Pedro Ákos Horváth Filipe da Costa

Degree in Computer Science and Engineering

Practical Aggregation in the Edge

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: João Carlos Antunes Leitão, Assistant Professor,
NOVA University of Lisbon

Examination Committee

Chairperson:	Jorge Carlos Ferreira Rodrigues da Cruz
Raporteur:	Hugo Alexandre Tavares Miranda
Member:	João Carlos Antunes Leitão

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2018

Practical Aggregation in the Edge

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To friends and family.

Acknowledgements

The work presented in this thesis would not have been possible with the hard work and dedication of several people with whom I worked during the course of this thesis. I would like to thank my advisor, João Leitão for his patience and dedication for helping me in achieving the goals presented here. Without his continued support none of this would have been possible.

I also want to give a special thanks to André Rosa, a student here at NOVA, that provided some results for Yggdrasil, one of the main contributions of this work. André Rosa, proved to be a very dedicated and enthusiastic student and I thank him for his hard work that proved to be fundamental for evolving Yggdrasil to a more mature state.

My thanks also extends to the Department of Informatics of the NOVA University of Lisbon and the NOVA LINCS research centre, which provided me with the necessary tools and that sparked my interest in the line of work presented here. I would like to thank the members of the research centre, and specially my colleges and friends with whom I work daily, for providing an excellent environment with laughter even in the grimmest and darkest hours.

Finally, I thank my friends and family that even though rarely see me, always supported me in my pursuit for my goals.

This work was partially supported by the European Research Project H2020 LightKone under grant agreement ID 732505, and FC&T through Project NG-STORAGE (contract PTDC/CCI-INF/32038/2017) and NOVA LINCS (grant UID/CEC/04516/2013).

Abstract

Due to the increasing amounts of data produced by applications and devices, cloud infrastructures are becoming unable to timely process and provide answers back to users. This has led to the emergence of the edge computing paradigm that aims at moving computations closer to end user devices. Edge computing can be defined as performing computations outside the boundaries of cloud data centres. This however, can be materialised across very different scenarios considering the broad spectrum of devices that can be leveraged to perform computations in the edge.

In this thesis, we focus on a concrete scenario of edge computing, that of multiple devices with wireless capabilities that collectively form a wireless ad hoc network to perform distributed computations. We aim at devising practical solutions for these scenarios however, there is a lack of tools to help us in achieving such goal. To address this first limitation we propose a novel framework, called Yggdrasil, that is specifically tailored to develop and execute distributed protocols over wireless ad hoc networks on commodity devices.

As to enable distributed computations in such networks, we focus on the particular case of distributed data aggregation. In particular, we address a harder variant of this problem, that we dub distributed continuous aggregation, where input values used for the computation of the aggregation function may change over time, and propose a novel distributed continuous aggregation protocol, called MiRAge.

We have implemented and validated both Yggdrasil and MiRAge through an extensive experimental evaluation using a test-bed composed of 24 Raspberry Pi's. Our results show that Yggdrasil provides adequate abstractions and tools to implement and execute distributed protocols in wireless ad hoc settings. Our evaluation is also composed of a practical comparative study on distributed continuous aggregation protocols, that shows that MiRAge is more robust and achieves more precise aggregation results than competing state-of-the-art alternatives.

Keywords: Edge Computing, Wireless Ad Hoc Networks, Aggregation, Frameworks

Resumo

Com o aumento do volume de dados produzido por aplicações e dispositivos, as infraestruturas de nuvem serão incapazes de processar e fornecer respostas aos utilizadores finais em tempo útil. Esta observação levou ao aparecimento do paradigma da computação na berma, que visa levar as computações para mais próximo dos dispositivos dos clientes finais. A computação na berma pode ser definida como efetuar as computações fora dos centros de dados. No entanto, isto pode ser materializado através de vários cenários considerando a multitude de dispositivos que podem ser utilizados para fazer computações na periferia do sistema.

Nesta tese, focamo-nos num cenário concreto de computação na berma, onde múltiplos dispositivos com capacidades de comunicação sem fios interagem através de uma rede ad hoc sem fios de forma a efetuar computações de forma distribuída. Pretendemos desenhar soluções práticas para estes cenários no entanto, existe uma falta de ferramentas de apoio para implementar protocolos e aplicações nestes ambientes. Para endereçar esta limitação, propomos uma nova framework, chamada Yggdrasil, que foi especificamente desenhada para apoiar o desenvolvimento e execução de protocolos distribuídos em redes ad hoc sem fios.

De forma a pavimentar o caminho para a execução de computações distribuídas nestas redes, abordamos o problema de agregação de dados distribuída. Em particular, endereçamos uma variante mais desafiante deste problema, que é a agregação contínua distribuída, onde os valores de entrada que são utilizados para computar a função de agregação podem variar ao longo do tempo de forma independente, e propomos um novo protocolo de agregação contínua, a que chamamos MiRAge.

Implementámos e validámos o Yggdrasil e o MiRAge através de uma avaliação experimental recorrendo a uma plataforma de teste composta por 24 Raspberry Pi's. Os nossos resultados mostram que o Yggdrasil fornece abstrações e ferramentas adequadas para implementar e executar protocolos distribuídos em redes ad hoc sem fios. Também apresentamos um estudo prático comparativo sobre protocolos distribuídos de agregação contínua, que mostra que o MiRAge é mais robusto e que alcança resultados de agregação mais precisos do que soluções alternativas do estado da arte.

Palavras-chave: Computação na Berma, Redes Ad Hoc Sem fios, Agregação, Frameworks

Contents

List of Figures xv

1 Introduction 1

2 Related Work 7
2.1 From the Cloud to the Edge . . . 7
2.2 The Wireless Medium . . . 9
2.2.1 MAC Layer Protocols . . . 10
2.2.2 Wireless Technologies . . . 11
2.2.3 Discussion . . . 12
2.3 Wireless Networking . . . 12
2.3.1 Wireless Ad Hoc Networks . . . 13
2.3.2 Discussion . . . 14
2.4 Wireless Ad Hoc Protocols . . . 15
2.4.1 Routing in Wireless Ad Hoc Networks . . . 16
2.4.2 Discussion . . . 18
2.5 Frameworks for building Distributed Protocols & Applications . . . 18
2.5.1 Isis and Horus . . . 18
2.5.2 APPIA . . . 19
2.5.3 TinyOS . . . 19
2.5.4 Impala . . . 20
2.5.5 Discussion . . . 21
2.6 Decentralised Communication Strategies . . . 21
2.6.1 Deterministic Communication Patterns . . . 21
2.6.2 Random Communication Patterns . . . 22
2.6.3 Discussion . . . 23
2.7 Aggregation . . . 23
2.7.1 Aggregation Computational Schemes . . . 25
2.7.2 Relevant Aggregation Protocols . . . 27
2.7.3 Discussion . . . 36
2.8 Self-Managed Overlay Networks . . . 37
2.8.1 Overlay Solutions . . . 37

2.8.2 Discussion . . . 40
2.9 Summary . . . 40

3 The Yggdrasil Framework **43**
3.1 Distributed Applications . . . 43
3.1.1 Requirements for Supporting Protocols . . . 44
3.2 Yggdrasil: Design & Implementation . . . 46
3.2.1 System Model . . . 46
3.2.2 Design Choices . . . 46
3.2.3 Architecture . . . 48
3.2.4 Implementation Details . . . 49
3.2.5 Applications in Yggdrasil . . . 53
3.3 Showcase Exercise . . . 54
3.4 Summary . . . 57

4 Multi Root Aggregation: MiRAge **59**
4.1 System Model . . . 59
4.2 Overview . . . 60
4.3 Multi Root Aggregation . . . 61
4.3.1 Aggregation Mechanism . . . 61
4.3.2 Tree Management Mechanism . . . 63
4.4 Summary . . . 67

5 Evaluation **69**
5.1 Experimental Methodology . . . 69
5.2 Experimental Tools . . . 70
5.2.1 Yggdrasil Control Process . . . 70
5.2.2 Topology Control . . . 72
5.3 Experimental Setup & Configuration . . . 73
5.4 Yggdrasil: Experimental Evaluation . . . 73
5.4.1 Protocol Implementation . . . 73
5.4.2 Performance Evaluation . . . 75
5.5 MiRAge: Experimental Evaluation . . . 78
5.5.1 Experimental Results . . . 79
5.6 Summary . . . 87

6 Conclusion and Future Work **89**

Bibliography **91**

List of Figures

2.1 Edge Spectrum. 8
2.2 Hidden Terminal Scenario. 9

3.1 Simplified Yggdrasil's Architecture. 48
3.2 Simple Discovery State. 54
3.3 Simple Discovery Handlers. 55
3.4 Simple Discovery Initialisation. 57

5.1 Distribution in Disperse Deployment. 72
5.2 Overlay Topology in Dense Deployment. 72
5.3 Broadcast Protocol: Delivery Ratio (CFD). 75
5.4 Routing Protocol: Comparison of Delivery Ratio per Node. 76
5.5 Aggregation Protocol: Precision of Aggregation Result. 77
5.6 Disperse Deployment. 80
5.7 Average Error in the Aggregated Value in Fault-free Scenario. 82
5.8 Average Error in Aggregated Value with Dynamic Input Values at Different Number of Nodes. 83
5.9 Overlay Topology after node failures. 85
5.10 Average Error in Aggregated Value with Variable Number of Node Failures. 85
5.11 Overlay Topology after link failures. 86
5.12 Average Error in Aggregated Value with Variable Number of Link Failures. . 87

Chapter 1

Introduction

Nowadays, many user-facing distributed applications rely on cloud-based infrastructures to process and manage user and application data, where client applications frequently interact with remote servers executing on data centres. This shift has been mostly motivated by increases in the user base and the amount of data that needs to be manipulated by such applications. Consider for example, social network applications such as Facebook, Twitter, or Instagram that have billions of users accessing the cloud at once, or IoT devices in a smart city that produce huge amounts of data that is uploaded to the cloud for processing [22, 75].

However, solely relying on cloud infrastructures can have its disadvantages. The increase on the required resources of the cloud also leads to an increase in the cost for application providers; the latency experienced by end users, that must constantly contact remote servers, increases; and security concerns arise from outsourcing data storage and computations [32] to infrastructures controlled and managed by third parties. These issues have motivated the need to move computations outside data centre boundaries, towards the edge of the system. This has led to the emergence of the edge computing paradigm.

Edge computing can be (broadly) defined as performing computation outside the cloud boundary, at devices that are closer to the source of data, which nowadays are mostly end user devices [75]. As one gets farther away from the cloud, one encounters multiple types of devices where intermediate computations can be performed. First ISP servers, then gateways, laptops, smart phones, and sensors/actuators. It is also important to note that, further from the cloud the number of devices increases, while the individual amount of resources per device decreases. Executing computations in such environments

becomes therefore, a complex task, as one has to deal not only with a large number of devices, but also manage their limited resources [52].

Another important aspect that one has to consider as we move towards the edge of the system, is that infrastructure support becomes increasingly lacking. This translates, for instance, in the fact that devices will be connected by limited capability links. In particular, one can expect to find most devices being connected through the wireless medium, that offers a potentially unreliable communication environment. Due to this, in this work we focus on the particularly challenging setting where all devices communicate through an infrastructure-less wireless medium.

Furthermore, achieving general purpose computations in the edge is not a trivial task. Consequently, we have opted to focus our efforts on a particular type of computation: *data aggregation*. This is a first, and essential, step towards enabling general computation at the edge. Aggregation can be computed in-network as devices exchange information and cooperate among them. TinyDB [60] showcases how this can be done in order to create a (distributed) database in the context of sensor networks. Additionally, Astrolabe [84] has shown how aggregation can be leveraged to perform monitoring tasks in large scale systems. Finally, ZebraNet [45] presents the importance of having a support runtime for a self-managed network. All of these application scenarios showcase the relevance of efficient aggregation mechanisms in paving the way for enabling the construction of more complex applications.

Aggregation mechanisms are provided by distributed aggregation protocols. However, to bring aggregation protocols to light in the context of edge computing, we must implement and evaluate them in real deployments. As mentioned before, at the edge of the system we will find commodity devices capable of communicating through wireless channels hence, we should leverage these devices that will become a relevant part of an edge system.

Motivation

The work presented on this thesis is motivated by the following:

- Addressing the lack of support tools for the development and execution of protocols and applications in wireless ad hoc networks, that can execute in commodity devices, running common operative systems.
- Develop efficient and reliable distributed aggregation protocols that allow all processes in a large-scale distributed system to obtain an accurate view of the system state.
- Attain a comprehensive understanding on the operation and behaviour of wireless ad hoc protocols through practical evaluation, by executing these protocols in real devices under real conditions.

Contributions

This work provides four main contributions, and can be summarised as follows:

1. A framework, named Yggdrasil, for the development and execution of protocols and applications for wireless ad hoc networks. The framework provides programmers with a set of abstractions. Namely, an event driven execution model; low level communication primitives; interaction mechanisms between protocols and applications executing in the same process; and mechanisms for multi-threaded execution that hide concurrency issues from protocol and application developers.

2. A novel aggregation protocol, named MiRAge, that builds and maintains a tree topology to support the computation of the aggregation function in a fully decentralised and fault tolerant fashion, without depending on a pre-configured or static root. This tree is built and maintained while the aggregation process evolves, meaning that there are no additional messages exchanged by the protocol.

3. An experimental evaluation of the Yggdrasil framework, that shows its correctness and validates the usefulness of the framework for implementing multiple types of distributed protocols for wireless ad hoc networks.

4. An experimental evaluation of distributed aggregation protocols using a real testbed composed of 24 Raspberry Pi, that shows the benefits of our own aggregation solution in relation to the state-of-the-art.

Research Context

The work conducted in the thesis is an integral part of the research agenda of the H2020 LightKone: Lightweight computation for networks in the edge (Project number 732505), founded by the European Commission.

Part of the contribution of this work appears as part as the "D5.1: Infrastructure Support for Aggregation in Edge Computing" and "D5.2 - Report on Generic Edge Computing" deliverables produced by the LightKone Consortium, in January 2018 and July 2018, respectively.

Publications

The work presented here generated the following publications:

Main:

- *Practical Continuous Aggregation in Wireless Edge Environments*
  Pedro Ákos Costa and João Leitão.
  Proceedings of the 37th IEEE International Symposium on Reliable Distributed Systems (SRDS 2018), October 2-5, 2018.

Portuguese:

- *Agregação Contínua e Prática em Ambientes Sem Fios na Berma*
  Pedro Ákos Costa and João Leitão.
  Proceedings of the 10th Simpósio de Informática (INFORUM'18), Coimbra, Portugal, Sep 2018.

- *Yggdrasil: Uma Framework para Desenvolvimento e Execução de Protocolos em Redes Ad Hoc*
  Pedro Ákos Costa and João Leitão.
  Proceedings of the 10th Simpósio de Informática (INFORUM'18), Coimbra, Portugal, Sep 2018.

Other:

- *Towards Enabling Novel Edge-Enabled Applications*
  João Leitão, Pedro Ákos Costa, Maria Cecília Gomes, and Nuno Preguiça.
  Technical Report, May 2018.
  https://arxiv.org/abs/1805.06989

- *A Case for Autonomic Microservices for Hybrid Cloud/Edge Applications*
  João Leitão, Maria Cecília Gomes, Nuno Preguiça, Pedro Ákos Costa, Vitor Duarte, André Carrusca, André Lameirinhas, and David Mealha.
  Technical Report, September 2018.

Thesis Structure

The rest of the document is organised as follows:

- Chapter 2 presents edge computing in more detail, discusses key properties of the wireless medium and the variants of wireless ad hoc networks, providing the description of the environment we will operate on. Additionally, Chapter 2 presents wireless ad hoc routing protocols in contrast to protocols that operate in wired environments and provide an overview on existing frameworks for building and executing distributed protocols and applications. This is followed by a study on decentralised computation, where we study multiple distributed aggregation protocols and provide some insight on algorithms used to build efficient tree topologies where aggregation can be performed.

- Chapter 3 details the design and implementation of the Yggdrasil framework. Presenting the rationale for its design, guided by properties of various types of distributed protocols for wireless ad hoc networks; its architecture and main components, as well as the details of its implementation. For completeness, Chapter 3 also

offers a simple example of the use of Yggdrasil for building a simple distributed protocol.

- Chapter 4 proposes a novel distributed aggregation protocol, named Multi Root Aggregation, or simply MiRAge. We introduce the rationale for the design of the protocol and fully specify the algorithm.
- Chapter5 presents an extensive experimental evaluation of the contributions of the presented work. First, we evaluate the use of Yggdrasil given the implementation of the set of protocols that served to guide its design and implementation. Second, we perform a comparative study of distributed aggregation protocols that includes our own protocol: MiRAge.
- Chapter 6 concludes the thesis and provides directions for future work.

Chapter 2

Related Work

In this Chapter we further detail issues related to cloud infrastructures and motivate the need for solutions that reside in the edge (Section 2.1), which guided us to study infrastructure-less networks. As such, we describe the wireless medium and the challenges associated with its use (Section 2.2), discussing how the wireless medium can be used to materialise an infrastructure-less network (Section 2.3).

