6a8a:b5ff:fe00:2225| isActive: true    | isLeader: false

Tracked Addresses By KeepAlive:
{fe80:0:0:0:6a8a:b5ff:fe00:22af=2, fe80:0:0:0:6a8b:b5ff:fe00:849=2, fe80:0:0:0:6a8a:b5ff:fe00:2225=2}

    #Communication Manager#
Inbox Messages: 0
Outbox Messages: 2
```

Figure 4.10: *Nodes one, two and three joined node zero group after simultaneous start.*

```
Peer id 0
Peer state:GROUP_LEADER
Peer ip: fe80:0:0:0:6a8a:b5ff:fe00:5f7

My Group Table
#Group Table#
ID: 0   | IP: fe80:0:0:0:6a8a:b5ff:fe00:5f7| isActive: true    | isLeader: true
ID: 1   | IP: fe80:0:0:0:6a8a:b5ff:fe00:22af| isActive: true    | isLeader: false
ID: 2   | IP: fe80:0:0:0:6a8b:b5ff:fe00:849| isActive: true    | isLeader: false
ID: 3   | IP: fe80:0:0:0:6a8a:b5ff:fe00:2225| isActive: true    | isLeader: false
ID: 4   | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228| isActive: true    | isLeader: false

Tracked Addresses By KeepAlive:
{fe80:0:0:0:6a8a:b5ff:fe00:22af=3, fe80:0:0:0:6a8b:b5ff:fe00:849=3, fe80:0:0:0:6a8a:b5ff:fe00:2228=3,
fe80:0:0:0:6a8a:b5ff:fe00:2225=3}

    #Communication Manager#
Inbox Messages: 0
Outbox Messages: 3
```

Figure 4.11: *Node four joins the group from node zero perspective.*

```
Peer id 4
Peer state:IN_GROUP
Peer ip: fe80:0:0:0:6a8a:b5ff:fe00:2228

My Group Table
#Group Table#
ID: 0   | IP: fe80:0:0:0:6a8a:b5ff:fe00:5f7| isActive: true    | isLeader: true
ID: 1   | IP: fe80:0:0:0:6a8a:b5ff:fe00:22af| isActive: true    | isLeader: false
ID: 2   | IP: fe80:0:0:0:6a8b:b5ff:fe00:849| isActive: true    | isLeader: false
ID: 3   | IP: fe80:0:0:0:6a8a:b5ff:fe00:2225| isActive: true    | isLeader: false
ID: 4   | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228| isActive: true    | isLeader: false

Tracked Addresses By KeepAlive:
{fe80:0:0:0:6a8a:b5ff:fe00:22af=2, fe80:0:0:0:6a8b:b5ff:fe00:849=2, fe80:0:0:0:6a8a:b5ff:fe00:2225=3,
fe80:0:0:0:6a8a:b5ff:fe00:5f7=2}

    #Communication Manager#
Inbox Messages: 0
Outbox Messages: 2
```

Figure 4.12: *Node four UI information after the simultaneous start.*

```
Peer id 6
Peer state:IN_GROUP
Peer ip: fe80::cc50:ec8f:e426:f07b:3006

My Group Table
#Group Table#
ID: 0   | IP: fe80::cc50:ec8f:e426:f07b:3000| isActive: true    | isLeader: true
ID: 1   | IP: fe80::cc50:ec8f:e426:f07b:3001| isActive: true    | isLeader: false
ID: 2   | IP: fe80::cc50:ec8f:e426:f07b:3002| isActive: true    | isLeader: false
ID: 3   | IP: fe80::cc50:ec8f:e426:f07b:3003| isActive: true    | isLeader: false
ID: 4   | IP: fe80::cc50:ec8f:e426:f07b:3004| isActive: true    | isLeader: false
ID: 5   | IP: fe80::cc50:ec8f:e426:f07b:3005| isActive: true    | isLeader: false
ID: 6   | IP: fe80::cc50:ec8f:e426:f07b:3006| isActive: true    | isLeader: false
ID: 7   | IP: fe80::cc50:ec8f:e426:f07b:3007| isActive: true    | isLeader: false
ID: 8   | IP: fe80::cc50:ec8f:e426:f07b:3008| isActive: true    | isLeader: false

Tracked Addresses By KeepAlive:
{fe80::cc50:ec8f:e426:f07b:3000=2, fe80::cc50:ec8f:e426:f07b:3001=1, fe80::cc50:ec8f:e426:f07b:3002=3,
fe80::cc50:ec8f:e426:f07b:3003=2, fe80::cc50:ec8f:e426:f07b:3004=2, fe80::cc50:ec8f:e426:f07b:3005=3,
fe80::cc50:ec8f:e426:f07b:3008=2, fe80::cc50:ec8f:e426:f07b:3007=3}