We study wireless ad hoc routing protocols (Section 2.4) to understand fundamental differences between protocols that operate over wired and wireless networks. This is followed by an overview of existing frameworks and toolkits for implementing and executing distributed protocols and applications (Section 2.5).

Lastly we focus on distributed computations, by first providing an overview of (decentralised) communication strategies (Section 2.6), as they are essential to perform any type of distributed computing. We follow this by presenting distributed data aggregation (Section 2.7), as our main focus of distributed computing, and finish the Chapter by providing some insight on distributed algorithms to build robust tree topologies where aggregation can be performed efficiently (Section 2.8).

2.1 From the Cloud to the Edge

The cloud computing paradigm has gained significant momentum in recent years. Cloud providers such as Amazon and Google have millions of user using their services to deploy applications and store data. In order to do so, the cloud environment presents a multitude of commodity servers working together in one data centre or across multiple data centres scattered throughout the world, which gives the possibility to tackle large scale computation problems with solutions like MapReduce [24], and provide backend services for large scale applications, such as Facebook.

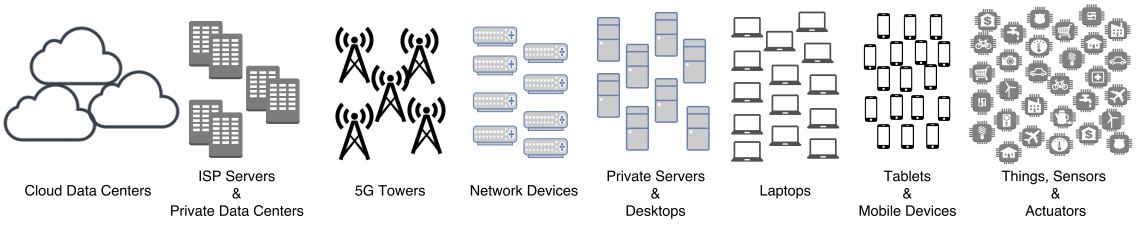


Figure 2.1: Edge Spectrum.

However, the cloud is not a panacea that solves all challenges. Data privacy presents itself as one of the main problems of the cloud as discussed in [32]; furthermore, issues regarding data management, resource allocation, and scalability still persist [26]. With the increasing number of applications and users accessing such applications, these issues start to become more pronounced. Applications and devices producing large amounts of data (e.g., daily-life devices; IoT devices; sensors; among others) present a significant overhead in data transfer and data management for the cloud. Even though the cloud infrastructure is said to be elastic, the network infrastructures that connect users to the cloud is not. The possibility for the network to become a bottleneck for users is highly likely, as it is estimated that the total amount of data produced by people and devices will reach 500 zettabytes by 2019 [22, 75], which can therefore, lead to a significant increase in latency experienced by the users, or even full disruption of applications operation.

With this in mind, we must look for solutions that are outside of the cloud, hence we must begin to look towards the edge of the system and on how, and what can be leveraged to relief the cloud from doing all the work. Outside the cloud there are (smaller) private or regional data centres; 5G towers; routers and gateways that can have some storage and computing capability; private servers; end user devices, such as laptops and smartphones; IoT devices; sensors and actuators; among other computational resources [52]. Figure 2.1 captures the spectrum of possible devices where computations can, and are, performed. We notice that farther from the cloud the devices become less powerful, but in higher numbers while having lower latency for end users. Given this scenario, the same computation and storage opportunities that were possible in the cloud become unfeasible thus, a new computational paradigm is required: the edge computing paradigm.

However, there is no unified vision on what edge computing is, due to the fact that every computation that can be executed outside the cloud can be considered to fall in the scope of edge computing, and any network/computational resource that is outside a data centre, can be viewed as an edge device. As such, edge computing materialises in various and different forms. Through fog computing [19, 61, 90], which tries to extend the cloud's infrastructure closer to data sources and end users. Through intelligent IoT or sensor networks that are able to pre-process data cooperatively without the need of the cloud [8], and even in other forms as described in [46, 81].

Consequently, the edge computing paradigm encompasses any possible computations

near data sources, as such, in the context of this thesis we focus on edge devices that are neither sensors, which have limited resources, nor as powerful as servers. Instead, we focus on edge devices that have some computational power, can be very disperse and have no infrastructural support (e.g., no access to routers or access points). A concrete example of such device are general purpose micro-computers such as the Raspberry Pi.

This implies that wiring devices together might not be possible hence, we must consider wireless as a possible means for supporting all communication and cooperation between devices in such edge computing settings.

2.2 The Wireless Medium

The wireless medium is a shared medium through which devices can communicate with each other by transmitting radio waves using a given frequency. The radio waves are generated by transceivers, or radios, which are usually omnidirectional and can reach every device that is within its transmission range. Consequently, when a message is sent from a wireless device, the message is delivered to every other device that is within that range, this is commonly refereed to as *one hop broadcast* [82]. This abstraction allows to build point-to-point and multicast primitives within the range of a sender by having receivers, for whom the message is not addressed, to simply drop that message.

Since the wireless medium is a shared medium between any device that intents to transmit on the same frequency, wireless communication suffers from contention, which is usually a byproduct of collisions. When two devices transmit a message at the same time and are within range of each other, a collision will most likely happen. Thus, none of the messages will be successfully received by other devices resulting in message loss.

To better understand the consequences of collisions, consider three devices A, B, and C, as in Figure 2.2, where A can only communicate with B; B can communicate with A and C; and C can only communicate with B. Now assume that, A and C intend to send a message to B. As A and C are unaware of each other they will both send the message to B simultaneously, causing a collision between the two messages and making B unable to receive either of them [74]. This has been described in the literature as the *hidden terminal problem*.

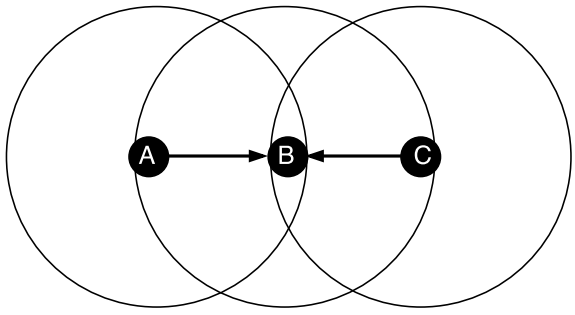


Figure 2.2: Hidden Terminal Scenario.

Extending this scenario to another with larger numbers of devices, where devices need to forward messages between them, we can incur in the *broadcast storm problem* [69]. Nodes will continuously transmit, or retransmit, messages that will collide and thus, saturate the network, rendering any form of communication impossible.

A strategy to minimise the effects of the hidden terminal and broadcast storm problems is to have nodes perform some form of access control to the wireless medium (we discuss how this is performed in detail further ahead). Unfortunately, this can lead to the *exposed terminal problem* [74]. This happens when a node decides not to transmit even if its transmission does not interfere with other ongoing transmissions, which can consequentially lead to a significant decrease of the network capacity, potentially disrupting the execution of protocols and applications over the wireless medium.

2.2.1 MAC Layer Protocols

To address the contention and collisions on wireless networks, many medium access control (MAC) layer protocols have been proposed [25]. We will discuss the most commonly used, which are the Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) and Time Division Multiple Access (TDMA).

CSMA/CA In CSMA/CA [92] when a device wants to transmit, it first listens (senses) the medium to see if no other device is transmitting. If the medium is occupied, then the device waits for some time and senses the medium again. If the medium is free, it proceeds to send a Request-To-Send (RTS) frame and waits for a Clear-To-Send (CTS) frame from the intended receiver. By using the RTS/CTS mechanism CSMA/CA avoids the hidden terminal problem, if no CTS frame is heard, it means that a collision has occurred due to a hidden terminal. In this case, the sender will wait some time and restarts the mechanism to access the medium (by sending the RTS frame). When the CTS frame is received, the device proceeds to transmit the data frame and waits for an ACK frame as a positive acknowledgement that the data frame was received. If the ACK frame is not received, then a retransmission must take place.

There is a exception when using CSMA/CA, when the data frame's destination is the (physical layer) broadcast address, the RTS/CTS mechanism cannot be used and ACK frames are not sent after a successful transmission [82].

TDMA The TDMA [89] protocol tries to ensure collision-free transmissions, by defining a schedule of non-overlapping time slots for each device. TDMA divides a time period into fixed-length slots where transmissions are possible, when two devices have the same slot they must coordinate to change their slots in order to avoid collisions. Needless to say that finding the optimal time slots for every device in a network in a distributed manner, is a NP-complete problem [28] thus, TDMA can

be highly inefficient in some cases, computing sub-optimal schedules that can lead to the low utilisation of the wireless medium.

Furthermore, the wireless environment presents more challenges than contention and collisions. A wireless link is usually more unstable than a wired link, since it is more affected by the physical environment [1]. This can lead to anomalies such as high bit error due to noise in the wireless medium. Additionally, links might not always be symmetric [50], due to different transmission power of radio devices, which might make it hard to reason about the behaviour of protocols and applications.

These are some considerations that must be taken into account when designing protocols and applications that leverage the wireless medium for communication. These are problems that are transversal to our work. Another aspect we consider here is the variety of available technologies that use the wireless medium. Each one has different reach, different data rates, different frequency ranges, availability on devices, and other particularities. This is specially important since in this work we focus on practical implementations, and must choose a developing platform.

2.2.2 Wireless Technologies

There are many wireless technologies that allow us to build a wireless network [35, 66, 71, 77]. We briefly discuss three main technologies currently used for wireless networking that use the 2.4Ghz frequency range. WiFi (Wireless Fidelity), which is the most common on commodity devices; Bluetooth, used for connecting peripheral devices; and ZigBee, that is commonly used in the context of sensors.

Wifi

The WiFi [77] technology is the most common in nowadays commodity hardware. It can be easily found in our laptops, mobile phones, tablets, gaming consoles, and many others. WiFi has a range of around one hundred meters and a data rate that ranges from approximately 50 Mbit/s to 150 Mbit/s, depending on the communication standard employed by the device.

Bluetooth

Bluetooth [35] is the technology most commonly used to connect peripheral devices, such as keyboards, computer mouses, headphones, among others. It is designed to work in short range having a reach in the order of ten meters. This technology was also designed to be energy efficient, having low data rates (around 1 Mbit/s however, this can differ with version and specification). It is usually seen operating in a master/slave configuration, where a device can be either a master or a slave. However, each master can only have up to seven active slave devices.

ZigBee

ZigBee [71] is one of the main technologies used in sensor networks and IoT devices, having a reach that varies between ten meters and one hundred meters, depending on the power output of devices and other factors (e.g., indoor and outdoor), and a very low data rate transmission (around 256 Kbit/s). As sensor and IoT devices are energy constrained, ZigBee is also designed to be highly energy efficient. A ZigBee network is composed by three types of devices: *coordinators*, *routers*, and *end devices*. Every network must have a coordinator that builds and configures the network; routers are then used to relay information between nodes; whereas end devices operate as data producers/consumers that must be connected to the coordinator, either directly or through routers.

2.2.3 Discussion

A fundamental property our development platform must provide is the potential for scalability. As such, we exclude devices that can only communicate through bluetooth, as bluetooth networks have a scalability limit. A second fundamental property of the network that we look for, is that it must be fully decentralised. The formation of the network should not depend on the activity of special devices in the system. Consequently, we exclude ZigBee as a communication technology.

This leaves us with WiFi. WiFi is mostly used by having devices connect to networks with the help of an access point. However, it support another mode of operation, the ad hoc mode which does not require infrastructure support. In the following section, we give detail on these networks.

2.3 Wireless Networking

Wireless networks enable multiple wireless devices to communicate, providing abstractions that hide concrete deployment aspects such as the relative positioning of the devices. The most common form of wireless network is based in infrastructure, where a set of devices connect to an access point or a router with access to the Internet or other infrastructure networks (e.g., a local area network).

However, in the context of large-scale IoT or sensor networks deployments, it is hard to have all devices in reach of an access point or wireless router. In these scenarios, we have to resort to infrastructure-less networks, where devices interact directly among themselves. These networks are usually refereed as *wireless ad hoc networks*.

Since we target systems that operate with no access to infrastructure and where the number of devices can grow at any point in time (in an organic fashion), ad hoc networks are a key aspect to be considered. We now discuss these networks in more detail.

2.3.1 Wireless Ad Hoc Networks

A typical ad hoc network is composed by a set of nodes that are interconnected arbitrarily forming a multi-hop network (i.e., a network where not all devices are directly reachable by a radio transmission by all other devices). These nodes can only communicate with their direct neighbours using one hop broadcast or point-to-point communication primitives. As such, there is usually no routing involved in these networks and routing among devices must be performed at the application level. Additionally, these networks might not even use IP addresses to identify nodes, and can rely solely on MAC addresses for communication.

Given the capability of nodes to interconnect in an arbitrary fashion, an ad hoc network is highly dynamic which led these to gain some popularity in use cases such as disaster relief operations [37], military operations [78], monitoring and sensing applications [91], and even in supporting communication among vehicles [31].

However, these networks can be materialised in various forms given the network's objective and the devices that compose it.

Wireless Mesh Networks

Wireless mesh networks build upon ad hoc networks to extend an existing network infrastructure [5]. A mesh network is composed by two types of devices, *mesh routers* and *mesh clients*. Mesh routers are assumed to be static and are responsible for forwarding packets in transit between devices. Mesh clients on the other hand, connect to the routers and only send and receive messages, being typically devices with lower amounts of resources. With this, a mesh network is capable of performing routing, enabling mesh clients to access infrastructural resources, such as file servers, application servers, and Internet gateways.

However, in typical mesh networks, routing requires explicit support. To ensure (effective) routing connectivity, mesh routers may need to be built using specialised hardware containing multiple antennas to leverage the full spectrum of the wireless medium [72]. Moreover, when considering highly dynamic networks, with nodes joining and leaving concurrently, these networks might experience high control overhead for maintaining routing information up-to-date.

Mobile Wireless Ad Hoc Network

In the literature [4, 29, 33, 39, 70], when wireless ad hoc networks are mentioned, they are usually referring to a mobile wireless ad hoc network (MANET), where nodes that materialise the network are mobile. This means that nodes will change position, and consequently, neighbourhoods, frequently, leading mobile wireless ad hoc networks to be highly dynamic.

This in turn, makes routing between the nodes even more complicated. As routes frequently change, it is not feasible to hold information such as the minimum number of hops between all pairs of nodes, with different techniques being explored to address this challenge [11, 47]. Furthermore, some also consider that the nodes composing the network might be resource constrained, as such, extending the network's life should be considered when incorporating routing in MANETs [64].

Vehicular Wireless Ad Hoc Network

These networks are a particular case of MANETs. Vehicular wireless ad hoc networks [31] (VANETs) assume that the network is mostly composed of vehicles that communicate with each other and with (fixed) stations that are positioned near the road to get useful information or access resources beyond vehicles computers.

These networks suffer from the same problems as MANETs related to routing however, the network's objective is fundamentally different. These networks may service vehicles to be more autonomous or to intelligently collect information from sensors scattered throughout an area [9]. Although, VANETs are composed by vehicles and, even though present an interesting edge computing scenario, they are outside the scope of this thesis.

Wireless Sensor Networks

We can perceive wireless sensor networks as being wireless ad hoc networks composed of vast numbers of specialised sensors that collect data from the environment. These sensors are usually very small and inexpensive [6], being perfect for mass deployment. Moreover, sensor networks can be deployed remotely on harsh environments, such as underwater [7] or in volcanic regions [87]. However, due to their size, sensors tend to be very resource constrained and therefore, extending their battery life tends to be the main focus of research on sensor networks.

A typical sensor network architecture is composed by a sink (or gateway) node that can be connected to an infrastructure, and a number of sensors that are connected to the sink or to each other, forming a multi-hop wireless network. In such deployments, the goal of each sensor is to report its collected data to the sink node, which is then responsible for processing that data [50].

2.3.2 Discussion

All of the previously mentioned wireless networks are particularly tailored for different edge scenarios. Wireless mesh networks are usually employed to extend existing infrastructures or to facilitate the access to resources in settings such as companies and office spaces, where there is (some) lack of infrastructure. In this context, routing might be a desirable feature to facilitate the interactions with these resources, that can be Internet gateways, file servers, or even printers. We note that these networks are not designed to

allow distributed computations to occur across the devices that compose it. Moreover, particularly on scenarios where all devices belong to a single entity (e.g., a company), one could leverage these networks to take advantage of idle resources in these devices as proposed in the context of peer-to-peer systems [10, 79].

Mobile and vehicular wireless ad hoc networks focus on the challenges that arise from the fact that devices are not stationary, which introduces additional dynamics for instance, regarding the stability of the neighbours of each device. The focus of these networks is usually to provide effective routing schemes among devices to support some form of cooperation among them, such as propagating information or sharing context-sensitive data.