    #Communication Manager#
Inbox Messages: 0
Outbox Messages: 2
```

Figure 4.13: *Sixth node internal information.*

4.3.2 Crashing nodes

The results obtained through the crash of nodes cannot be displayed by providing information in the machine that crashed. Instead, we will describe the scenario as we present the pictures associated to the nodes that kept displaying their normal behaviour during the crash detection and afterwards.

We will induce a succession of *crashes* within the nodes, by ultimately reducing the group to just one node which we will present the information after the stabilization of the environment.

Granted five nodes already in group, we start to prove the validation of *group leader election* by *crashing* the *leader*. In this case, node zero. Upon crashing the node, we expect that node one will assume control over the group given the *election mechanism*. Additionally, it is also possible to notice through the interface that the value of the counter in the *heartbeat* is decreasing which will trigger the *crash detection* regarding the group leader. In the section of "Tracked address by keep alive" that the following address fe80:0:0:0:6a8a:b5ff:fe00:5f7 belonging to node zero, has a counter value of *one*. On the next heartbeat it will reach zero due to the crash. This same mechanism is common to all the remaining nodes. After the election, node one extends is info to its neighbours by broadcasting *info* with its group table. The validation is asserted when one checks the state and associated information of the remaining nodes. Node two UI displays *correct* information regarding the simulation. Ultimately, checking node four interface we realise the *election process* came through with success. Given that the nodes responded well to the election mechanism we additionally crashed two nodes. The nodes which we choose to do so were nodes one and three. The former is the *group leader* and the latter a simple member.

It is expected that node two assumes control over the situation (figure 4.20) ultimately stabilizing the group. This scenario will also be used as a proof towards sensor aggregation since during these tests devices where attached to nodes. Although, that will only be shown in the following subsection. After crashing the mentioned nodes we took a look at the results in node two UI. First it was node one who was firstly detected to have suffered a crash, leading

```
Peer id 0
Peer state:GROUP_LEADER
Peer ip: fe80:0:0:0:6a8a:b5ff:fe00:5f7

My Group Table
#Group Table#
ID: 0   | IP: fe80:0:0:0:6a8a:b5ff:fe00:5f7| isActive: true    | isLeader: true
ID: 1   | IP: fe80:0:0:0:6a8a:b5ff:fe00:22af| isActive: true    | isLeader: false
ID: 2   | IP: fe80:0:0:0:6a8b:b5ff:fe00:849| isActive: true    | isLeader: false
ID: 3   | IP: fe80:0:0:0:6a8a:b5ff:fe00:2225| isActive: true    | isLeader: false
ID: 4   | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228| isActive: true    | isLeader: false

Tracked Addresses By KeepAlive:
{fe80:0:0:0:6a8a:b5ff:fe00:22af=3, fe80:0:0:0:6a8b:b5ff:fe00:849=3, fe80:0:0:0:6a8a:b5ff:fe00:2228=3,
fe80:0:0:0:6a8a:b5ff:fe00:2225=3}

    #Communication Manager#
Inbox Messages: 0
Outbox Messages: 3
```

Figure 4.14: *Node zero UI as Group Leader*

```
Peer id 4
Peer state:IN_GROUP
Peer ip: fe80:0:0:0:6a8a:b5ff:fe00:2228

My Group Table
#Group Table#
ID: 0   | IP: fe80:0:0:0:6a8a:b5ff:fe00:5f7| isActive: true    | isLeader: true
ID: 1   | IP: fe80:0:0:0:6a8a:b5ff:fe00:22af| isActive: true    | isLeader: false
ID: 2   | IP: fe80:0:0:0:6a8b:b5ff:fe00:849| isActive: true    | isLeader: false
ID: 3   | IP: fe80:0:0:0:6a8a:b5ff:fe00:2225| isActive: true    | isLeader: false
ID: 4   | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228| isActive: true    | isLeader: false

Tracked Addresses By KeepAlive:
{fe80:0:0:0:6a8a:b5ff:fe00:22af=2, fe80:0:0:0:6a8b:b5ff:fe00:849=2, fe80:0:0:0:6a8a:b5ff:fe00:2225=2,
fe80:0:0:0:6a8a:b5ff:fe00:5f7=1}

    #Communication Manager#
Inbox Messages: 0
Outbox Messages: 2
```

Figure 4.15: *Node four is not receiving heartbeats from node zero*

```
Peer id 1
Peer state:GROUP_LEADER
Peer ip: fe80:0:0:0:6a8a:b5ff:fe00:22af

My Group Table
#Group Table#
ID: 1   | IP: fe80:0:0:0:6a8a:b5ff:fe00:22af| isActive: true    | isLeader: true
ID: 2   | IP: fe80:0:0:0:6a8b:b5ff:fe00:849| isActive: true    | isLeader: false
ID: 3   | IP: fe80:0:0:0:6a8a:b5ff:fe00:2225| isActive: true    | isLeader: false
ID: 4   | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228| isActive: true    | isLeader: false

Tracked Addresses By KeepAlive:
{fe80:0:0:0:6a8b:b5ff:fe00:849=2, fe80:0:0:0:6a8a:b5ff:fe00:2228=2, fe80:0:0:0:6a8a:b5ff:fe00:2225=2}

    #Communication Manager#
Inbox Messages: 0
Outbox Messages: 1
```

Figure 4.16: *Node one UI, now as Group leader.*

```
Peer id 2
Peer state:IN_GROUP
Peer ip: fe80:0:0:0:6a8b:b5ff:fe00:849

My Group Table
#Group Table#
ID: 1»  | IP: fe80:0:0:0:6a8a:b5ff:fe00:22af| isActive: true»   | isLeader: true
ID: 2»  | IP: fe80:0:0:0:6a8b:b5ff:fe00:849»| isActive: true»   | isLeader: false
ID: 3»  | IP: fe80:0:0:0:6a8a:b5ff:fe00:2225| isActive: true»   | isLeader: false
ID: 4»  | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228| isActive: true»   | isLeader: false

Tracked Addresses By KeepAlive:
{fe80:0:0:0:6a8a:b5ff:fe00:22af=2, fe80:0:0:0:6a8a:b5ff:fe00:2228=2, fe80:0:0:0:6a8a:b5ff:fe00:2225=2}

»   »   #Communication Manager#
Inbox Messages: 0
Outbox Messages: 2
```

Figure 4.17: *Node two UI*

```
Peer id 3
Peer state:IN_GROUP
Peer ip: fe80:0:0:0:6a8a:b5ff:fe00:2225

My Group Table
#Group Table#
ID: 1»  | IP: fe80:0:0:0:6a8a:b5ff:fe00:22af| isActive: true»   | isLeader: true
ID: 2»  | IP: fe80:0:0:0:6a8b:b5ff:fe00:849»| isActive: true»   | isLeader: false
ID: 3»  | IP: fe80:0:0:0:6a8a:b5ff:fe00:2225| isActive: true»   | isLeader: false
ID: 4»  | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228| isActive: true»   | isLeader: false

Tracked Addresses By KeepAlive:
{fe80:0:0:0:6a8a:b5ff:fe00:22af=3, fe80:0:0:0:6a8b:b5ff:fe00:849=2, fe80:0:0:0:6a8a:b5ff:fe00:2228=3}

»   »   #Communication Manager#
Inbox Messages: 0
Outbox Messages: 2
```

Figure 4.18: *Node three UI*

```
Peer id 4
Peer state:IN_GROUP
Peer ip: fe80:0:0:0:6a8a:b5ff:fe00:2228

My Group Table
#Group Table#
ID: 1»  | IP: fe80:0:0:0:6a8a:b5ff:fe00:22af| isActive: true»   | isLeader: true
ID: 2»  | IP: fe80:0:0:0:6a8b:b5ff:fe00:849»| isActive: true»   | isLeader: false
ID: 3»  | IP: fe80:0:0:0:6a8a:b5ff:fe00:2225| isActive: true»   | isLeader: false
ID: 4»  | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228| isActive: true»   | isLeader: false

Tracked Addresses By KeepAlive:
{fe80:0:0:0:6a8a:b5ff:fe00:22af=3, fe80:0:0:0:6a8b:b5ff:fe00:849=3, fe80:0:0:0:6a8a:b5ff:fe00:2225=2}

»   »   #Communication Manager#
Inbox Messages: 0
Outbox Messages: 0
```

Figure 4.19: *Node four UI*

```
Peer id 2
Peer state:GROUP_LEADER
Peer ip: fe80:0:0:0:6a8b:b5ff:fe00:849

My Group Table
#Group Table#
ID: 2    | IP: fe80:0:0:0:6a8b:b5ff:fe00:849| isActive: true    | isLeader: true
ID: 3    | IP: fe80:0:0:0:6a8a:b5ff:fe00:2225| isActive: true    | isLeader: false
ID: 4    | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228| isActive: true    | isLeader: false

Tracked Addresses By KeepAlive:
{fe80:0:0:0:6a8a:b5ff:fe00:2228=3, fe80:0:0:0:6a8a:b5ff:fe00:2225=0}

        #Communication Manager#
Inbox Messages: 0
Outbox Messages: 0
```

Figure 4.20: *Node two UI with node three and four still in group*

```
Peer id 2
Peer state:GROUP_LEADER
Peer ip: fe80:0:0:0:6a8b:b5ff:fe00:849

My Group Table
#Group Table#
ID: 2    | IP: fe80:0:0:0:6a8b:b5ff:fe00:849| isActive: true    | isLeader: true
ID: 4    | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228| isActive: true    | isLeader: false

Tracked Addresses By KeepAlive:
{fe80:0:0:0:6a8a:b5ff:fe00:2228=2}

        #Communication Manager#
Inbox Messages: 0
Outbox Messages: 2
```

Figure 4.21: *Node two UI with node four remaining*

---

to the group election between the remaining nodes. Thus, the UI displayed is shown on figure 4.20.

Upon such, the reaction from the heartbeat regarding the *counter values* can be observed (see figure 4.20). Node three, identified by the address *fe80:0:0:0:6a8a:b5ff:fe00:2225* on the Keep alive is displaying the value zero. Thus, the crash detection is soon take place. After a few seconds, the UI displayed that node three has been removed from the group where two and four remain.

The remaining address in the keep alive of both nodes is the address of the opposite member in the group. Ultimately, we crashed every node but the fourth. Thus, it detected the *crash* on node two and assumed control.

4.3.3 Sensor aggregation

We will present the results regarding sensor aggregation considering the *crash* of nodes enunciated before. Additionally the results shown below are related to the snapshots presented

```
Peer id 4
Peer state:IN_GROUP
Peer ip: fe80:0:0:0:6a8a:b5ff:fe00:2228

My Group Table
#Group Table#
ID: 2»  | IP: fe80:0:0:0:6a8b:b5ff:fe00:849»| isActive: true»   | isLeader: true
ID: 4»  | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228| isActive: true»   | isLeader: false

Tracked Addresses By KeepAlive:
{fe80:0:0:0:6a8b:b5ff:fe00:849=3}

»   »   #Communication Manager#
Inbox Messages: 0
Outbox Messages: 2
```

Figure 4.22: *Node four UI with node two as Group Leader*

```
Peer id 4
Peer state:GROUP_LEADER
Peer ip: fe80:0:0:0:6a8a:b5ff:fe00:2228

My Group Table
#Group Table#
ID: 4»  | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228»   | isActive: true»   | isLeader: true