Wireless sensor networks focus on an extreme edge computation scenario, where large numbers of devices, typically resource constrained and heterogeneous in nature, interact among each other to propagate sensed information towards a sink node that exists somewhere in the network. Often, these networks are supported by low cost wireless technologies such as ZigBee. In this context however, we note that some of the processing performed by the sink node could be performed in the network, if there were some nodes in it that had additional memory and computational capability (and eventually a source of power).

Altogether, these networks build upon a wireless ad hoc network to provide additional functionalities and to address specific challenges characteristic of each of these scenarios.

In this work we aim at bringing computations towards the edge of the system, which implies that computations should be performed in-network, as devices exchange information among them. Contrary to the ad hoc networks presented above, whose focus lies on enabling any pair of nodes to exchange information through routing in adverse scenarios, we envision systems where devices interact naturally to achieve a common goal. Moreover, we aim at building systems that can grow organically without support from infrastructure and with self-organising capabilities.

To achieve these goals we will operate directly at the ad hoc network level, without assuming any form of routing, nor any particular interaction pattern from devices. More specifically, routing should be handled at the application level such that it can operate specifically to provide the minimal functionality required by protocols or applications with minimal cost and overhead in the wireless medium.

We intent to develop practical solutions for wireless ad hoc settings. As such, we must understand key aspects of protocols for these networks. Because the most prevalent class of protocols developed for wireless ad hoc networks are routing protocols (as described in this Section), in the next Section we study how do these protocols operate.

2.4 Wireless Ad Hoc Protocols

We look into the operation of distributed protocols in wireless ad hoc networks. The most prevalent protocols that can be found in the literature regarding wireless ad hoc networks

are routing protocols (as it is implied in the previous Section). Wireless ad hoc routing protocols have key distinctions in relation to wired/infrastructure routing protocols, as the environment on which they operate over is fundamentally different.

2.4.1 Routing in Wireless Ad Hoc Networks

In a nutshell, a routing protocol is responsible for delivering a message from a node to any other specific node in the system.

In wired networks routing among nodes is supported by the IP layer that is well defined. Routing relies on specialised (and dedicated) hardware (switches and routers). Routing in wired networks usually rely on link-state protocols [20] that assume that wired links are not only stable but also possible to monitor.

In sharp contrast, in wireless ad hoc networks there is no dedicated hardware assisting in routing activities. This requires nodes to coordinate among themselves to achieve multi-hop communication. In some cases however, routing is avoided by transmitting messages through flooding of the network, and then have each node inspect the contents of the message to decide if it should process or discard it.

We now present several routing protocols that have been proposed to enable routing capabilities in wireless ad hoc networks.

Ad-Hoc On-Demand Distance Vector Routing

Ad-Hoc On-Demand Distance Vector Routing [70] (AODV) is a routing protocol designed for large scale and highly dynamic wireless ad hoc networks. It is a reactive routing scheme, in the sense that routes are established when a node wants to contact another node to which a route is (yet) unknown.

The algorithm is based on sending a message throughout the whole network in search of the destination node. These messages contain a sequence-number to overrule older messages, a hop-counter to determine the shortest path, and the source node's identifier (IP address). Nodes forward these messages, incrementing the hop-counter and updating their routing table (storing the source identifier and the hop-counter); when these messages reach the destination node, it replies to the message with the lowest hop-counter; when the reply reaches the source, a route has been established.

The routes established are temporary, as they are invalidated from time to time. This allows AODV to handle link changes, but on the other hand, it might also cause a high amount of retransmissions in order to reestablish routes that are frequently needed.

Optimized Link State Routing Protocol

In contrast to AODV, the Optimized Link State Routing Protocol [40] (OLSR) is a proactive routing protocol that relies on periodic routing table updates. The protocol has neighbouring nodes coordinate to decide on *multipoint-relay* nodes that cover two-hop neighbour nodes which are then tasked with the forwarding of data packets.

The algorithm begins by having each node sense their direct neighbours and then locally broadcasting their list of neighbours. The nodes obtain information about their two-hop neighbours and decide which nodes will be used as multipoint-relays, by computing the intersections between two-hop neighbours. The nodes then locally broadcast their list of multipoint-relays nodes (notice that it is the same message as before, but only containing a subset of neighbours). The chosen multipoint-relay nodes will broadcast the message throughout the network, while the nodes that are not present in the list will only process it. Nodes will update their routing table with the received information about multipoint-relay nodes which will then be used to establish routes.

The use of these multipoint-relay nodes, allows OLSR to minimise the effects of a full network broadcast, thus having a lower risk of saturating the network. This is very well suited for dense wireless networks however, as the network size increases, so will the control packets necessary for maintaining the routing information.

Better Approach to Mobile ad hoc Networking

Better Approach to Mobile ad hoc Networking [44] (B.A.T.M.A.N.) is a proactive routing protocol similar to OLSR although, it tries to improve on the shortcomings of OLSR (e.g., maintenance of routing tables, bigger control messages). As such, B.A.T.M.A.N. relies solely on announcements that provide routing information.

Each node periodically announces a packet containing it's identifier (IP address), the identifier of the node who sent it (at the start, the originator node), and a sequence number to overrule older announcements. When a node receives an announcement, it changes the sender identifier to its own, and re-transmits the announcement. Nodes count how many announcements have been received from each node by each link (direct neighbour) in sliding windows, that move based on the sequence number. The number of announcements that fall into the windows is then used to determine the best (probable) next-hop for forwarding messages towards that node. The rationale is that windows containing lower numbers of gaps denote more stable paths in the network.

The algorithm presents no more control messages than the announcement messages that are kept with constant size. Nevertheless, the amount of effective retransmitted messages can grow uncontrollably when the number of devices grows. Additionally, if all nodes transmit their announcements simultaneously a large number of collisions can happen.

Greedy Perimeter Stateless Routing

Greedy Perimeter Stateless Routing [47] (GPSR) is a reactive routing protocol that uses the positioning of nodes to determine where to forward a packet. However, in order to obtain the position of neighbouring nodes, GPSR relies on a proactive mechanism that has nodes periodically (locally) announce their positioning.

The protocol combines two techniques in order to establish routes between nodes: *greedy* routing and *perimeter* routing. Greedy routing has nodes forward packets to the node that is closest to the destination node. When a greedy decision cannot be made (i.e., there is no (known) neighbour closer to the destination), GPSR resorts to perimeter routing, where the node will make an heuristic-based decision to forward the packet around the perimeter of the area that lies between the current node and the destination (that is unoccupied by other devices).

The protocol introduces minimal control overhead of routing packets. However, nodes are required to have a way of determining their position (e.g., GPS or other means) which might be unfeasible in some scenarios (e.g., sensor networks).

2.4.2 Discussion

Routing protocols pose a very interesting case study, as nodes themselves have to perform routing without the help of specialised hardware (i.e., routers, switches). Routing protocols provide the ability of communicating with any arbitrary node in the system, which can be a beneficial abstraction to applications and other protocols, .

In this work we intend develop practical solutions for wireless ad hoc networks. In that regard, we need a way to efficiently develop and execute distributed protocols for wireless ad hoc settings. Consequently, next we study frameworks and tools for efficiently develop correct and efficient implementations in these scenarios.

2.5 Frameworks for building Distributed Protocols & Applications

Implementing a distributed protocol can be a arduous and time consuming task. As such, efficient development and execution tools for distributed protocols have been proposed in the past. Here, we present some of the most prevalent ones however, and as it will be discussed, these frameworks and toolkits have been developed for different purposes than the one considered in this work (e.g., group communication or sensor networks).

2.5.1 Isis and Horus

Isis [16–18] and Horus [83] are both toolkits that provide abstractions for programmers aiming at building distributed applications for clusters of computers using group communication [85].

Both Isis and Horus are protocol kernels, where each protocol is composed by a set of layers that are implemented by micro-protocols. When a message is sent or received it must pass through all layers.

These layers can be arranged in any order as long as they respects their (individual) dependencies. Each protocol must implement interfaces that support the upwards and downwards interactions with other protocols.

Isis focus lies on providing fault-tolerance mechanisms to the programmer however, Isis assumes a model where nodes can crash but no network partitions can occur. Horus, on the other hand, builds further on the Isis programming model to support network partitions, while focusing on important group communication primitives, such as membership management, message ordering, view synchrony, among others. Furthermore, Horus also implements special protocols that allow messages to bypass layers within the stack in order to improve performance.

Both of these frameworks are implemented in different languages, namely C and C++, and support low level programming. Although, as their focus lies on complex and costly group communication abstractions, there is a lack of support for wireless network primitives.

2.5.2 APPIA

Appia [63] is a protocol kernel that is tailored for building Quality of Service (QoS) protocols by combining micro-protocols to support applications that need to use different communication channels (e.g., multimedia applications). The QoS can be viewed as a stack of layers. Each layer is implemented by a micro-protocol in which, each instance is termed a session.

The micro-protocols are event based, and interact with each other through events. Appia implements a single threaded event scheduler that handles all events in the Appia system. This event scheduler is responsible for handling and delivering each event to the micro-protocols that should process the event, respecting their order in the stack.

Appia is implemented in Java and as such, leverages on the inheritance mechanisms of Java Objects to allow programmers to customise their protocols and events easily just by extending top level objects defined by the Appia framework.

However, Appia was developed to support view synchrony and group communication protocols hence, it considers a wired environment and does not support wireless primitives, such as one hop broadcast. Furthermore, Java has no support for low level network primitives, as its network primitives consider TCP and UDP traffic only, which are not suited to build protocols and applications specifically for wireless ad hoc networks.

2.5.3 TinyOS

TinyOS [55] is a lightweight operating system designed to provide tools and abstractions for programmers building sensor network applications. It is based on a set of reusable components that fit together to support some specific application. The programming model is event driven and is based on split phases, asynchronous *events*, and computational *tasks*.

Each application built in TinyOS is composed by a set of *components wired* together. A component defines interfaces that provide three abstraction: *command*, *event*, and *task*. Commands are explicit requests for a component, events are asynchronous responses to

commands or other interrupts (e.g., hardware signals, message arrivals), and task are computational tasks triggered by either commands or events. In other words, commands and events are inter-components interactions , while tasks are intra-component interactions.

All these abstractions are processed asynchronously and handled by TinyOS's event scheduler, that by default implements a FIFO policy. These components interact with each other by wiring them together, which requires providing the complete set of components that an application uses at compile time.

Furthermore, TinyOS already provides several components that include abstractions for sensors, single hop networking, ad hoc routing, power management, timers, and non-volatile storage.

When compiling a TinyOS program, the binary generated will be a single process application that runs on a few specialised hardware, which makes sense in a sensor perspective, since sensors have limited resources and usually run a single application. However, if we consider commodity hardware that runs a Linux based operating systems for example, this is not ideal as we may want to have a set of services or applications running independently for different purposes.

2.5.4 Impala

Impala [56] is a middleware system that is intended to act as a lightweight operating system for sensors. The key concern of Impala is the ease of deployment and updates of sensor network applications thus, it privileges modular applications. Moreover, Impala also presents another interesting feature, which is the ability to change application on the fly adapting to the conditions of the device or the environment.

The applications are built by a set of protocols that are event driven. Each protocol is implement as a set of event handlers. Impala further offers a user library containing network utilities, timer utilities, and device utilities.

Impala is able to have multiple application loaded however, only one can be active at each time. All applications share the same storage, and as such, must agree upon the basic storage organisation (i.e., their data model).

To manage applications and the reception of events, Impala is composed by three main components: the *application adapter*, the *application updater*, and the *event filter*.

The application adapter is responsible to change between applications and manage their life cycle. The application updater is responsible for keeping track of the versions of the modules used by the applications in order to perform remote updates.

The event filter captures and dispatches events between the device and the application. Impala defines five types of events: *i)* ***Timer***, which signal timers that have expired; *ii)* ***Packet***, that represents network messages; *iii)* ***Send Done***, is a notification about a successful, or unsuccessful, transmitted packet; *iv)* ***Data***, represents new information sensed by sensors; and, *v)* ***Device*** that signals a device failure.

Impala was implemented and tested in a Linux based system, with the objective of later being ported to sensors in ZebraNet [57]. However, Impala presents a limitative execution model as it only supports the execution of a single process at once.

2.5.5 Discussion

As we have discussed in this Section, some of the existing frameworks do not address or take into consideration the wireless setting, like Isis, Horus, and Appia. Others, like TinyOS and Impala, are too specific to concrete scenarios. There is also more recent work on frameworks for IoT applications [58, 86], but these do not consider wireless ad hoc networks as a possible means of communication, since interactions are mainly considered to be held between IoT devices and either cloud or fog servers (through infrastructure networks).

We notice therefore, that there is a gap in the available frameworks to build distributed protocols. This limitation is addressed by the first contribution of this work, a framework specially tailored for the development and execution of distributed protocols and applications for wireless ad hoc networks, named Yggdrasil. We detail the design and implementation of Yggdrasil further ahead in the next Chapter of the thesis.

While tools for developing and executing distributed protocols in wireless ad hoc settings are indispensable, we also study how we can perform computations in these settings. For this, we continue the Chapter by presenting distributed communication strategies that are fundamental abstractions for performing distributed computations; we further discuss distributed data aggregation as a form of distributed computing, and focus on building efficient and decentralised topologies for efficient and reliable distributed data aggregation.

2.6 Decentralised Communication Strategies

A basic requirement to perform distributed computations is the ability to communicate and exchange information with other nodes, to either coordinate, or transfer relevant data. As previously discussed, we target highly decentralised systems where nodes do not have access to infrastructure and hence, have to communicate using decentralised approaches.

We now study some of these mechanisms, that can broadly be characterised as being deterministic or random in relation to the patterns of messages exchanged among a set of nodes in a network.

2.6.1 Deterministic Communication Patterns

A deterministic communication pattern has nodes communicate in a predefined pattern, in other words, all messages transmitted by a node will take the same path in the network, to reach their destination. We can define two main deterministic communication patterns: *flooding/broadcast* and *topology dependent*.

Flooding/Broadcast

In a flooding/broadcast [54] pattern, a message sent by a node has, usually[1] the objective of reaching all nodes. In this approach, nodes send the message to all their neighbours and, receivers of that message, keep forwarding it, repeating the same pattern.

A simple materialisation of this communication pattern is an epidemic broadcast protocol, that operates in the following way. When a node has a message (typically generated by another protocol or an application) to disseminate, it sends the message to all nodes that it can reach. Whenever a node receives a message for the first time, it repeats this behaviour by retransmitting the message. Otherwise, if the received message is a duplicate, the protocol simply ignores it.

Topology Dependent

A topology dependent [41] pattern has nodes leveraging an underlying (logical) topology to propagate their messages to a particular element in the network (such as a sink node), with the objective of minimising the communication cost.

Frequently used topologies include: *i)* ***single-path tree***, where nodes are arranged in a tree, and information can easily flow towards the root node, or be broadcasted downstream to all or a subset (i.e., branch) of elements in the tree; *ii)* ***multi-path tree*** is similar to the previous one with the exception that it contains redundant paths along the tree; *iii)* ***ring*** topology where nodes are arranged in a ring, allowing simple communication to relative portions of the ring; and finally *iv)* ***cluster-based*** topologies, where nodes are organised into clusters, which has nodes within a cluster communicating frequently, whilst communication between clusters is performed at a higher hierarchical level (i.e., cluster representative nodes communicating with each other).

2.6.2 Random Communication Patterns

Random communication patterns are, as the name implies, patterns that have nodes exchanging messages through random local decisions. There are two main random patterns: *gossip-based patterns* and *random walks*[2], that we now briefly define.

Gossip-Based Patterns

Gossip-based [51] patterns rely on having nodes exchange messages by selecting subsets of other nodes in direct reach at random, usually in a per message basis.

The effective size of the subset selected for transmitting a message depends on the goal of the communication step, as well as from operational aspects of the system deployment,

[1] There are flooding techniques with limited horizon, where a message will be forwarded only for a pre-defined number of hops.

[2] Random walks are a particular case of gossip-based patterns, we however, give special emphasis to it due to its frequent use in multiple protocols and distributed scenarios.

such as the total number of nodes in the system or the number of reachable nodes in relation to the node that is propagating a message.

The exchange of information among nodes usually follows a pair-wise model (where two nodes interact directly), and can be implemented using the following strategies: *i)* in ***push*** strategies, nodes that have relevant information immediately send it to the subset of receivers selected at random; in contrast, *ii)* ***pull*** strategies have nodes selecting random subsets of other nodes in their vicinity, and ask for relevant information through a pull request. Nodes that have relevant information can reply to these requests directly; finally, there are *iii)* ***hybrid*** approaches that combine the two previous ones, usually using push to quickly disseminate relevant information, and pull to recover lost updates.

Random Walks

Random walks [62] consists of a node choosing a random neighbour to propagate a message. This neighbour will forward the message following the same pattern (picking another random neighbour). This process continues until some criteria is met (e.g., maximum number of hops, reach target node or originator).

This mechanism allows messages to transverse the network through a random path, resulting in a low communication overhead. Additionally, it is well suited to obtain random samples of information distributed across many nodes.