Tracked Addresses By KeepAlive:
{}

»   »   #Communication Manager#
Inbox Messages: 0
Outbox Messages: 2
```

Figure 4.23: *Node four as the remaining node, now as Group Leader*

before in the last section. From them we will be able to see the aggregation of devices to the nodes in the universe and as the nodes crash, devices will search for new masters.

Stable environment

Figure 4.24 is a capture from node zero while being the *Group leader*. Below, on "Devices Cluster" we can see the information regarding the plugs and their measurements.

Node zero is the group leader. Clearly, it is handling a large portion of the overall *plugs* in the universe. Here a mechanism of load balancing could be implemented in order to balance the network (see section 5.2 on page 78). Node one (see figure 4.25), also receives data from the plugs. The group information is solid and the heartbeat is showing a consistent connection between all the remaining nodes. Additionally only two *plugs* connected to it.

Alike node one, node two, figure 4.26, is a member within its group, and has one more plug attached in comparison to the preivous node. Its keep alive is also displaying a stable connection to all the other nodes. The remaining nodes are three (on figure 4.27) and four (on fig. 4.28, which are expected to have six plugs connected to them. Hopefully the heartbeat is also showing a normal behaviour regarding the tracked addresses.

Both nodes three and four comply with what was observed in the previous members, alike their heartbeats. These have the remaining plugs attached to them.

Following the crashes of nodes zero, one and three the only *gateways prototypes* remaining were the ones with ID two and four. Thus, sensors needed to react to the crashes and search for a new master to be attached to.

On the previous pictures regarding the five UIs it is possible to count twenty-one sensors attached to the nodes. Additionally, the distribution is not load balanced resulting in the mass aggregation of *plugs* in certain nodes within the group.

Aftermath

Moreover, the nodes that crashed had seventeen *plugs* attached to them, this means a lot of entropy is generated due to these crashes. On the following picture we can grasp the final aggregation of the whole group of plugs distributed between node two and node four.

As expected, node two assumed control of the group and handled the loose *plugs* from the nodes that crashed.

Finally, node four UI (figure 4.30) with the *plugs* that did not attach to node two, plus the ones node four already was connected to.

The final scenario is happen after node two crashes, consequently all the plugs needed to re-attached to a node, in this case it is node four.

```
Peer id 0
Peer state:GROUP_LEADER
Peer ip: fe80:0:0:0:6a8a:b5ff:fe00:5f7

My Group Table
#Group Table#
ID: 0    | IP: fe80:0:0:0:6a8a:b5ff:fe00:5f7| isActive: true    | isLeader: true
ID: 1    | IP: fe80:0:0:0:6a8a:b5ff:fe00:22af| isActive: true    | isLeader: false
ID: 2    | IP: fe80:0:0:0:6a8b:b5ff:fe00:849| isActive: true    | isLeader: false
ID: 3    | IP: fe80:0:0:0:6a8a:b5ff:fe00:2225| isActive: true    | isLeader: false
ID: 4    | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228| isActive: true    | isLeader: false

Tracked Addresses By KeepAlive:
{fe80:0:0:0:6a8a:b5ff:fe00:22af=3, fe80:0:0:0:6a8b:b5ff:fe00:849=2,
fe80:0:0:0:6a8a:b5ff:fe00:2228=2, fe80:0:0:0:6a8a:b5ff:fe00:2225=2}

        #Communication Manager#
Inbox Messages: 0
Outbox Messages: 2

        #Device Cluster#
Plug Address: aaaa:0:0:0:212:4b00:3cd:7381
        {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=221518, AMPERES=13}
Plug Address: aaaa:0:0:0:212:4b00:3cd:7385
        {REGISTRY_STATUS=1, WATTAGE=1, VOLTAGE=224097, AMPERES=20}
Plug Address: aaaa:0:0:0:212:4b00:3d1:3208
        {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=221518, AMPERES=13}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6f99
        {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=222378, AMPERES=0}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6fbe
        {REGISTRY_STATUS=1, WATTAGE=3, VOLTAGE=221518, AMPERES=22}
Plug Address: aaaa:0:0:0:212:4b00:3d1:3422
        {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=221805, AMPERES=15}
Plug Address: aaaa:0:0:0:212:4b00:3cd:708f
        {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=222091, AMPERES=0}
Plug Address: aaaa:0:0:0:212:4b00:3cd:70d7
        {REGISTRY_STATUS=1, WATTAGE=2, VOLTAGE=223238, AMPERES=18}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6fc7
        {REGISTRY_STATUS=1, WATTAGE=2, VOLTAGE=222951, AMPERES=15}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6fd5
        {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=221805, AMPERES=20}

##### END OF STATUS #####
```

Figure 4.24: *Node zero, UI with aggregated sensors*

```
Peer id 1
Peer state:IN_GROUP
Peer ip: fe80:0:0:0:6a8a:b5ff:fe00:22af

My Group Table
#Group Table#
ID: 0»   | IP: fe80:0:0:0:6a8a:b5ff:fe00:5f7»| isActive: true»    | isLeader: true
ID: 1»   | IP: fe80:0:0:0:6a8a:b5ff:fe00:22af| isActive: true»    | isLeader: false
ID: 2»   | IP: fe80:0:0:0:6a8b:b5ff:fe00:849»| isActive: true»    | isLeader: false
ID: 3»   | IP: fe80:0:0:0:6a8a:b5ff:fe00:2225| isActive: true»    | isLeader: false
ID: 4»   | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228| isActive: true»    | isLeader: false

Tracked Addresses By KeepAlive:
{fe80:0:0:0:6a8b:b5ff:fe00:849=3, fe80:0:0:0:6a8a:b5ff:fe00:2228=3,
fe80:0:0:0:6a8a:b5ff:fe00:2225=3, fe80:0:0:0:6a8a:b5ff:fe00:5f7=2}

»    »    #Communication Manager#
Inbox Messages: 0
Outbox Messages: 2

»    »    #Device Cluster#
Plug Address: aaaa:0:0:0:212:4b00:3cd:7087
»    »    {REGISTRY_STATUS=1, WATTAGE=1, VOLTAGE=222665, AMPERES=11}
Plug Address: aaaa:0:0:0:212:4b00:3cd:70a9
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=223238, AMPERES=0}