There are also variants of this approach where the selection of the node to whom to forward the message is not entirely random, but biased by some application specific criteria [13, 88].

2.6.3 Discussion

The decentralised communication patterns discussed in this Section are tightly related to how protocols exchange data to perform their (distributed) computations. In the following Section, we present data aggregation as a form of distributed computation, in which we study several protocols that use these communication patterns to compute an aggregation function in a decentralised and distributed fashion.

2.7 Aggregation

While it is important to support general purpose computations in the edge, in the context of this thesis we focus on building aggregation protocols at the edge. Aggregation protocols can serve as a building block for more general purpose computations. Aggregation can also help in reducing the amount of communication among nodes and can be used to monitor the network's state. However, given the distributed nature of edge computing scenarios in general, and wireless ad hoc networks in particular, we must carefully evaluate the properties of distributed aggregation computations.

Aggregation computations can be defined as computing an aggregation function [41] over a set of input values, where each device (or node) holds one of these values. Traditional aggregation functions such as COUNT, SUM, AVERAGE, MIN, and MAX, present different properties which we must consider to understand the challenges of computing such functions in distributed environments. To this end, we will consider the definitions given in [41], which presents aggregation functions as having two properties: *Decomposability* and *Duplicate Sensitiveness*.

Decomposability: Decomposability is a property of an aggregation function that can be defined by composing other functions. However, we can distinguish functions that are self-decomposable and decomposable. Self-decomposable functions are functions that have commutative and associative properties, much like MIN, MAX, SUM, and COUNT. In more detail, these functions can be computed by employing recursive strategies.

Decomposable functions are functions that can be composed by applying some function to a self-decomposable function. The AVERAGE is an example of a decomposable function, which can be obtained by having pairs of values $(x, 1)$ where x is the input value and 1 is the count of values, summing these pairs as $(x + y, 1 + 1)$, and finally, computing the division of both computed sums.

Duplicate Sensitiveness: An aggregation functions is considered to be duplicate sensitive if its result is sensitive to the presence of duplicates. For example, SUM and COUNT will output unfaithful values when duplicates are present, while MIN and MAX will output the same values regardless of duplicate values. In other words, this depends on the function being idempotent or not.

In this work we go a step further regarding the computation of aggregation functions. Normally, when considering distributed aggregation, the input values are considered to be static. What this means is that each node holds one input value that will remain unchanged for the rest of the system's operation. This is highly unrealistic in scenarios where an aggregation protocol is being leveraged, for instance, for monitoring systems. Hence, in this work we not only consider distributed aggregation, but also *distributed continuous aggregation*.

In the continuous aggregation problem, each node holds an input value that can change over time in an independent fashion. A distributed continuous aggregation algorithm must find a way to incorporate the changes of input values in the aggregation result. Furthermore we also argue that the aggregation result should be computed by each node in the system. This way, all nodes will have a summary of global information that can then be leveraged by applications, or other protocols, to make local decisions regarding their operation.

This brings additional challenges when computing aggregation functions in a distributed manner. Where MIN and MAX seemed easier at first given their duplicate

insensitive nature, they are as complex as calculating the SUM, COUNT, and AVERAGE, in this particular context. This is due to fact that when values cease to exists, they must be invalidated through some (distributed) mechanism.

We identify two approaches of performing distributed continuous aggregation. The first is through the use of *epochs*, the second is to have a *natural continuous* aggregation algorithm that is able to cope with the input value changes on the fly.

Epochs: Using epochs is the easiest way to deal with input values changes. This approach is based on having the distributed aggregation protocol restart periodically, where each period is considered to be an epoch. However, identifying the optimal epoch period is not an easy task [49], as this depends on the system size and on the frequency at which values changes, which might be unknown and dynamic parameters of the system.

Natural Continuous: Designing a natural continuous aggregation algorithm is a more challenging task. A natural continuous aggregation algorithm must be able to continuously recalculate the aggregation result without the need to be restarted. In other words, the algorithm must seamlessly incorporate input value changes, adding new input values and removing old ones from the aggregation result as part of its regular operation.

2.7.1 Aggregation Computational Schemes

There are different schemes that can be employed to compute an aggregation function in a distributed fashion. Each scheme has its own advantages and drawbacks. Some schemes try to reach the exact aggregation result by controlling the employed communication pattern. Other schemes focus on reaching approximated estimations on the aggregation result in favour of simpler communication management or obtaining additional information regarding the distribution of input values across the system.

In short, we can define two main categories of techniques to compute aggregation functions, the ones that reach exact values, and the ones that calculate approximations and estimations.

2.7.1.1 Exact Value Computations

Exact value computations strive to obtain the exact value of the aggregation function that is being computed. In that regard, every input value present in the network must be taken into account. As a consequence, all nodes within the system must transmit their values at least one time (as long as there are no message losses).

Therefore, we describe two techniques that reach the exact value: *Full Dissemination* and *Hierarchical Aggregation*.

Full Dissemination Full dissemination techniques is a simple approach to compute distributed aggregation functions. A node in the system floods the network with a request for input values, and every other node responds with their own input value. The computation is then performed in the initiator node.

Since every node is disseminating their input values, any aggregation function can be computed at each node. However, this approach will cause a high message overhead in the network, which can easily lead to its saturation.

Hierarchical Aggregation In hierarchical aggregation techniques, nodes are arranged in a logical hierarchical topology (typically a tree), where the top of the hierarchy can be viewed as a special node (*sink* node). This node is responsible for building the hierarchy; after this, the nodes that are at the bottom of the hierarchy will send their values to upper levels. Upper level nodes, compute an intermediate aggregate with the values received from lower levels, and pass the result to upper levels. This process continues until it reaches the sink node.

These techniques, reduce the amount of messages transmitted in the network, as messages pass through pre-defined paths. Nevertheless, the cost of maintaining the hierarchy might be high, and on the presence of faults, hierarchical aggregation can easily reach unfaithful results.

2.7.1.2 Estimation Computations

Estimation computations rely on probabilistic methods to reach the approximate result of an aggregation function. They focus on reducing message size, reducing communication, and in some cases, ensuring fault tolerance. We describe three main estimation computation techniques: *Sampling*, *Data Representation Computation*, and *Iterative Approaches*.

Sampling Sampling techniques tend to focus on probabilistic counting techniques to determine an estimation (usually to determine the network size). These techniques commonly use random walks, or random selection of nodes within the system, to collect samples of input values that can be used to infer or compute an approximation of the global aggregation result.

However, these techniques rely on probabilistic methods, and as such, they can easily incur in situations where the sampling is biased. This can happen due to an imbalance in input values for the aggregation across the network.

Data Representation Computation Data representation computation relies on auxiliary data structures to compute estimates. These data structures represent the values in the system, although the representation of values differs from protocol to protocol. Some protocols focus on representing values in an histogram to answer more complex aggregation functions (e.g., median, mode, among others), while others represent values through bit masks as a way of compressing them, or in a subset of representative values.

Nevertheless, these techniques require estimation functions to extract the aggregation result from the data structures. Furthermore, the compression techniques used are typically resource heavy, and might incur in high computational overheads that may be unacceptable in some cases (e.g., sensor networks).

Iterative Approaches Iterative approaches exploit mathematical properties of aggregation functions to compute estimations. These approaches usually rely on the principle of "mass conservation", which dictates that as long as the sum of the aggregated values in the network remain constant, it is possible to reach the correct approximation [48]. Consequently, nodes exchange information in order to compute partial aggregate results. The more exchanges happen, the more approximate the result will be to the exact value.

However, these techniques can be very sensitive to duplicate messages and message loss, since it translates to gain or loss of "mass" which will violate the principle of "mass conservation".

2.7.2 Relevant Aggregation Protocols

We now present some of the exiting distributed aggregation algorithms that illustrate the classes of solutions discussed above. Some of these solutions will be used in our experimental work for accessing the benefits of our own proposal, MiRAge.

Tiny Aggregation

Tiny Aggregation [59] (TAG) was developed to support aggregation queries in wireless sensor networks using TinyOS [55]. It performs computations using an hierarchical approach on top of a topology dependent communication pattern and provides an SQL-like query language. As such, it provides basic aggregation functions such as Count, Sum, Average, Min, and Max. TAG also allows queries to use the Group By operator, enabling aggregation functions to be captured over subsets of input values given certain node properties (e.g., the total sum of all input values of nodes provided their geographical area).

The protocol is composed of two phases: *distribution* and *collection*. The distribution phase consists in creating the routing topology (typically tree-based) from the *sink* node (root node) to all other devices. To do this, the root node broadcasts a query request message; every node within range that receives this message will mark the message originator node as its parent and adjust their radio wake up interval according to the parent node. Nodes keep forwarding the aggregation request until all nodes are assigned a parent.

In the collection phase, each parent node must wait for the values of its children. Although, since the children have adjusted their wake up interval with the parent, the parent node expects that all of its children report their values within that interval.

When the parent node receives the values from all its children nodes, it computes a partial aggregate result with its own value an forwards the partial result to its parent node. This process continues until the root node receives replies from all its children.

Given TAG's computation pattern, the final result of the aggregation is only computed in the root node. If the result is relevant to the other nodes in the system, the root node must broadcast it throughout the network. TAG is not a natural continuous algorithm, as such, it must resort to execute in epochs to solve the continuous aggregation problem.

Regarding fault tolerance, a single fault could disconnect a large portion of the network and incur in gross aggregation errors. TAG tries to mitigate these problems, by having each node keep a list of neighbours allowing them to switch parents, snooping messages by utilising the wireless medium, and nodes keeping cached values of their children's previously reported values that can be used if no reply is received from one of the children nodes. However, the authors do not consider the fault of the sink node, which renders the protocol incapable of operating.

Directed Acyclic Graph

Similarly to TAG, Directed Acyclic Graph [67] (DAG) is an aggregation protocol that relies on an hierarchical topology to perform hierarchical computations. Although, the routing topology offers multiple paths among nodes instead of a single path. It support the same aggregation functions as TAG, with the exception of the Group By functionality, since it does not offer an SQL-like interface.

When an aggregation request is received, the sink node broadcasts it to the network. The aggregation request performs similarly to the one described in TAG however, adding a list of parents to the message. This list allows for child nodes to choose a grandparent, where their input value will be used to compute a partial result thus, allowing nodes to have multiple parents.

The key difference between DAG and TAG, is the fact that the topology contains redundant paths to the root. As such, the final aggregation result is again only computed in the root node, and must be broadcasted across the network if it is relevant for the rest of the nodes in the system. Similarly, DAG must run in epochs to solve the continuous aggregation problem.

The multi-path routing tree of DAG brings some advantages in regard to fault tolerance within the network. As input values are propagated through multiple paths, the loss of a single value through a path does not affect the computation of the aggregation result. However, this comes at the cost of larger messages, the failure of a grandparent node can still lead to gross aggregation errors, and, similarly to TAG, the failure of the sink node renders the protocol unable to proceed.

Generic Aggregation Protocol

The Generic Aggregation Protocol [23] is another algorithm that relies on an hierarchical (tree) topology to perform (hierarchical) aggregation. However, contrary to TAG and DAG, the management of the topology in GAP happens naturally with the exchange of values among nodes, instead of needing an explicit broadcast from the root node to build the tree topology. GAP can compute common aggregation functions as the COUNT, SUM, AVERAGE, MIN, and MAX. We discuss the operation of this protocol in more detail, as this is the closest solution to the MiRAge protocol that we propose in Chapter 4.

In GAP, instead of receiving an explicit aggregation request to begin establishing the tree and compute the aggregation result, nodes start by exchanging information. However, as we detail further ahead, the tree still needs to have a fixed and pre-defined root to enable the formation of the tree topology. The process of building the tree is governed by a parameter maintained by each node called its *level*, which is initially set to an arbitrary large value. Furthermore, each node maintains additional information concerning their neighbouring nodes, with whom they exchange information. The maintained information contains, for each neighbour, its current level on the tree; its relative relation with the local node in the tree, that can either be `PARENT`, `CHILD`, or `PEER`; and the aggregated value. A node also maintains information regarding itself, being similar to the one maintained for each neighbour, having a relation of `SELF`, and its aggregated value being its input value.

To build the tree topology, a root must be appointed. This is done by adding a virtual neighbour to the appointed root. This virtual node has a constant level of -1 and becomes the parent node of the root node. The level of a node is calculated by considering the level of its parent node (the parent level plus one) therefore, the root will have a level of zero.

Messages exchanged by the GAP protocol are called updated vectors. An update vector contains the node's identifier, its current level, its current aggregated value so far, and the identifier of the parent node.

When a node first discovers a neighbouring node (through the help of an underlying discovery protocol), it sends the update vector to the newly found node (using a point-to-point message). A node that receives an update vector from another node, first updates the information regarding the sender node. For that, the local node updates the level and the aggregated value of the sender node. The local node must also update its relative relation to the sender node in the tree. If the sender node has the local node marked as its parent (information enclosed in the received message), the local node changes the sender's node relation to `CHILD`. If the sender node does not mark the local node as its parent, but the previous known relation by local node was `CHILD`, the sender node is marked as being a `PEER`. If none of the above cases hold, the relation remains unchanged. Note that when two nodes discover each other for the first time their relation is set to `PEER`.

After updating the local information, a node runs a stabilisation mechanism. This aims to enforce a set of invariants that collectively ensure the construction and maintenance of the tree topology. The invariants include: only one node can be marked as PARENT, the PARENT node must have the lowest level among all other known neighbours, and the local node must have the level of its parent plus one. In other words, this stabilisation allows a node to choose a correct parent, and adjust its level accordingly.

After this, a node computes a new update vector that reflects all changes that might have happened to its local state. For this, the node calculates the aggregation function with its own aggregated value (its input value) and the aggregated values received from the neighbours marked as CHILD. If the update vector differs in some way to the previous one, it is sent to every neighbour, otherwise the node does not transmit the update vector.

As the algorithm progresses, the first nodes to have a parent will be the neighbours of the root node, as this is the one with the lowest level in the system. The root's neighbouring nodes will propagate the level change, causing their neighbours to choose appropriate parents. Nodes will calculate the aggregation function based on their input value and the values received from their children, which will be propagated towards the root node hence, calculating the aggregation result. A node stops sending messages when there are no local modifications to its state.

In GAP, only the root is capable of calculating the final aggregation result, and as in TAG and DAG, this result must be propagated across the network, if one needs the rest of the nodes in the systems to become aware of it. However, GAP was designed to support continuous aggregation in a natural way. This is because a message is always sent if it differs from the previously sent message, and since a node in GAP calculates the aggregation function based on its own input value, it means that when the input value is changed, so will the resulting (local) calculation of the aggregation function hence, leading to the propagation of the change throughout the system.

The implication for this, is that any event that results in a variation visible in the result of the aggregation function being computed (i.e., the value of the local node or a neighbour changes, a new neighbour is discovered, or a fault happens) leads a node to propagate that change naturally.

In addition, whenever a node detects the failure of a neighbouring node, it runs the stabilisation mechanisms to ensure the tree topology is correct. However, when the root node fails ensuring the correctness of the topology becomes impossible, since no other node will be able to have the lowest level in the system with a valid parent.

Push-Sum Protocol

The Push-Sum protocol presented in [48], relies on gossip communication combined with an iterative approach to compute aggregation functions, such as Average, Sum, and Count. The protocol operates by having nodes exchange messages with their neighbours in a pair-wise fashion.

In Push-Sum, each node contains a pair (v_i, w_i), where v_i is the local value of node i, and w_i is an additional parameter associated to the value, called weight. The initial local value of a node is its input value and the initial weigh depends on the aggregation function to be computed. These local values are propagated through the network via message exchange in the following way:

A node splits its local value and weight in half, sends one half to a randomly chosen neighbour, and keeps the other. When a node receives a message it simply adds the received value and weight to its local state. An estimation of the aggregation result value can be calculated at any given time by dividing the local value and weight (v_i/w_i).

As more iterations of the protocol are performed, the approximation being computed becomes more accurate. To calculate different aggregation functions the protocol allows the manipulation of the initial weights of values used by each node. For AVERAGE the weights are set to one in all nodes; for SUM, the initiator node, has weight one, while all the other nodes have weight set to zero. COUNT differs from SUM by having the value set to one in all nodes (maintaining the weight set to one in the initiator and zero in all other nodes).

In Push-Sum all nodes compute the aggregation result however, it is uncertain when has a node computed the final aggregation result. Furthermore, as the initial input value is lost during the execution of the protocol, the protocol must run in epochs in the case of input value changes to solve the continuous aggregation problem.

Finally, the protocol is only correct if the principle of "mass conservation" is kept thus, the links between nodes must be reliable (i.e., messages cannot be lost or duplicated). Regarding changes in the system's filiation (i.e., entry/failure of nodes), Push-Sum is able to cope with new nodes joining the system, as it only means that nodes have another neighbour with whom they can exchange messages. However, when a node fails, Push-Sum is unable to recover the values that where transferred to the failed node, and remove the values transferred by the failed node from the system.