##### END OF STATUS #####
```

Figure 4.25: *Node one, UI with aggregated sensors*

```
Peer id 2
Peer state:IN_GROUP
Peer ip: fe80:0:0:0:6a8b:b5ff:fe00:849

My Group Table
#Group Table#
ID: 0»   | IP: fe80:0:0:0:6a8a:b5ff:fe00:5f7»| isActive: true»    | isLeader: true
ID: 1»   | IP: fe80:0:0:0:6a8a:b5ff:fe00:22af| isActive: true»    | isLeader: false
ID: 2»   | IP: fe80:0:0:0:6a8b:b5ff:fe00:849»| isActive: true»    | isLeader: false
ID: 3»   | IP: fe80:0:0:0:6a8a:b5ff:fe00:2225| isActive: true»    | isLeader: false
ID: 4»   | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228| isActive: true»    | isLeader: false

Tracked Addresses By KeepAlive:
{fe80:0:0:0:6a8a:b5ff:fe00:22af=2, fe80:0:0:0:6a8a:b5ff:fe00:2228=2,
fe80:0:0:0:6a8a:b5ff:fe00:2225=2, fe80:0:0:0:6a8a:b5ff:fe00:5f7=2}

»    »    #Communication Manager#
Inbox Messages: 0
Outbox Messages: 1

»    »    #Device Cluster#
Plug Address: aaaa:0:0:0:212:4b00:3d1:35d5
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=221232, AMPERES=0}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6fe5
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=221805, AMPERES=18}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6fba
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=222378, AMPERES=0}

##### END OF STATUS #####
```

Figure 4.26: *Node two, UI with aggregated sensors*

```
Peer id 3
Peer state:IN_GROUP
Peer ip: fe80:0:0:0:6a8a:b5ff:fe00:2225

My Group Table
#Group Table#
ID: 0»   | IP: fe80:0:0:0:6a8a:b5ff:fe00:5f7»| isActive: true»   | isLeader: true
ID: 1»   | IP: fe80:0:0:0:6a8a:b5ff:fe00:22af| isActive: true»   | isLeader: false
ID: 2»   | IP: fe80:0:0:0:6a8b:b5ff:fe00:849»| isActive: true»   | isLeader: false
ID: 3»   | IP: fe80:0:0:0:6a8a:b5ff:fe00:2225| isActive: true»   | isLeader: false
ID: 4»   | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228| isActive: true»   | isLeader: false

Tracked Addresses By KeepAlive:
{fe80:0:0:0:6a8a:b5ff:fe00:22af=3, fe80:0:0:0:6a8a:b5ff:fe00:2228=3,
fe80:0:0:0:6a8b:b5ff:fe00:849=3, fe80:0:0:0:6a8a:b5ff:fe00:5f7=3}

»    »    #Communication Manager#
Inbox Messages: 0
Outbox Messages: 0

»    »    #Device Cluster#
Plug Address: aaaa:0:0:0:212:4b00:3d1:35d5
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220659, AMPERES=0}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6f8b
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=221805, AMPERES=11}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6ff5
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220372, AMPERES=13}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6fba
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=222091, AMPERES=0}
Plug Address: aaaa:0:0:0:212:4b00:3cd:72d6
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=221232, AMPERES=13}

##### END OF STATUS #####
```

Figure 4.27: *Node three, UI with aggregated sensors*

```
Peer id 4
Peer state:IN_GROUP
Peer ip: fe80:0:0:0:6a8a:b5ff:fe00:2228

My Group Table
#Group Table#
ID: 0»  | IP: fe80:0:0:0:6a8a:b5ff:fe00:5f7»| isActive: true»   | isLeader: true
ID: 1»  | IP: fe80:0:0:0:6a8a:b5ff:fe00:22af| isActive: true»   | isLeader: false
ID: 2»  | IP: fe80:0:0:0:6a8b:b5ff:fe00:849»| isActive: true»   | isLeader: false
ID: 3»  | IP: fe80:0:0:0:6a8a:b5ff:fe00:2225| isActive: true»   | isLeader: false
ID: 4»  | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228| isActive: true»   | isLeader: false

Tracked Addresses By KeepAlive:
{fe80:0:0:0:6a8a:b5ff:fe00:22af=2, fe80:0:0:0:6a8b:b5ff:fe00:849=2,
fe80:0:0:0:6a8a:b5ff:fe00:2225=3, fe80:0:0:0:6a8a:b5ff:fe00:5f7=2}

»   »   #Communication Manager#
Inbox Messages: 0
Outbox Messages: 2

»   »   #Device Cluster#
Plug Address: aaaa:0:0:0:212:4b00:3cd:70b8
»   »   {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=222665, AMPERES=9}

##### END OF STATUS #####
```

Figure 4.28: *Node four UI, with aggregated sensors*

```
Peer id 2
Peer state:GROUP_LEADER
Peer ip: fe80:0:0:0:6a8b:b5ff:fe00:849

My Group Table
#Group Table#
ID: 2»   | IP: fe80:0:0:0:6a8b:b5ff:fe00:849»| isActive: true»    | isLeader: true
ID: 4»   | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228| isActive: true»    | isLeader: false