LiMoSense

The LiMoSense [30] protocol is a fault-tolerant variant of the Push-Sum protocol. LiMoSense tries to overcome the limitations present in the Push-Sum protocol by storing the accumulation of the transferred values between each pair of (communicating) nodes. A node i maintains information about the total values transferred and received by each neighbouring node j. Another key difference of the LiMoSense protocol, in relation to Push-Sum, is that instead of each node maintaining a pair of value and weight, it maintains a pair of the estimate of the aggregation function and the weight of that estimate (est_i, w_i). Due to this transformation, and how the protocol handles it, LiMoSense can only be used to calculate the AVERAGE aggregation function.

When a message is to be sent to a node j by node i, i starts by adding the values (its current estimate and half of its current weight) to be transferred to the total amount of

values previously transferred to j. The addition is calculated by a weighted sum. The resulting new total is stored locally, and sent to j.

When j receives the message sent by i, j calculates the difference between the locally stored total received value from i and the recently received value by i. The difference is calculated similarly to the addition of values, with one of the parcels having a negative weight. The difference is incorporated in j's local estimation and weight, and the received message from i is stored as being the new total received value from i.

As the protocol progresses the estimation of the aggregation value will become more accurate as more messages are exchanged. As in Push-Sum the estimation will be computed by every node in the system. Moreover, in LiMoSense the input value is stored independently and thus, it is not lost during the protocol's execution. The input value is then used when it changes value, to calculate the new estimate, meaning that LiMoSense supports continuous aggregation naturally.

Because each node maintains the total values transferred and received, the protocol is able to recuperate "mass" in the presence of faults. When a message is lost, the next message will contain the lost values. However, the stored values can grow with no bound. To deal with this issue, LiMoSense also employs a mechanism based on binary serial numbers to reset stored data.

When a node leaves a neighbourhood, either by a link fault or a crash, the nodes that detect this change must compensate the lost input value. To do so, they first add the total amount of value sent to the faulty node to their own local values, then they add the total amount of received value from that node to a special register. The value in this special register will be lazily removed from the local node's local values when sending messages, effectively removing the values transferred from the faulty node from the system.

Distributed Random Grouping

Distributed Random Grouping [21] (DRG), is an aggregation protocol tailored for wireless sensor networks that leverages the broadcast nature of the wireless medium to create random local groups that calculate local aggregates. The algorithm allows to calculate Average, Max, and Min, but does not specify how other functions could be calculated. The algorithm follows iterative steps, until the aggregate value converges. In each step, nodes can be presented as having three states: *idle*, *member*, and *leader*.

A step of the protocol consists of an idle node choosing with some probability p to become a leader of a group. When one does, it sends a message to all reachable neighbours to notify them that it is the leader of a group. All other idle nodes that receive this message send an acknowledgement message (JACK) to the leader node containing their local aggregated value (at the beginning their input value). Once the leader receives the JACK from its neighbours, it calculates the aggregated value with its own value, and broadcasts again to its neighbours the aggregated value thus, updating the local group currently computed aggregated value.

In order to converge, groups must overlap overtime propagating the previously computed aggregate from one group to another hence, having every node in the system compute a valid estimation of the final aggregation result. However, the algorithm does not consider the changing of input values, as such, to solve the continuous aggregation problem it must resort to epochs.

The algorithm is not fault-tolerant, as it does not consider the implications of the failure of a node. Furthermore, the loss of a JACK message or an updating message, implies "mass" loss, which leads to the computation of incorrect aggregation values.

Flow Updating

Flow Updating [42, 43] performs gossip-based communication using an iterative approach based on the concept of *flows* from graph theory, where each flow is a representation of a differential between the approximation computed by two nodes. We define the flow from node i to j as f_{ij}, and the flow from j to i as f_{ji}. Flows have a symmetry property, and as such, $f_{ij} = -f_{ji}$. Flow Updating leverages this property to calculate estimations about the real aggregation result, while in some cases obtaining the exact result. It is tailored to compute the AVERAGE aggregation function.

Instead of exchanging "mass" like Push-Sum, LiMoSense, or DRG, Flow Updating has nodes exchange flows and estimates which are based on the original input value of each node that is kept unaltered. With this, the protocol is able to recover "mass" in the event of message losses.

At the start of the algorithm, a node sets the flows and estimates of its neighbours to zero, as it has no knowledge about them. Given this, it calculates its local estimate by subtracting all flow values to its original value. It further computes the flows for each neighbour by updating the currently held flow to a neighbour with the difference between the previously calculated local estimate and the previously held local estimate calculated by that neighbour (at the start zero). Then, the nodes send to each of their neighbours a message containing the flow computed to that neighbour and the calculated local estimate. When a node receives these messages, it updates the flow to the originator by storing the symmetric of the received flow ($f_{ij} = -f_{ji}$) and the received estimate.

Periodically (after receiving the updates), each node can recalculate the local estimate with the updated values for the flows of each neighbour and the estimate values received, and recompute the flows to each neighbour. Nodes exchange this information indefinitely, obtaining the aggregated result in all nodes eventually. Every calculation performed by the algorithm is based on the original input value, if this changes, the algorithm starts to consider the new input value causing a change in the computed flows and estimates, which will be propagated through the system. Given this, Flow Updating is an algorithm that support continuous aggregation naturally.

As in LiMoSense, this protocol is able to sustain message loss, without incurring in permanent errors in the computed aggregated result. When a node fails, all neighbours

of the faulty node stop considering its flow and estimation, causing the calculation of the new estimation to change. This change will be propagated through the network causing the contribution of the faulty node to be forgotten.

Extrema Propagation

Extrema Propagation [14], combines a data representation computation technique with a gossip communication strategy. With this, Extrema Propagation is able to compute an estimation of the real aggregation result of the SUM aggregation function. Each node produces a vector with k random numbers that follow a known distribution (e.g., Exponential) and exchange this vector with their direct neighbours.

When a node receives a vector, it calculates the pointwise minimum[3] of its local vector and the received one, keeping the result as its local vector. Each node proceeds by exchanging vectors until there are no changes for T rounds (T is a configurable parameter). When the algorithm terminates, each node will have the same vector containing the minimum numbers in the system for each position in the vector. The algorithm terminates by calculating the maximum likelihood of the minimum vector, obtaining an estimation about the aggregate result based on the mathematical properties of the initial distribution of values generated by nodes.

An interesting aspect in this protocol, is that the error of the estimation depends only on the size of the exchanged vector. The bigger the vector, the lower the error bound. This could come as an advantage in very large systems, where messages need to be kept small. Nevertheless, in smaller systems it is hard to justify the effort of generating random numbers that follow a known distribution, to obtain an approximate result, when the input values of all nodes fit in a single message (that is relatively small) and, contrary to this solution, able to reach an exact result.

Q-Digest

Q-Digest [76] leverages on a data representation computation mechanism to provide the ability to compute more complex aggregation functions as the median and mode. Each node computes a data structure, named quantile digests (q-digests) and propagates these data structures (in a compact form) along a spanning tree to reach a sink node. A q-digest is a data structure that consists of a set of buckets that represent the frequency of values within a range (i.e., an histogram of classes of values).

Each node contains a set of values within some range and the frequency of their values. The objective of the q-digest is to compress this data. To do so, these values are represented in a binary data tree. Each leaf node represents a value within a range. Values flow up the tree when their frequency is not representative enough, following properties that relate to the total count of values in a sub-tree to a compression level.

[3] The minimum number at each position of a set or function.

In other words, one could see each step of the tree as a bucket of values, the higher it is in the tree, the more values are represented by the bucket. The q-digest, once computed, will be composed by the set of buckets that are most representative (given the total count of values).

The q-digests are propagated through the spanning tree where parent nodes are tasked to merge the received q-digests with their own. This is done by simply adding the counts of the received buckets to their own and recomputing the q-digest. The algorithm terminates, when the sink node computes the final q-digest, obtaining a full histogram of the most representative values in the network.

This protocol, by leveraging on the q-digests, presents a compression mechanism that allows to compute more complex queries. Although, q-digests require that there is some knowledge about the values that the network is producing, so that it is possible to build compatible q-digests (that have the same initial range of values) to merge them efficiently, avoiding to generate a single huge bucket that represents all possible individual values. However, this might be impossible to know a priori in some cases.

Randomized Reports

Randomized Reports [15] is a probabilistic polling (sampling) method to determine an estimate of the network size, as such, it only supports the COUNT aggregation function. It relies on a node flooding the network with a request message and setting a timer T.

Each node that receives the message, replies to the originator with some probability p. When T expires, the originator node counts the number of replies and estimates the size of the network by scaling the obtained count by $1/p$.

This is a very simple algorithm that is able to mitigate the effects of flooding messages by not having all nodes respond to the request message. Nevertheless, this algorithm only supports the COUNT aggregation function, and cannot be easily adapted to compute other aggregation functions, such as AVERAGE, where more information is required.

Random Tour

Random Tour [62] consists in utilising random walks to collect samples on the network's size in peer-to-peer networks. An initiator node i, begins the process by sending a sample message to a randomly chosen neighbour containing a tag. This tag contains a counter X with the value $1/d_i$ (where d_i corresponds to the degree[4] of the node i) and the node's identifier. When a neighbour node j receives the sampling message, it increments the counter X by $1/d_j$ and forwards the sampling message to another randomly chosen neighbour. Once the message is received again by the initiator node i, the node estimates the network size by calculating $d_i X$.

[4]Number of neighbours of a node.

The Random Tour algorithm is able to achieve good estimates of the network size, but when the network is very large (e.g., up to 100,000,000 nodes) it will take a long time to reach the estimated value at the cost of a significant number of message exchanges.

Sample & Collide

Sample & Collide [62], similarly to Random Tour, leverages on random walks to perform probabilistic counting to determine an estimate of the network size in peer-to-peer networks. The algorithm is inspired by the "Inverted Birthday Paradox" [15], as it acquires random samples from peers, and then calculates estimates based on how many random samples are required before two samples return to the same peer.

The algorithm executes as follows: First, the initiator node i sets a timer with value T and sends a sample message containing T, to a randomly chosen neighbour. Upon receiving the sampling message, a node j, computes a uniformly distributed random number U between $[0,1]$ and decrements T by $-log(U)/d_i$ (i.e., $T + log(U)/d_i$), where d_i is the degree of node i. Then the node verifies if T is less or equal to zero; if it is, the node is sampled and returns its identifier to the initiator node; otherwise, it will forward the message to a randomly chosen neighbour with an updated timer.

The algorithm terminates when some node has been sampled for a configurable number of times l, and an estimate of the network size is calculated using a Maximum Likelihood method.

The algorithm, when compared to Random Tour, achieves lower accuracy with fewer message exchanges. As such, the algorithm is best suited for larger systems where estimations can have lower accuracy. Nevertheless, the algorithm presents the same key limitation as Randomized Reports, being that it can only compute the COUNT function.

2.7.3 Discussion

We begin by noting that aggregation protocols that use computational schemes that rely on data representation and sampling techniques, have very specific objectives. The algorithms that make use of sampling (and Extrema propagation) are ultimately tailored to determine a good estimate over the network size in a fast and efficient way. Algorithms that leverage on data representation, as Q-Digest, have objectives that usually go beyond calculating an aggregation function, since they are tailored to retrieve more information about the system using effective compression mechanisms. While these are also relevant aggregation problems in distributed systems, they fall out of the scope of this thesis and hence, will not be considered further.

We focus on algorithms that can compute any aggregation function, and that can be leveraged on a real system to perform management decisions, for instance. As such, the considered algorithms of this work must be fault tolerant, being able to cope with message loss, link failures, and node crashes, without affecting the computation of the aggregation function significantly or permanently. Furthermore, input values being used

for the aggregation function, on a real system, may change over time. Requiring the aggregation protocol to be restarted periodically is not a desirable feature, as this may affect negatively any decisions being made regarding the system's operation.

Consequently, the algorithms we consider as valid solutions for the aggregation problem this work focusses on are: GAP, an algorithm that relies on a tree topology to compute any aggregation function; LiMoSense, and Flow Updating, two algorithms that make use of an iterative approach. We further note, that these algorithms base their computations on the original input value, and have mechanisms to deal with the change of input value, which is a requirement to support continuous aggregation naturally.

We also point out that algorithms that rely on a tree topology, such as GAP, have two main drawbacks. The first one is that these algorithms require the root to be preconfigured and static. The second is that, because of the static root, they cannot deal with the fault of the root node, which will render the algorithm unable to perform any correct operation. Therefore, we study previously proposed mechanisms to build and maintain tree topologies in the next Section, which are potentially relevant to overcome this key limitation.

2.8 Self-Managed Overlay Networks

Since we aim at designing systems that are decentralised and have self-management properties, we need an abstraction that simplifies the coordination among devices. Moreover, as presented before, some aggregation protocols utilise communication strategies that rely on concrete topologies. One way to achieve both aspects is to leverage on (logical) *overlay networks*.

Many peer-to-peer systems effectively operate on top of logical networks. These networks are called overlay networks because they operate above another network layer, typically the physical network. Although these networks do not usually consider wireless environments, they provide interesting management mechanisms that should be considered in the context of the work presented here.

2.8.1 Overlay Solutions

There are two main classes of overlay networks: *structured* and *unstructured* overlays.

Structured overlays are logical networks that have a known topology (e.g., a ring [12, 38, 80], a tree [53], among others), on the other hand, unstructured overlays present a random topology. Structured overlays tend to achieve a higher performance for particular objectives (e.g., resource location or application level routing), but usually have high maintenance overheads. In contrast, unstructured overlays have a lower maintenance cost, but might not be as efficient as structured overlays, being more suitable for cooperative dissemination of information.

Overlays are built and maintained by distributed algorithms. These algorithms are associated with membership protocols. However, achieving a global membership (logical connections to all nodes) is a challenge in large scale systems, since maintaining up-to-date information about every node in the systems leads to significant control overhead. Consequently, these protocols rely on partial memberships (connections to logical direct neighbour nodes usually, a subset of all nodes) to achieve global connectivity.

Here we focus on structured overlays that are based on tree topologies, since these are the ones that are commonly used in aggregation protocols to provide efficient routing for values to be propagated, whilst performing computations ensuring no input value duplication.

Plumtree

An effective way of building tree topologies is through the use of flooding, much like how it is done in TAG [59] and DAG [67]. In Plumtree [53] however, instead of the topology being leveraged to aggregate values ensuring that no value is aggregated twice, the topology is used to provide an efficient mechanism to broadcast messages. Plumtree relies on two fundamental gossip communication mechanisms to build and maintain the tree topology, eager push and lazy push. Eager push is used to quickly pass messages, while lazy push is used to recuperate lost messages.

The tree is built by having a node broadcast a message through the network. Nodes that receive duplicate messages answer to those duplicate messages with a negative acknowledgement. This causes the link between those two nodes to become `PASSIVE` meaning that link will only be used to recover messages, rather than to send messages.

When a node perceives that it receives a high number of messages by utilising one of its `PASSIVE` links, it modifies the link to `ACTIVE` meaning that link will again be used to send messages, effectively in some cases repairing the tree naturally.

This however, is a tricky algorithm to be implemented in a wireless setting. This is mainly due to the fact that one wants to leverage the additional communication primitive provided by the medium (one-hop broadcast) to minimise the number of total messages sent, and that Plumtree relies on links between the nodes that are able to be monitored.

Self-Stabilisation Algorithms

Self-stabilisation algorithms while not being directly related to overlay networks, are fundamental for the design of distributed algorithms that build spanning trees over a network [34]. Self-stabilisation algorithms have emerged from the theoretical field, as their main focus is proving that an algorithm can stabilise in a bounded amount of rounds or message exchanges nevertheless, they can still be used to inspire practical solutions. This is the case of GAP, that is presented as a modified version of the breadth-first search algorithm described in [27]. Not all self-stabilisation algorithms have the need of a fixed

and static root, as such, here we present two algorithms that have no such prerequisite and that present some similarities with the MiRAge protocol (presented in Chapter 4).

The Algorithm by Afek, Kutten, and Yung The algorithm described in [3], has every node in the system attempt to construct a tree covering the whole network rooted on itself. For this, each node has a unique identifier, being the node with the largest identifier the root of the tree that will cover the entire network. Nodes have knowledge of their neighbourhoods, meaning that a node knows for each of its neighbour, its identifier, the identifier of the root of the tree to which the neighbour is attached, and its distance to that root. This knowledge is obtained by a process that is outside the scope of the algorithm however, a simple materialisation would be to periodically transmit the required information.

To actually form the tree, nodes must be able to leave their current tree and join a new one. A node checks the information regarding its neighbours, and if it finds that one of its neighbours is in a tree higher than its current tree (the root of the tree of the neighbour is higher than the local node's one), it sends a join request messages to that neighbour. The neighbour will forward this message to root of the tree. The root of the tree will answer with a grant request message that will be sent back to the node that made the request. The node will mark as its parent the neighbour from which it receives the grant message.

The algorithm presents two undesirable features, first it produces a higher overhead with the need to send specific join and grant messages. Second is that, in order to achieve stability (i.e., form the tree), the algorithm is restarted in each node a multitude of times until all nodes are attached to the same tree. This happens because every time a node detects any instability (i.e., some event that forces the local topology to change) through the observation of its local state, it reverts back to its original state of being the root of its own tree and starts over.