Tracked Addresses By KeepAlive:
{fe80:0:0:0:6a8a:b5ff:fe00:2228=3}

»    »    #Communication Manager#
Inbox Messages: 0
Outbox Messages: 2

»    »    #Device Cluster#
Plug Address: aaaa:0:0:0:212:4b00:3cd:7087
»    »    {REGISTRY_STATUS=1, WATTAGE=1, VOLTAGE=221518, AMPERES=11}
Plug Address: aaaa:0:0:0:212:4b00:3cd:7381
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220372, AMPERES=15}
Plug Address: aaaa:0:0:0:212:4b00:3cd:7385
»    »    {REGISTRY_STATUS=1, WATTAGE=1, VOLTAGE=222951, AMPERES=20}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6fba
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220945, AMPERES=0}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6f99
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=221518, AMPERES=0}
Plug Address: aaaa:0:0:0:212:4b00:3cd:72d6
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220659, AMPERES=13}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6fbe
»    »    {REGISTRY_STATUS=1, WATTAGE=3, VOLTAGE=220372, AMPERES=22}
Plug Address: aaaa:0:0:0:212:4b00:3cd:70a9
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=221805, AMPERES=0}
Plug Address: aaaa:0:0:0:212:4b00:3d1:3422
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220659, AMPERES=20}
Plug Address: aaaa:0:0:0:212:4b00:3d1:35d5
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220086, AMPERES=0}
Plug Address: aaaa:0:0:0:212:4b00:3cd:70d7
»    »    {REGISTRY_STATUS=1, WATTAGE=2, VOLTAGE=221805, AMPERES=20}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6fe5
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220945, AMPERES=13}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6ff5
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=219512, AMPERES=20}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6ff8
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220659, AMPERES=15}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6fd5
»    »    {REGISTRY_STATUS=1, WATTAGE=1, VOLTAGE=220945, AMPERES=20}

##### END OF STATUS #####
```

Figure 4.29: *Node two, UI with aggregated sensors*

```
Peer id 4
Peer state:IN_GROUP
Peer ip: fe80:0:0:0:6a8a:b5ff:fe00:2228

My Group Table
#Group Table#
ID: 2»   | IP: fe80:0:0:0:6a8b:b5ff:fe00:849»| isActive: true»    | isLeader: true
ID: 4»   | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228| isActive: true»    | isLeader: false

Tracked Addresses By KeepAlive:
{fe80:0:0:0:6a8b:b5ff:fe00:849=2}

»    »    #Communication Manager#
Inbox Messages: 0
Outbox Messages: 0

»    »    #Device Cluster#
Plug Address: aaaa:0:0:0:212:4b00:3cd:708f
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220372, AMPERES=0}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6f8b
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220945, AMPERES=15}
Plug Address: aaaa:0:0:0:212:4b00:3cd:70b8
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=221232, AMPERES=20}
Plug Address: aaaa:0:0:0:212:4b00:3d1:3208
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220372, AMPERES=11}
Plug Address: aaaa:0:0:0:212:4b00:3cd:72f9
»    »    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=221232, AMPERES=0}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6fc7
»    »    {REGISTRY_STATUS=1, WATTAGE=2, VOLTAGE=221518, AMPERES=18}

##### END OF STATUS #####
```

Figure 4.30: *Node four UI, with aggregated sensors*

```
Peer id 4
Peer state:GROUP_LEADER
Peer ip: fe80:0:0:0:6a8a:b5ff:fe00:2228

My Group Table
#Group Table#
ID: 4    | IP: fe80:0:0:0:6a8a:b5ff:fe00:2228    | isActive: true    | isLeader: true

Tracked Addresses By KeepAlive:
{}

    #Communication Manager#
Inbox Messages: 0
Outbox Messages: 0

    #Device Cluster#
Plug Address: aaaa:0:0:0:212:4b00:3cd:7087
    {REGISTRY_STATUS=1, WATTAGE=1, VOLTAGE=221805, AMPERES=18}
Plug Address: aaaa:0:0:0:212:4b00:3cd:7381
    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220372, AMPERES=13}
Plug Address: aaaa:0:0:0:212:4b00:3cd:7385
    {REGISTRY_STATUS=1, WATTAGE=1, VOLTAGE=222665, AMPERES=18}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6f8b
    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=221232, AMPERES=13}
Plug Address: aaaa:0:0:0:212:4b00:3cd:70b8
    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=222091, AMPERES=11}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6f99
    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=221518, AMPERES=0}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6fba
    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220945, AMPERES=0}
Plug Address: aaaa:0:0:0:212:4b00:3d1:3208
    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220372, AMPERES=18}
Plug Address: aaaa:0:0:0:212:4b00:3cd:72d6
    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220372, AMPERES=15}
Plug Address: aaaa:0:0:0:212:4b00:3cd:72f9
    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=221805, AMPERES=0}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6fbe
    {REGISTRY_STATUS=1, WATTAGE=3, VOLTAGE=220372, AMPERES=20}
Plug Address: aaaa:0:0:0:212:4b00:3cd:70a9
    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=221805, AMPERES=0}
Plug Address: aaaa:0:0:0:212:4b00:3d1:3422
    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220659, AMPERES=18}
Plug Address: aaaa:0:0:0:212:4b00:3cd:708f
    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220945, AMPERES=0}
Plug Address: aaaa:0:0:0:212:4b00:3d1:35d5
    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220372, AMPERES=0}
Plug Address: aaaa:0:0:0:212:4b00:3cd:70d7
    {REGISTRY_STATUS=1, WATTAGE=1, VOLTAGE=221805, AMPERES=18}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6fe5
    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220945, AMPERES=24}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6ff5
    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=219226, AMPERES=20}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6fc7
    {REGISTRY_STATUS=1, WATTAGE=2, VOLTAGE=221805, AMPERES=15}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6ff8
    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=108323, AMPERES=27}
Plug Address: aaaa:0:0:0:212:4b00:3cd:6fd5
    {REGISTRY_STATUS=1, WATTAGE=0, VOLTAGE=220659, AMPERES=15}

##### END OF STATUS #####
```

Figure 4.31: *Node four UI, with all sensors aggregated*

Chapter 5

Conclusion

> "First we thought the PC was a calculator. Then we found out how to turn numbers into letters with ASCII and we thought it was a typewriter. Then we discovered graphics, and we thought it was a television. With the World Wide Web, we've realized it's a brochure." Douglas Adams

One has to wonder where technology will lead us. Fog Computing is an appealing approach if one aims to develop services for pervasive scenarios. Even so, this paradigm does not only handle the Internet of Things, it also gives new meaning to current computational environments, unveiling new possibilities for our society to grow as technology provides the means to do so.

5.1 Achievements

In the opposite direction to what has been deployed, we ensure that communication is established through the network without requiring any gateway to achieve such purpose. Nodes can and will communicate with each other due to their IPv6 scheme. Additionally, the P2P design imposed ensure that this system has the characteristics required by *distributed pervasive systems*, thus, communication is completely decentralized and flatly-layered. Besides such, the topology of the Fog layer internals can be arranged as required. The last characteristics remove the constraints imposed by *Zigbee* which acts as a communication interface but also as a controller within its sensor network. We have proven that it is possible to replace dedicated *daemons*, like the Zigbee controller module. to perform a specific behaviour having everything handled at the software layer with high-level procedures.

The increase of software intelligence in each node also made possible the accomplishment of an infrastructure, i.e. *groups of nodes* where they can organize and maintain themselves autonomously. The inherent mechanisms behind such behaviour make the system *fault tolerant*, providing a continuous service in case of a failure in communications between a *master* node and attached sensors. All of which results in a system capable of acting accordig to its environment, guaranteeing the delivery of services throughout IoT clusters.

After all, it is possible to consider Fog Computing as an *homogeneous* system, capable of handling the *heterogeneity* found in "things" ultimately providing a better service to the end-user which also implies an increase in quality of life.

In sum, we created a software-based entity which performs an autonomous behaviour. In a group of entities, they can recognize and be continuously aware of each other

5.2 Further Improvements

A system is only capable of surviving if it keeps improving, triggering new needs towards better services. Thus, throughout the development and maintenance of technology, further improvements are an important subject to consider as the system grows.

First of all there is a lot of potential towards generating a solution that takes advantage of Fog nodes and context networks where SaaS can also take place, oriented to the end client. Additionally, it is crucial to define an homogeneous communication between the Cloud and the Fog. This further improvements can enhance the quality of the service provided, thus, further research in this area can help the development of the Cloud-Fog continuum. For instance, regarding the electrical grid, *smart plugs* which are attached to *smart energy gateways* provide real-time data to the latter which in turn can be captured by the cloud if a certain client desires to see its current electric consumption. Moreover, using this approach with *smart meters* instead of *plugs* allows a more dynamic control over the supplied power and tariffs. This means a client can adjust the supplied power by the used of dedicated intelligence, brought by Fog nodes. Additionally, a Fog node can also control the power supply allowance granting more, or less, consequently applying the appropriate tariff.

Another improvement, and this one is considered to be crucial, is in relation to the *load balancing* of the sensor networks. There is no direct control over the attachment of nodes. Although we tried to apply a distribution directive to each group leader, the efforts proved to be worthless since in the proof of concept sensors would have constrained visibility. The sensors utilized in the *proof of concept* have a constrained signal reach, even so, a *load balancing* mechanisms which allows nodes to forward plugs to other nodes is a serious subject. This would increase the efficiency of networks and the amount processing of nodes.

The final consideration is in regards to the breakthrough that was made during the writing, self-reflective, process. As referred in section 3.3.3 on page 46, we discovered a simpler approach towards achieving stability in *simultaneous start*. Before the defense of the present document one will test this enhancement allowing an assertion over the improvement itself.

Bibliography

- [Tanenbaum and van Steen (2004)] Andrew S.Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Pearson, Prentice Hall, 2nd Edition, 2004.

- [Lickleider and Clark (1962)] Joseph C. R. Licklider and Welden E. Clark. *On-line Man-Computer Communication*, 1962. (Online)Available from: `http://cis.msjc.edu/courses/internet_authoring/CSIS103/resources/` (As in 15th May 2017).

- [Leiner et al. (2009)] Barry M. Leiner, Robert E. Kahn, Jon Postel, Vinton G.Cerf, Leonard Kleinrock, Larry G. Roberts, David D.Clark, Daniel C. Lynch, and Stephen Wolff. *A Brief History of the Internet*. ACM SIGCOMM Computer Communication Review, 39(5):1–3, October 2009.

- [Roberts (1967)] Lawrence G. Roberts. *Multiple Computer Networks and Intercomputer Communication*, 1967. (Online)Available from: `https://people.mpi-sws.org/~gummadi/teaching/sp07/sys_seminar/` (As in 6th June 2017)