The Algorithm by Afek and Bremler In [2] the authors present the notion of *"power supply"* that is used to describe their self-stabilisation algorithm. This notion describes root nodes as sources of "power" that is then propagated through the network. The authors present a synchronous and an asynchronous variant, here we will only focus on the asynchronous one. Similarly to the previous algorithm, each node contains a a unique identifier and a distance to the root, this information is then used to establish a spanning tree whose root is the node with the lowest identifier in the network. Nodes maintain information about their neighbours and the currently known root of the tree.

As the previous algorithm, nodes start as the root of their own tree. Furthermore, the algorithm specifies two types of messages, *strong* and *weak*. Strong messages carry power from a root, while weak messages are exchanged periodically to verify inconsistencies of the local state of neighbouring nodes. For a node to change to a lower root, it must receive enough power from it. Enough is described as receiving two sequential strong messages that contain power from the new root through the same neighbour. The first

message causes an instability in the local node's state, causing it to revert to its initial state, the second message serves as confirmation that the root is still alive and that the node can attach itself to the tree. Consequently, the node marks the neighbour from whom it receives the messages as its parent, and propagates the strong message to its neighbours.

Similar to the previous algorithm, every time a node detects an instability in its local state, either by the reception of a weak or strong message, the node reverts back to its initial state of being the root of its own tree. Furthermore, the need for having two types of messages also introduces additional overhead.

2.8.2 Discussion

Plumtree presents interesting features regarding tree management, as it can easily repair the tree topology during its operation, by simply changing the status of a link. However, the mechanism used by Plumtree to change link status, may rely on too much message exchanges when considering the algorithm for wireless ad hoc settings, which can lead to the contention of the wireless medium.

The stabilisation mechanisms by the other protocols, in contrast, must restart the protocol every time an instability is detected locally but, present two main interesting features. First, every node in the system actively competes to become the root of a tree covering the entire system, which enables the construction of a tree with no fixed or statically configured root. The second is the notion of power that enables other nodes in the system to verify that the root of the (current) tree is still alive without needing to explicitly contact it.

In Chapter 4 we propose a novel distributed aggregation protocol that leverages some of these features in a way that avoids the need to send additional messages to manage the spanning tree that supports the aggregation process.

2.9 Summary

In this Chapter we presented the limitations of cloud computing, which motivated us focus on edge computing scenarios. Hence, we detailed the edge computing paradigm that includes a broad spectrum of different materialisations.

We chose to focus on the particular edge scenario of wireless ad hoc networks in commodity devices, and presented the challenges that are inherent to the wireless medium. We studied existing wireless ad hoc networks, and realised that none is notably suited for efficient edge computing.

We discussed the importance of understanding how protocols operate in wireless ad hoc settings and how to implement and execute them on real devices. Hence, we first discussed distributed protocols in the context of wireless ad hoc networks, namely routing protocols, to understand the key differences of the operation of wired and wireless protocols. We followed this discussion by presenting frameworks for efficiently developing

distributed protocols and applications. We noted that none of the existing frameworks provided the needed support, which leads to proposal of the first contribution of this work, the Yggdrasil Framework (detailed in the next Chapter).

In this work, we focus on enabling computations in these networks. To do so, not only do we need the support to develop and execute these protocols, but also need protocols that effectively leverage the network to perform computations. Therefore, we began by looking into decentralised communication patterns as a fundamental piece of abstraction for distributed protocols that aim at performing computations within the network.

Because supporting generic computations over the network is a daunting task, we studied distributed data aggregation as a simple, yet fundamental, form of distributed computation that leverages the network. We studied several protocols with different properties. We noted that many of the existing protocols do not support networks with dynamic properties, such as the change of input values that each node holds or the instability of the network topology employed, or are unable to reach exact results.

Given this limitations, we looked for efficient mechanisms to build tree topologies that can support the computation of exact aggregation results and that are robust even in the presence of the failure of the root of the tree. This inspired the second contribution of this thesis (described in Chapter 4), the proposal of a novel distributed aggregation protocol name MiRAge.

Chapter 3

The Yggdrasil Framework

To allow the design, validation, and evaluation of distributed protocols for wireless ad hoc networks we must develop correct implementations that can be executed on real devices. In the previous Chapter, we discussed existing tools to implement distributed protocols however, we have shown that none of the existing solutions are adequate to develop distributed protocols for wireless ad hoc settings, particularly when considering their execution on commodity devices, that support the operation of future edge systems.

In this Chapter we propose a novel framework, named Yggdrasil, that provides support for developing and executing distributed protocols and applications in wireless ad hoc settings. We begin by studying properties of distributed applications and protocols, by looking into three different classes of protocols, which motivates the basic functionalities that Yggdrasil must support (Section 3.1). We continue to present the architecture and design of Yggdrasil, and discuss some of the key aspects in Yggdrasil's implementation (Section 3.2). For completeness, and to end the Chapter, we provide an example on how to use Yggdrasil to implement a simple protocol (Section 3.3).

3.1 Distributed Applications

A distributed application is an application that leverages the computational resources of various devices to solve a concrete problem. However, solving a problem in a distributed setting is a daunting task. For this, distributed applications rely on a set of distributed protocols to provide abstractions that aid in solving concrete parts of the problem that the application is trying to solve. In other words, protocols can be viewed as modules or components that are combined and used in the construction of (complex) applications [36].

A distributed protocol can be modelled as a state machine. As such, a protocol is described as having an internal state, that changes in reaction to the occurrence of one

or more events. Hence, protocols contain a set of event handlers that either modify or expose portions of the protocol's state.

From here, we extract two fundamental properties that should be provided by a framework that supports the implementation of distributed protocols. First, it should allow protocols to be built in a modular way with well defined APIs, so that the implementation of a protocol can be leveraged across a variety of applications that require the same abstraction. Second, the implementation of a protocol should follow as close as possible to the specification of the protocol, as to minimise the introduction of undesirable bugs.

3.1.1 Requirements for Supporting Protocols

In the concrete setting of wireless ad hoc networks, protocols should be provided with fundamental abstractions for their operation. To identify these abstraction we study three classes of distributed protocols in the wireless ad hoc setting. These three classes include: a broadcast protocol, a routing protocol, and an aggregation protocol.

In more detail, we study the operation of a broadcast protocol, that follows the decentralised communication strategy based on flooding, and that introduces small random delays in the retransmission of messages (to avoid collisions in the wireless medium); B.A.T.M.A.N. [44], a routing protocol that periodically sends announcements to construct routing tables; and GAP [23], an aggregation protocol that although, not originally proposed for wireless networks poses interesting and relevant properties in this context. We further argue that the aspects found in the operation of these protocols are common aspects found in other distributed protocols.

We now present the set of requirements that a framework such as Yggdrasil must provide. These are motivated by the design and operation of the case study protocols mentioned above.

Communication Abstractions: One-hop Broadcast

Any distributed protocol needs communication abstractions to support interactions among different nodes of the system. In our three case studies, there are communication steps that benefit from a primitive that sends a message to all (direct) neighbours of a node in the ad hoc network. In particular, the broadcast protocol transmits and retransmits messages using this approach; B.A.T.M.A.N. disseminates announce messages in a similar fashion; and in GAP, nodes propagate updates summarising their current view of the tree topology and aggregation to all neighbours. This implies that the framework should facilitate the use (and expose) a **one hop broadcast communication primitive** [82].

Communication Abstractions: Point-to-Point

Distributed protocols cannot solely rely on an one hop broadcast primitive. Often messages are only intended to a single destination, and having the neighbouring nodes waste

CPU cycles to process irrelevant messages is undesirable. This is clear when considering the operation of B.A.T.M.A.N., where messages that are routed are sent to particular nodes that are responsible to keep forwarding the message, always following the same pattern, until it reaches the node to which the message is addressed. Consequently, it is beneficial to also expose a **point-to-point communication primitive** that allows a node to send a message directly to a specific direct neighbour in the wireless ad hoc network.

Protocol Interaction: Request & Reply

Protocols do not execute in complete isolation. For instance, messages disseminated by the broadcast protocol or transmitted through the routing protocol are generated by either an application or some other protocol, that delegates, to one of these protocols, the responsibility to deliver it to the appropriate nodes or node. Another relevant example of this can be found in the operation of the aggregation protocol. In this case, while the protocol can compute the aggregation without external interactions, the result of the aggregation should be exposed to other protocols (or application) when requested. To deal with this need of interaction among protocols, the framework should provide a mechanism that allows a protocol to **send a request** to another protocol. These requests will typically require a concrete action to be taken by the receiving protocol (e.g., broadcast a message; forward a message). Some of these requests might need to produce a reply (e.g., report the aggregated value when requested), therefore the framework should also support the option of a **reply to be sent back to the protocol or application that created a request**.

Protocol Interaction: Notification

Various protocols operating within the context of a single process may depend on the information that another protocol obtains during its execution. They could retrieve this information through the use of the previous interaction, although this would be highly inefficient as they are not aware when the relevant information may become available. Consider the example of GAP that relies on the operation of a companion discovery protocol. Every time the discovery protocol discovers a new neighbour, it should make this information available to GAP and any other protocol that requires it. Hence, the discovery protocol should have a mechanism to notify these protocols that a new neighbour has been discovered. However, during the implementation of the discovery protocol there is no knowledge on which protocols may need the information at runtime. The consequence is that another, more indirect, form of interaction must be made available for protocols. To address this, the framework should support a mechanism of **notification**, where a protocol can at any time send a notification, with a pre-defined identifier, that is transparently delivered to any protocol that registered interest in that type of notification. This operates akin to a publish-subscribe system however, limited to the scope of protocols in the context of a single process.

Timer Management

A prevalent behaviour found in the specification of protocols, which is illustrated by our routing and aggregation case studies, is the need to perform actions periodically, such as sending a message. Due to the frequent need of this behaviour, and to assist developers in handling this potentially error prone task, providing support for timed actions is relevant. To that end, the framework should support an integrated management mechanism for timers (i.e., operations whose execution depends on the passing of time). **Timers** can be **periodic**, for when a protocol needs to perform a given action at recurrent time intervals (until the timer is canceled); or be **unique**, triggering once after a given amount time. The latter is relevant, for instance, to manage timeouts, an aspect found in many protocols in actions related with fault-tolerance, and that is illustrated in the broadcast case study, where a random small delay is applied before retransmitting messages.

3.2 Yggdrasil: Design & Implementation

We now discuss the design and implementation of the Yggdrasil framework. We start by defining our system model through a set of assumptions, that in conjunction with the previously presented requirements, motivate our design choices. We then present a high level overview of the architecture of the framework, and discuss its main components. Finally, we detail how we have implemented the framework using the C language.

3.2.1 System Model

We consider a system operating as a set of processes that cooperate and interact to materialise an application. Each process resides in a node which has its own resources (CPU, memory, disk, etc). All interactions between processes are performed through the exchange of messages via the wireless medium. We do not assume any form of native routing, meaning that messages are only exchanged by processes that are directly reachable through their radio device (i.e., within radio range).

A single process combines logic provided by the development and execution framework (Yggdrasil), a set of protocols, and one or more application components. Protocols provide services or abstractions to other protocols or application components. Application components contain logic specific to the application, and are responsible for interactions that go beyond the system (i.e., users or other systems).

3.2.2 Design Choices

State Machine Model

In Yggdrasil, protocols are modelled (and implemented) as a state machine. This means that each protocol has its own internal state that evolves according to the reception and processing of events. Protocols can themselves generate events to be processed by that

protocol, or to be delivered to other protocols. The events that guide the execution of protocols are of four types and are motivated by frequent protocol patterns as discussed previously: *i*) Messages, that are the only type of event that can be transported between protocols that execute in independent Yggdrasil instances (i.e., different processes/nodes); *ii*) Timers, that notify a protocol to execute some periodic task or that a local timeout occurred; *iii*) Requests/Replies, that allow the direct interaction between protocols in the context of one Yggdrasil instance; and finally, *iv*) Notifications, that allow the indirect interaction between protocols in the context of one Yggdrasil instance.

Taking Advantage of Multi-Core Devices

An application for wireless ad hoc networks can (easily) require the support of multiple and independent protocols to operate. This can lead these protocols to compete for device resources during execution, namely CPU, which could have an overall negative impact on application performance. Since nowadays many devices have multiple CPUs (for instance Raspberry Pi 3 - Model B has four CPUs [73]), we decided that we should promote parallel and concurrent execution of protocols and applications. To this end, protocols have the possibility of executing in the context of independent threads or alternatively, some protocols can be executed by a shared thread. This should be defined by application developers, according to particularities of the hardware where they plan to execute their applications, and properties of the protocols being used. However, developers should not be required to handle complexities related with concurrent execution. Consequently, the framework is responsible for transparently *manage the execution of various execution threads*, hiding concurrency issues that might occur (e.g., various protocols try to send a message through the radio at same time).

Dynamic Management of Protocols

We note that in practice, there are different protocols that offer the same service/abstraction to other protocols and applications components. Often different alternatives exhibit different performance on different execution environments (which might only be possible to infer at runtime). For example, if the nodes are too clustered a broadcast protocol based on flooding, might generate too many redundant receptions; whereas in less clustered environments, effectively all nodes should retransmit each message. This creates the need to enable the activation/deactivation or switching [65] of protocols at runtime, as to pave the way for autonomic management mechanisms. We note that, although this feature is available in Yggdrasil, it is not explored in the context of this thesis.

Support for Debugging and Experimental Validation

One of the potential benefits of employing a framework to develop and execute protocols and applications in wireless ad hoc networks, are the existence of support tools for performing validation and testing. Due to this, we believe that a minimal set of tools to

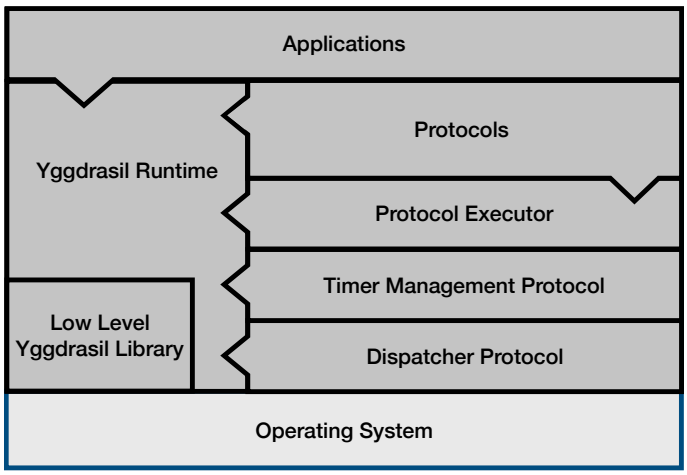


Figure 3.1: Simplified Yggdrasil's Architecture.

simplify these tasks should be provided with the framework. We have developed a first prototype of such a tool, that enables the coordination of the execution of applications using the framework across multiple devices without the need for devices to be plugged to a (control) infrastructure network. We provide some details on this tool in the context of our own experimental work reported on Chapter 5.

3.2.3 Architecture

Figure 3.1 depicts a high level view of the Yggdrasil architecture. Yggdrasil is a framework that operates above the operating system. In particular, Yggdrasil was developed considering a unix-based general-purpose operating system. From the operating system we assume abstractions to configure the radio device and consequent wireless network, besides standard abstractions and programming interfaces (such as concurrent execution and synchronisation mechanisms as mutex and semaphores).

The core component of Yggdrasil is the *Yggdrasil Runtime*. This component offers all of the high level abstractions of the framework (many of them discussed previously), the programming API for developers, and the support for the runtime execution of protocols and application components. The core of Yggdrasil is small as to ensure its simplicity and robustness.

The Yggdrasil Runtime relies on the *Low-Level Yggdrasil Library* to interact with the operating system and manage the radio device and network configuration. The library is responsible for exposing an abstraction through which communication (via message exchange) is achieved among multiple devices. This abstraction is named a *Channel*. The motivation for having this library is two fold. First, it allows us to isolate this component from the Yggdrasil Runtime, minimising the probability of introducing bugs in the core. Second, it simplifies the growth of the framework to support other communication environments, by either replacing or adding new libraries that expose their functionality through a channel.

Some mechanisms of Yggdrasil are supported via specialised protocols (that we named *core protocols*) that are more tightly coupled with the Yggdrasil Runtime than regular user protocols. This coupling is materialised in the form of API functions that directly interact with these protocols. The core protocols are: *i*) **Dispatcher**, which is responsible for managing the transmission and reception of network messages using the channel abstraction exposed by the Low Level Library; *ii*) **Timer Management**, which is responsible for the management of timers for all other protocols executing in a process; and *iii*) **Executor** that allows the developer to have multiple protocols sharing a single execution thread (the default execution model in Yggdrasil is to have each protocol to execute in the context of its own private execution thread). While the framework offers implementations for the three core protocols, the user can replace any of them by one of her own design. This is useful, for instance, for debug purposes or to extend the functionality of the framework. As a concrete example, for debug purposes, one could replace the Dispatcher's implementation such that all messages sent or received are logged to a file.