- [Open Fog Consoritum (2017)] OpenFog Consortium Architecture Working Group. *OpenFog Reference Architecture*, 2017. (Online)Available from: `https://www.openfogconsortium.org/wp-content/uploads/OpenFog_Reference_Architecture_2_09_17-FINAL.pdf` (As in 5th June 2017)

- [Bonomi et al. (2012)] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, Sateesh Addepalli. *Fog Computing and Its Role in the Internet of Things*, 2012. (Online)Available from: `http://www.ce.uniroma2.it/courses/sdcc1415/progetti/Fog_bonomi2012.pdf` (As in 1st June 2017)

- [Herschel et al. (2012)] Richard Herschel, Patricia D. Rafferty. *Understanding RFID Technology within a Business Intelligence Framework*, 2012. (Online)Available from: `http://file.scirp.org/pdf/IIM20120600021_19438961.pdf` (As in 1st June 2017)

- [Hillman] Matt Hillman. *An Overview of ZigBee Networks A guide for implementers and security testers*, undated. (Online)Available from: `https://www.mwrinfosecurity.com/our-thinking/an-overview-of-zigbee-networks/` (As in 5th June 2017)

- [Bisdikian (2002)] Chatschik Bisdikian. *An Overview of the Bluetooth Wireless Technology*. IEEE Communications Magazine 39(12):86–94, December 2001.

- [Reina et al. (2013)] Daniel G. Reina , Sergio L. Toral, Federico Barrero, Nik Bessis, and Eleana Asimakopoulou. *The Role of Ad Hoc Networks in the Internet of Things: A*

Case Scenario for Smart Environments, 2013. Internet of Things and Inter-cooperative Computational Technologies for Collective Intelligence. Springer 460:89–113.

- [Karagiannis et al. (2011)] Georgios Karagiannis, Onur Altintas, Eylem Ekici, Geert Heijenk, Boangoat Jarupan, Kenneth Lin, and Timothy Weil. *Vehicular Networking: A Survey and Tutorial on Requirements, Architectures, Challenges, Standards and Solutions*, 2011. IEEE Communications Surveys & Tutorials 13(4):584–616, Fourth Quarter 2011.

- [Ko et al. (2010)] JeongGil Ko, Chenyang Lu, Mani B. Srivastava, John A. Stankovic, Andreas Terzis, Matt Welsh. *Wireless Sensor Networks for Healthcare*. Proceedings of the IEEE, 98(11):1947–1960, September 2010

- [Foster et al. (2001)] Ian Foster, Carl Kesselman, Steven Tuecke. The Anatomy of the Grid, Enabling Scalable Virtual Organizations. Journal of Supercomputer Applications, (15)3:200-222, Fall 2001.le

- [Madden et al. (2005)] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong. *TinyDB: An Acquisitional Query Processing System for Sensor Networks*. ACM Trims. Database Syst. 30(1):122-173, 2005.

- [Coulouris et al. (2012)] George Coulouris, Jean Dollimore, Tim Kindberg, Gordon Nlair. *Distributed Systems: Concepts and Design*. Addison-Wesley, Prentice Hall, 5th Edition, 2012.

- [Ding et al. (2016)] Choon Hoong Ding, Sarana Nutanong, and Rajkumar Buyya. *Peer-to-Peer Networks for Content Sharing* , 2016. (Online)Available from: `http://www.cloudbus.org/papers/P2PbasedContentSharing.pdf` (As in 13th May 2017)

- [Lalitha and Subbarao (2012)] B. Lalitha, Dr. Ch. D. V. Subbarao. *Peer-to-Peer Systems: Taxonomy and Characteristics*. International Journal of Computer Science and Technology 3(2):886–897, June 2012.

- [Grimm et al. (2004)] Robert Grimm, Janet Davis, Eric Lemar, Adam Macbeth, Steven Swanson, Thomas Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, David Wetherall. *System support for pervasive applications*. ACM Transactions on Computer Systems (TOCS) TOCS 22(4):421–486, November 2004.

- [Ganesan et al. (2003)] Prasanna Ganesan, Qixiang Sun, Hector Garcia-Molina. *YAPPERS: A Peer-to-Peer Lookup Service over Arbitrary Topology*. INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies 3:1250–1260, July 2003.

- [Tian et al. (2005)] Ruixiong Tian, Yongqiang Xiong, Qian Zhang, Bo Li, Ben Y. Zhao, Xing Li. *Hybrid Overlay Structure Based on Random Walks*. In: Castro M., van Renesse R. (eds) Peer-to-Peer Systems IV. Lecture Notes in Computer Science, Springer 3640:152–162, 2005.

- [Kubiatowicz et al. (2000)] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, Ben Zhao. *OceanStore: an architecture for global-scale persistent storage*. ACM SIGPLAN Notices 35(11):190–201, November 2000.

- [Kirrmann (2005)] Hubert Kirrmann. *Fault Tolerant Computing in Industrial Automation*, 2nd Edition 2005. (Online) Available from: `http://lamspeople.epfl.ch/kirrmann/Pubs/FaultTolerance/Fault_Tolerance_Tutorial_HK.pdf` (As in 6th June 2017).

- [Weimer and Necula (2008)] Wesley Weimer and George C. Necula. *Exceptional situations and program reliability.* ACM Transactions on Programming Languages and Systems (TOPLAS) 30(2):Article 8, March 2008.

- [Aguilera et al. (2005)] Marcos Kawazoe Aguilera, Wei Chen, Sam Toueg. *Heartbeat: A timeout-free failure detector for quiescent reliable communication.* In: Mavronicolas M., Tsigas P. (eds) Distributed Algorithms. Lecture Notes in Computer Science, Springer 1320:126–140, June 2005.

- [Defense Communications Agency (1978)] Defense Communications Agency. *ARPANET Information Brochure*, 1978. (Online) Available from: `http://www.dtic.mil/dtic/tr/fulltext/u2/a482154.pdf` (As in 2nd May 2017).

- [Zhonghong Ou (2010)] Zhonghong Ou. *Structured peer-to-peer networks : hierarchical architecture and performance evaluation*, 2010. (Online) Available from: `http://jultika.oulu.fi/Record/isbn978-951-42-6248-7` (As in 15th May 2017).

- [Cristian (1991)] Flavin Cristian. *Understanding fault-tolerant distributed systems.* Communications of the ACM 34(2):56–78, February 1991.

- [Kaur et al. (2011)] Mandeep Kaur, Manjeet Sandhu, Neeraj Mohan and Parvinder S. Sandhu. *RFID Technology Principles, Advantages, Limitations & Its Applications.* International Journal of Computer and Electrical Engineering 3(1):151–157, February, 2011.