Finally, a main application and its protocols/components (either provided by the framework, or user defined) will interact with the Yggdrasil Runtime and its core protocols. The main application is responsible for configuring the Runtime through the API exposed by it. This configuration includes specifying the protocol and application components that are going to be executed, and optionally defining some properties for the wireless ad hoc network to be employed, such as the network name and WiFi radio channel to be used.

3.2.4 Implementation Details

A prototype of Yggdrasil was developed in the C language. This prototype includes all the previously described functionalities and provides implementations of additional protocols. These include: neighbour discovery; failure detection; aggregation protocols; communication protocols, such as routing, broadcast, and reliable point-to-point delivery; topology control; and experiment management protocols. The last two are dedicated to simplify experimental validation and benchmarking activities. The prototype also provides support libraries with auxiliary functions for the creation and manipulation of the different types of events, generic data structures, and other utilities. In total, the current prototype contains more than 35.000 lines of code. We now give some details on this implementation and its main features[1].

Protocol Definition:

The basic unit of Yggdrasil are protocols and application components. This is the most relevant part for programmers when leveraging Yggdrasil's abstractions, as it is on the development and execution of protocols and applications that Yggdrasil focusses on. As previously described, protocols are modelled as state machines (we discuss application

[1] This code is available online at: https://github.com/LightKone/Yggdrasil

components further ahead). As such, protocols must contain a *defined state* containing all the variables that will be manipulated or used by the protocol.

The internal state of a protocol must evolve, and do so by the processing of events. As such, a protocol must have a way of handling the processing of events. Yggdrasil provides developers with two methods of defining the logic of event handlers, we named these *Main Loop* and *Event Handlers*.

Main Loop: Using this method the developer must provide an implementation of a main control loop function of a protocol. This main control loop is then responsible for consuming events and process them accordingly. Yggdrasil provides a mechanisms (described ahead) that allows the main control loop to wait until an event for that protocol becomes available. This methodology provides programmers with more control over the behaviour of a protocol when implementing it.

Event Handlers: Using this method the developer only has to implement handlers (i.e., individual functions) for the types of events that are relevant for her protocol. For clarity, this means for instance that if a protocol does not consume a notification event, it does not need to implement a notification event handler. These handlers have predefined interfaces, that must be respected by the developer. In contrast to the previous methodology, this methodology provides programmer with a simpler and more systematic interface for developing their protocols.

The implementation of protocols is not exclusive to one of these methodologies. In fact, the implementation of a protocol can combine both methodologies, by using a main control loop that calls the appropriate event handler. Nevertheless, we perform a distinction over them because they internally represent different execution models. While defining a main loop is associated with the protocol executing in a single independent thread, defining event handlers is associated with the protocol executing in the context of a shared thread. This behaviour is defined upon the initialisation of the protocol.

Protocols should follow two additional implementation requirements. The implementation of a protocol should define an initialisation function and a destructor function. The initialisation function is responsible for initialising the protocol's state, preparing the triggering of initialisation events (e.g., setup a periodic timer), and creating a *spec*. This spec is used by the Runtime to correctly prepare and manage the execution of the protocol. As such, this spec should contain the *protocol id*, a unique internal numeric identifier; the *protocol's name*, a string for debug purposes; the *protocol's state*, so that the Runtime can instantiate the protocol upon its execution; the *notifications it produces and consumes* if any, enabling the publish-subscriber interface used for indirect interactions among protocols; the function pointer to the *protocol's main loop* or the collection of function pointers to the *relevant event handlers*, defining the execution model; and finally, the function pointer to the *destructor function*. The destructor function is responsible for releasing the memory allocated by the protocol's state, if the protocol is terminated.

Protocol Interaction:

In Yggdrasil, as explained previously, interactions between protocols are performed via events. Yggdrasil supports four types of events detailed previously (timers, messages, requests/replies, and notifications), and offers specific data types that represents each one. All data types include a header part that identifies the protocol (via its protocol id) which generated said event, and a payload field that is managed by the protocol that creates/consumes the event. Each data structure has its how particularities, for example, the payload of the message data type is of constant size; the timer data type has a randomly generated identifier and two time fields, a first one containing the absolute time for when the timer should first trigger, and a second one containing the period, if any, for recurrent triggers for periodic timers; notification and request/reply data types also contain the numeric identifier of the destination protocol, with the difference that in the notification data type this field is manipulated by the Runtime, while in request/replies it is manipulated by the protocol that generates the event.

Each data type, has additionally, auxiliary functions to initialise the header fields, and add, retrieve, and free the information in the payload field. In particular, timer data types have an additional auxiliary function to set the relative time for when the timer needs to be triggered.

Protocols need to dispatch these events in order for them to be delivered to the appropriate destinations. For this purpose, the Runtime provides a clear API to dispatch events. Each type of event has its own function to be called for triggering its delivery to other protocol or protocols. The Runtime simply checks the destinations and delivers the event to the appropriate protocol(s). Again, there are some particularities that are specific for the message, timer, and notification event types. Message event types, can either be dispatched to the network or delivered to the appropriate protocol once it is received from the network. Timer event types can be set or canceled. Regarding notifications, as they have no concrete destination upon its creation, the Runtime checks and delivers them to all protocols that registered interest in receiving said notification.

Protocols consume these events through an *event queue*. This event queue is provided and managed by the Runtime upon the registration of protocols that are part of an application.

Events are consumed from the event queue in FIFO order however, different types of events have different priorities. The current priority is the following: Timers, Notifications, Messages, Requests/Replies. The motivation for this ordering is related with the typical delay sensitivity of the different types of events although, the ordering can be easily changed by redefining the structure of an enumerate.

To handle this type of behaviour, given that events have different sizes, the event queue is implemented by having a set of inner queues that are dedicated to the handling of each type of event. When an event is to be popped from the queue, the queue performs a search over the contents of inner queues in order, and retrieves an element from the first

one that has an element. Each inner queue is materialised by a circular buffer to avoid continuous reallocation of memory however, the circular buffer may grow when needed.

As a final note, event queues have mechanisms to be blocking, in other words, if there is nothing to be consumed from an event queue, the protocol must wait. This mechanism can however be bypassed, as event queues also implement a function where protocols can specify a maximum time for waiting for an event. This can be useful for protocols that need to perform specific actions when there is no event to be consumed.

Core Protocols:

The framework offers implementations for all core protocols. In particular, we note that the Dispatcher protocol is responsible for processing all messages sent by protocols, by serialising them, and sending them to the network. Additionally, this protocol also waits for messages to be received. Upon reception, the protocol deserialises the messages, and delivers them (through the Runtime) to the appropriate protocol (being silently dropped by the Runtime if the destination protocol does not exist). The Dispatcher ensures that information about the sender (i.e., MAC address) is correct when sending messages to the network. Our implementation of the Dispatcher also contains an ignore list, that can be manipulated by other protocols through request events, to ignore certain origin nodes.

The Timer Management protocol allows for various protocols in an Yggdrasil instance to register periodic or unique timers. As such, every timer that is set by a protocol will be delivered to the Timer Management protocol. The protocol maintains an ordered queue of all registered timers, and when these expire, the protocol delegates to the Runtime the delivery of a copy of that Timer event to the appropriate protocol. Timers can be canceled as discussed previously, which will trigger the Runtime to deliver a copy of the Timer event to be canceled with both time fields set to zero.

Yggdrasil has a third core protocol named *Executor*. This executor is a *pseudo-protocol* that can be used to at runtime to dynamically register or unregister protocols. The Executor is responsible for managing the execution of those protocols in the context of a single shared execution thread. This is achieved by having all those protocols sharing a single event queue. The Executor thread then monitors this queue for events to any of its registered protocols. Whenever an event arrives, the Executor executes the handler function associated with that type of event and the destination protocol operating over the internal state of the protocol.

Other Features:

The framework has other features that have proved to be beneficial for implementing some common behaviours in distributed systems and also to optimise the behaviour of multiple protocols executing in parallel. One example can be found on protocols that aim at *piggyback* information in the messages exchanged by other protocols. This is a very common technique to reduce the number of messages exchanged through the

network, where some additional information is added to the payload of messages (without surpassing the maximum size of a frame in the network) that is transparently processed by the receiver.

Yggdrasil supports this feature by enabling a protocol to intercept the event queue of another. This means that whenever some protocol delivers an event to the original queue, the event is instead transparently delivered to the queue of the intercepting protocol, which then becomes responsible to deliver the event to the original event queue. Piggyback in this context is implemented as follows.

A protocol that wants to piggyback information, intercepts the event queue of the Dispatcher protocol. Hence, whenever a protocol generates a message to be sent, instead of the message being put into the Dispatcher protocol's event queue, it is put into the piggybacking protocol event queue. The protocol receives the message, and verifies if there is enough free space in the payload to add its information. If so, it starts by adding to the payload information about the original destination of the message (both MAC address and protocol id). It then adds the information to be piggybacked to the payload of the message, and updates the destination protocol to be itself (so that the message is processed by the same protocol on arrival at the destination node). Optionally, the MAC address of the destination can be updated if it is destined to a particular node, to the broadcast address (other nodes will discard the message after processing the piggybacked data). The message is then directly delivered to the Dispatcher for transmission.

Additional Abstractions:

A common pattern found in the state maintained by protocols is the management of collections of information, usually stored in linked list. To facilitate this, Yggdrasil also offers an implementation of a generic linked list. Furthermore, Yggdrasil offers a specialised implementation of this list to hold information regarding neighbouring nodes.

In addition, Yggdrasil provides implementations of common interaction interfaces. These are mainly for the delivery and reception of notifications and requests/replies regarding the discovery of a new neighbour, the request to route a message to a specific destination, and the request to obtain the aggregation result.

3.2.5 Applications in Yggdrasil

As mentioned before, an application in Yggdrasil is composed by a set of protocols and application components. Application components contain logic specific to the application, and as such, we delegate to the programmer the design, implementation, and execution model of the component. The only restrictions imposed to programmers is the interaction model with protocols, which has to be done via events (naturally, the same events that protocols use). Similarly to protocols, events are consumed via event queues that are managed by the Yggdrasil Runtime, consequently, application components must be registered in the Runtime, as to obtain their own event queue.

3.3 Showcase Exercise

Algorithm 1: Simple Discovery

1: **Local State:**
2: N_{id} //Node identifier
3: Neighs //Set: (N_{id})

4: **Upon Init ( ) do:**
5: Neighs $\longleftarrow$ {}
6: **Setup Periodic Timer** Beacon (ΔT)

7: **Upon** Beacon **do:** //every ΔT
8: **Trigger** OneHopBCast (N_{id})

9: **Upon** Receive (id) **do:**
10: **if** $\nexists\ id \in$ Neighs **then**
11: Neighs $\longleftarrow$ Neighs $\cup$ {id}
12: **Trigger** Notification New Neigh(id)

We now present an example to show how one can leverage the previously explained abstractions provided by Yggdrasil to efficiently implement protocols. For this, we will use a simple protocol that is fundamental for the operation of many of the presented protocols in Chapter 2, a discovery protocol that enables other protocols executing simultaneous to learn which are the other nodes with whom they can communicate. We provide the pseudo code of this protocol in Algorithm 1.

The discovery operates as follows: every ΔT it announces the node's identifier (a random bit string) through one hop broadcast (Alg. 1 line 8). Upon the reception of an announcement, the algorithm checks if it already knows the identifier in the message (Alg. 1 line 10). If the identifier is new (not contained in the set hold by the algorithm), it is added to the list of known identifiers and a notification of a new neighbour is sent (Alg. 1 lines 11 – 12).

In the following, we show how this algorithm is translated to C code in Yggdrasil. We will begin by explaining how the state of the protocol is defined.

Protocol State:

```
1  typedef struct _simple_discovery_state {
2    uuid_t myid;
3    list* neighbours;
4    short proto_id;
5    YggTimer announce;
6  }simple_discovery_state;
```

Figure 3.2: Simple Discovery State.

By using Yggdrasil, the programmer must first define a structure that contains the

state of the protocol, as depicted in Figure 3.2. The first two variables defined are related to the actual state maintained by Algorithm 1 as demonstrated at lines 2 and 3, where the variable type `uuid_t` is, in short, a redefinition of a `char` array with 16 positions provided by a C library (`uuid`) that provides functions to generate universally unique identifiers (uuids). The variable type `list` is the linked list provided by Yggdrasil explained previously.

The last two variables defined in the state encode information that must be maintained by Yggdrasil, such as the protocol's numerical identifier (`proto_id`) and the periodic timer that the protocol uses for its periodic beacon action (`announce`), where the data type `YggTimer` represents a Timer event. This last variable, is not mandatory to be maintained by this protocol per se however, it is useful to maintain the original Timer event for instance, to cancel it or change its parameters.

Protocol Handlers:

```
1  static short process_timer(YggTimer* timer, simple_discovery_state* state) {
2    YggMessage msg;
3    YggMessage_initBcast(&msg, state->proto_id);
4    YggMessage_addPayload(&msg, (void*) state->myid, sizeof(uuid_t));
5
6    dispatch(&msg);
7
8    return SUCCESS;
9  }
10
11 static short process_msg(YggMessage* msg, simple_discovery_state* state) {
12   uuid_t id;
13   void* ptr = YggMessage_readPayload(msg, NULL, id, sizeof(uuid_t));
14
15   if(neighbour_find(state->neighbours, id) == NULL) {
16     neighbour_item* newnei = new_neighbour(id, msg->srcAddr, NULL, 0, NULL);
17     neighbour_add_to_list(state->neighbours, newnei);
18
19     send_event_neighbour_up(state->proto_id, newnei->id, &newnei->addr);
20
21   }
22   return SUCCESS;
23 }
```

Figure 3.3: Simple Discovery Handlers.

We next define the handlers that will process each type of event relevant for the operation of the protocol. The protocol needs to perform timed actions and receive messages hence, we will define handlers for Timer events and Message events. This is shown in Figure 3.3. Each handler receives the respective event (Timer or Message), represented by the data types `YggTimer` and `YggMessage` respectively, and the state to be modified or consulted. Furthermore, handlers are specified to return if they succeeded or failed in their execution.

As specified by Algorithm 1, when the protocol receives a Timer event, it should send a message through one hop broadcast. For that end, in the implementation (function `process_timer`) we begin by defining a message data structure (`msg`), we initialise this data structure to be sent through one hop broadcast (function `YggMessage_initBcast`).

This entails setting the destination address of the message to be the physical broadcast address (`ff:ff:ff:ff:ff:ff`) and tagging the message with the protocol's numeric identifier. We must add the node's identifier to the message's payload (function `YggMessage_addPayload`). The message is then dispatched to the network by sending the message to the Dispatcher protocol (function `dispatch`).

When the protocol receives a message (function `process_msg`), it must check if it already knows the identifier of the sender. To this end, we must extract the information contained in the payload of the message. This is achieved by the function `YggMessage_readPayload` that reads the payload of the message (first parameter `msg`), from the position pointed by the pointer in the second parameter (`NULL` in case if it is the beginning of the payload), and stores the read contents in the provided third parameter (in this case an `uuid_t id`). The fourth parameter denotes the number of bytes to read. Additionally, the function returns the pointer that points to the last position that was read from the payload, allowing to reuse it to read additional elements from the payload. In this example this feature is not used as we only have one element to read from the payload.

Next, we search the neighbour list for the received identifier (function `neighbour_find`), which will either return a pointer for the neighbour item (i.e., an item in the list that encodes information related to a neighbouring node) that has that identifier, or `NULL` if it does not exist. If the neighbour does not exist, we create a new neighbour item with the received information (function `new_neighbour`) and add it to the list of neighbours (function `neighbour_add_to_list`). The protocol must now send a new neighbour notification, which achieved by the function `send_event_neighbour_up`. We do not show the implementation of this function however, what it does is create a Notification event, add to the notification's payload the identifier and MAC address of the new neighbour (similarly to how the message data structure is initialised), and ask the Runtime to deliver the event to every interested protocol/component (similarly to the `dispatch` function).

Protocol Initialisation:

The only thing left for the programmer to do, is the initialisation function that will be used to register the protocol in the Yggdrasil Runtime. Figure 3.4 presents the C code for this. Additionally, Figure 3.4 also presents the destructor function (`simple_discovery_destroy`) that frees the state allocated by the protocol. The initialisation function (`simple_discovery_init`) receives the parameters used by the protocol, this is encoded in the data structure `simple_discovery_args` that should be defined in the corresponding header file of the protocol. In this case this structure contains the announce period of the protocol.

The initialisation function first initialises the state of the protocol, and then creates the spec for the protocol (function `create_protocol_definition`) where the programmer must provide the protocol's numerical identifier (also defined in the header file), the protocol's name, the state of the protocol, and the protocol's destructor function. Then this spec must be configured, to this end, the programmer must provide the notifications that

```
1  static short simple_discovery_destroy(simple_discovery_state* state) {
2      cancelTimer(&state->announce);
3      neighbour_list_destroy(state->neighbours);
4      free(state->neighbours);
5      free(state);
6
7      return SUCCESS;
8  }
9
10 proto_def* simple_discovery_init(simple_discovery_args* args) {
11
12     simple_discovery_state* state = malloc(sizeof(simple_discovery_state));
13     getmyId(state->myid);
14     state->neighbours = NULL;
15     state->proto_id = PROTO_SIMPLE_DISCOVERY_ID;
16     state->neighbours = list_init();
17
18     proto_def* discovery = create_protocol_definition(state->proto_id, "Simple Discovery", state,
        simple_discovery_destroy);
19     proto_def_add_produced_events(discovery, 1); //NEIGHBOUR_UP
20
21     proto_def_add_msg_handler(discovery, process_msg);
22     proto_def_add_timer_handler(discovery, process_timer);
23
24     YggTimer_init(&state->announce, state->proto_id, state->proto_id);
25     YggTimer_set(&state->announce, args->announce_period_s, args->announce_period_ns, args->
        announce_period_s, args->announce_period_ns);
26
27     setupTimer(&state->announce);
28     return discovery;
29 }
```

Figure 3.4: Simple Discovery Initialisation.

the protocol will produce (function `proto_def_add_produced_events`), in this case only one notification is produced, the new neighbour notification. The programmer must also provide the spec with the protocol's handlers (functions `proto_def_add_msg_handler` and `proto_def_add_timer_handler`). Finally the periodic Timer event is configured by initialising the Timer event data structure (function `YggTimer_init`) with the protocol's numeric identifier, and by setting its time parameters (function `YggTimer_set`). The Timer event is then delivered to the Timer Management protocol through the Runtime (function `setupTimer`).

3.4 Summary

In this Chapter we presented the first contribution of this thesis, the Yggdrasil framework, that provides the grounding for the rest of our work. We began by motivating Yggdrasil's design with the help of the operation of some protocols presented in the previous Chapter, we then detailed Yggdrasil's implementation and provided a showcase example for the reader to understand how can one make use of Yggdrasil. We conclude this Chapter with some final remarks.

The Yggdrasil framework began with the development of its low level library. Its first objective was to configure the device radio programatically and send messages through one hop broadcast. Since then, the Yggdrasil framework evolved with the Runtime to support the execution of multiple protocols. Protocols where viewed as independent execution threads that interacted through the Runtime. Latter, we designed the Executor

protocol so that we could support protocols executing in the context of shared threads, and to date we are still exploring other useful abstractions and methodologies that could be incorporated in Yggdrasil.

The Yggdrasil framework is still being developed as to address more use cases and other relevant issues that may arise in the future. However, the prototype presented in this Chapter is a stable version of the Yggdrasil framework. In particular, Yggdrasil was already used by a student to develop and evaluate different flavours of broadcast protocols for wireless ad hoc networks. Moreover, Yggdrasil was crucial to the development and experimental evaluation of the second contribution of this thesis, that we present in the next Chapter.

Chapter 4

Multi Root Aggregation: MiRAge

In this Chapter we discuss the design of our own distributed aggregation protocol that is capable of performing continuous aggregation naturally in an efficient and reliable fashion. We begin by providing the system model we consider (Section 4.1), we then proceed to present a brief overview of the protocol (Section 4.2). We follow this by providing the details of the protocol's operation (Section 4.3) that is divided into two parts. A first one where we describe how values are aggregated in our protocol, and a second one where we discuss how we build and maintain the tree topology that supports the aggregation process.

4.1 System Model

We assume a distributed system where nodes communicate via message exchange. Furthermore, we assume that devices are equipped with a WiFi radio capable of operating in ad hoc mode. Each node is pre-configured to join a single ad hoc network. We do not assume any routing algorithm or infrastructure access. Devices can transmit messages using one hop broadcast, where the message can be received (with some probability) by all, or a subset of devices in the transmission range of the sender.

No node is aware of the total number of nodes in the system. However, we assume that each node has a unique identifier (this can be achieved by having each node generate a large random bit string at bootstrap) that can be ordered within the system. We do not make any assumption regarding clock synchronisation although, we assume that each node perceives the passing of time at a similar (albeit, not necessarily equal) rate.

Finally, we assume that each device runs a discovery protocol, similar to the one described in the previous Chapter, where periodically the node transmits (in one-hop broadcast) an announce containing its own identifier. The period of this transmission

is controlled via a parameter ΔD. This protocol is also used by each node as an unreliable failure detector, where if the announce of a known node is not received for a (large enough) consecutive number of transmission periods, the node becomes suspected of having failed, generating a notification to the aggregation protocol. The number of transmissions that a node can miss before suspecting another one is a parameter denoted K_{fd}. This is an assumption made by many other aggregation protocols described previously in Chapter 2 namely GAP [23], LiMoSense [30], Flow-Updating [42], among others.

4.2 Overview

Our protocol is inspired in the design of GAP, but generalises its design to remove the dependence of a single root. To this end, our protocol leverages on a self-healing spanning tree to *support* efficient continuous aggregation. In our protocol, that we named Multi Root Aggregation, or simply MiRAge, all nodes compete to build a tree rooted on themselves, similarly to the self-stabilisation algorithms explain previously however, without the need for additional messages.

The competition is controlled via the identifier of each node (a large random bit string) and a monotonic sequence number (i.e., a timestamp) controlled by the corresponding root node. Additionally, our protocol was designed to ensure that all nodes in the system are able to continually compute and update the result of the aggregate function while the tree is being constructed and maintained.

To provide a clearer intuition of our protocol we present two complementary concepts regarding the construction and maintenance of the support spanning tree. First, a tree is considered **correct** while its root is non-faulty, which is verified by nodes through the observation of increasing sequence numbers (i.e., timestamps) for that tree. Since we must have a single tree that spans the network to provide an efficient and reliable way to aggregate values, we define the concept of **dominating tree**. The dominating tree is the correct tree that will ultimately span the entire network. We further detail that a tree a dominates over a tree b if the identifier of the root of a is lower than the identifier of the root of b. Hence, we can define the dominating tree as the correct tree whose root has the lowest identifier within the whole network.

Nodes periodically exchange information regarding the tree they are attached to, and the aggregated value being computed. When a node receives information of a correct tree that dominates over its current tree, the node simply attaches itself to the new tree.

When performing aggregation, unlike in GAP and other topology-based approaches, in MiRAge, nodes have no need to know who are they children and parent in the tree. They only need to keep track of the links that belong to the tree. Intuitively, and because a tree topology represents a single path (in any direction) for information to be propagated, each node can be viewed as a root of the tree to which aggregated values are propagated to. In more detail, a node propagates to a neighbour the resulting aggregated value without the effects of the contribution of that neighbour.

4.3 Multi Root Aggregation

We now present the design of MiRAge, which is divided into two parts for better comprehension. We start by explaining how the aggregation result is computed by each node in the system. We then explain how the natural evolution of the protocol through the exchange of messages between nodes, allows to maintain the self-healing spanning tree that supports the aggregation. These two parts are respectively represented in Algorithms 2 and 3 in pseudocode.

4.3.1 Aggregation Mechanism

Algorithm 2 describes the local state maintained by each node executing MiRAge and the components of the protocol related with computing the aggregation function at each node.

In MiRAge, each node owns a unique node identifier (Alg. 2 line 2) and stores its own input value for the aggregation (Alg. 2 line 3). Additionally, each node maintains its current estimate of the aggregation result (Alg. 2 line 4) and a set of known neighbours, where the following information is maintained for each neighbour: *i*) the neighbour's node identifier; *ii*) the latest received estimation; *iii*) the neighbour's current status in the support spanning tree, being `Active` if the neighbour shares a tree link with the local node and `Passive` otherwise; *iv*) the identifier of the tree that the neighbour is connected to; and *v*) its level in that tree (Alg. 2 line 5).

Each node also owns a set of local variables that capture its current position and configuration in the support spanning tree. These include: the identifier of the tree to which the node is currently connected (tree identifiers are the identifier of the tree root node), its level (the root of the tree has level zero), the highest timestamp observed for the current tree, and the identifier of the parent node (Alg. 2 lines 6 – 9). In addition, each node stores a map that associates tree identifiers to the highest observed timestamp for that tree (Alg. 2 line 10).

When a node is initialised (Alg. 2 line 11) it has no knowledge regarding existing neighbours. Due to this, it assumes that the result of the aggregation function is its own value, and initialises the state related with the support spanning tree to reflect a tree rooted on itself (the tree identifier being its own identifier). Additionally, the node setups a periodic function named Beacon that is executed every ΔT, which corresponds to the main aggregation logic of our algorithm. A typical value for ΔT is one or two seconds.

When the Beacon procedure is executed, a node will start by updating its local estimate. This is achieved through the execution of the `updateAggregation` procedure, which applies the aggregation function represented by the operator $\oplus$ in Algorithm 2 line 32, to the input value of the node with the received estimates of neighbours whose link with the local node has been marked as belonging to the node's current tree (i.e., status = `Active`).

Algorithm 2: MiRAge: Aggregation Function Computation.

1: **Local State:**
2: N_{id} //Node identifier
3: Value //current input value
4: Aggregation //Current result of aggregation
5: Neighs //Set: (N_{id}, value, status, T_{id}, T_{lvl})
6: T_{id} //Unique identifier of the tree to which the node is currently attached (N_{id} of tree root)
7: T_{lvl} //Level of the node in its current tree
8: T_{ts} //Higher timestamp of current tree
9: P_{id} //Identifier of current tree parent
10: Trees //Map: $Trees[T_{id}] \rightarrow$ Timestamp

11: **Upon Init ( ) do:**
12: Value $\longleftarrow initValue()$ //initial input value
13: $T_{id} \longleftarrow N_{id}$
14: $T_{lvl} \longleftarrow 0$
15: $T_{ts} \longleftarrow now()$ //now( ) = current time
16: $P_{id} \longleftarrow N_{id}$
17: Neighs $\longleftarrow \{\}$
18: Aggregation $\longleftarrow$ Value
19: **Setup Periodic Timer** Beacon (ΔT)

20: **Upon** Beacon **do:** //every ΔT
21: **Call** updateAggregation()
22: **if** ($T_{id} = N_{id}$) **then**
23: $T_{ts} \longleftarrow now()$
24: msg $\longleftarrow \{\}$
25: **foreach** ($id, val, stat, tid, tlvl$) $\in$ Neighs $\wedge T_{id} = tid \wedge stat =$ Active **do**
26: msg $\longleftarrow$ msg $\cup$ (id, Aggregation $\ominus$ val)
27: **Trigger** OneHopBCast (< N_{id}, T_{id}, T_{lvl}, T_{ts}, P_{id}, Value, msg>)

28: **Procedure** updateAggregation()
29: Aggregation $\longleftarrow$ Value
30: **foreach** ($Neigh$, V_{neigh}, $Status$, T_{neigh}, L_{neigh}) $\in Neighs$ **do**
31: **If** ($Status =$ Active) **then** //Active $\rightarrow T_{id} = T_{neigh}$
32: Aggregation $\longleftarrow$ Aggregation $\oplus$ V_{neigh}

33: **Upon** Receive (< id, tid, $tlvl$, tts, pid, val, msg >) **do:**
34: **Call** updateNeighbourEntry(id, tid, $tlvl$, val)
35: **if** $\exists(N_{id}, RecvVal) \in$ msg **then**
36: **Call** updateNeighbourEntry(id, tid, $tlvl$, $RecvVal$)
37: **Call** updateTree(tid, tts, $tlvl$, pid, id)

38: **Procedure** updateNeighbourEntry(id, tid, $tlvl$, val)
39: **if** ($\nexists$ (id', _, $stat$, _, _) $\in$ Neighs : $id' = id$) **then**
40: Neighs $\longleftarrow$ Neighs $\cup$ (id, val, Passive, tid, $tlvl$)
41: **else**
42: Neighs $\longleftarrow$ Neighs $\setminus(id, \_, stat, \_, \_) \cup (id, val, stat, tid, tlvl)$

After updating its local estimate, the node will prepare a message to be transmitted through one-hop broadcast. This message must contain information that encodes a sense of directionality for all nodes to compute the correct aggregation result. Hence, the node computes a partial aggregated value that should be propagated towards each Active

neighbour. This is done by applying some function or method (operator $\ominus$ in Alg. 2 line 26) that removes the effects of the contribution of a neighbour from the previously computed local estimate. The $\ominus$ operator depends on the aggregation function, as it should revert specific modifications applied by the aggregation function (operator $\oplus$ in Alg. 2 line 32) to the locally calculated estimate. For instance, if the aggregation function is the sum function, the $\ominus$ can be specified as being the subtract function effectively removing specific portions of the local estimation. Unfortunately, in the case of the max and min functions it is not as easy to define the $\ominus$ operator as a simple function hence, an effective solution is to recalculate the local estimation without the effects of the specific node's aggregated value. We note that this must be done to guarantee that old maximums or minimums are forgotten by the system.

The computed partial aggregated values are then tagged with the corresponding neighbour's identifier whose contribution effects where removed, effectively forming a tuple for each neighbour. The message is also tagged with the local node's identifier, and the information on the support tree that the node is currently attached to, including the tree identifier, the level of the node, the highest tree timestamp observed, and the identifier of the node's parent (Alg. 2 line 27).

Upon receiving one of these messages (Alg. 2 line 33), a node starts by updating the information it knows about the sender by calling the `updateNeighbourEntry` procedure. This procedure either creates or updates the entry for a given neighbour in a node's neighbour set. The information that is updated is the latest input value observed by that neighbour, the identifier of the tree to which the neighbour is currently attached, and its current tree level (the status of the node remains unchanged and is set to `Passive` if this was the first message received from that node). However, if the sender considered the local node as an `Active` neighbour, then the message contains a tuple with the local node's id and a partial aggregate that must be considered to ensure that all nodes compute the correct aggregated value. In this case, the local node updates again the information regarding the neighbour with the received aggregated value (Alg. 2 line 36).

4.3.2 Tree Management Mechanism

Upon initialisation, each node joins the tree rooted on itself (see Alg. 2 line 13). Whenever nodes exchange aggregation information, they also propagate information regarding their current tree and their position in that tree.

Whenever a node processes an aggregation message (Alg. 2 line 33), it takes advantage of this information to run a local stabilisation mechanism to manage the support tree (Alg. 2 line 37). This is achieved through the procedure `updateTree`, presented in Alg. 3. In a nutshell, this procedure is responsible for ensuring three complementary goals: *i*) the correct dominating tree covers all nodes; *ii*) in presence of failures, the tree is repaired (potentially establishing a new dominating tree); and *iii*) avoid a node to connect to a tree whose root has failed (i.e., tree stability). We now explain how each of these goals is

achieved in MiRAge.

Algorithm 3: MiRAge: Tree Management.

1: **Procedure** updateTree($tid, tts, tlvl, pid, id$) **do:**
2: **if** ($N_{id} = pid$) **then** // I am neighbour's parent
3: **if** ($P_{id} = id$) **then**
4: **Call** commuteToMyTree ( )
5: **else if** ($T_{id} = tid$) **then**
6: **Call** changeNeighbourStatus (id, Active)
7: **else if** ($P_{id} = id$) **then** // Neighbour is my parent
8: **if** ($T_{id} = tid \vee (tid < N_{id} \wedge$ (Trees$[tid] = \perp \vee$ Trees$[tid] < tts$)) **then**
9: **Call** changeNeighbourStatus (id, Active)
10: $newTs \longleftarrow tts$
11: **if** ($T_{id} = tid$) **then** $newTs \longleftarrow \max(T_{ts}, tts)$
12: **Call** updateTreeStatus ($tid, tlvl + 1, newTs, id$)
13: **else**
14: **Call** commuteToMyTree ( )
15: **else** // None of the above
16: **if** ($tid <$ Trees$[tid] = \perp \vee tts <$ Trees$[tid]$)) **then**
17: **Call** changeNeighbourStatus (id, Active)
18: **Call** updateTreeStatus ($tid, tlvl + 1, tts, id$)
19: **else**
20: **Call** changeNeighbourStatus (id, Passive)
21: **if** ($tid \neq N_{id} \wedge$ (Trees$[tid] = \perp \vee$ Trees$[tid] < tts$)) **then**
22: Trees$[tid] \longleftarrow tts$
23: **Call** checkTreeTopology ($T_{lvl} - 1$)

24: **Procedure** checkTreeTopology(lvl) **do:**
25: **foreach** ($id, \_, stat, tid, tlvl$) $\in$ Neighs **do**
26: **if** ($stat =$ Passive $\wedge\ tid = T_{id} \wedge tlvl < lvl$) **then**
27: **Call** changeNeighbourStatus (P_{id}, Passive)
28: **Call** changeNeighbourStatus (id, Active)
29: **Call** updateTreeStatus ($T_{id}, tlvl + 1, T_{ts}, id$)

30: **Upon NeighborDown** (id) **do:**
31: **if** ($P_{id} = id$) **then** // My parent failed
32: **Call** checkTreeTopology (T_{lvl})
33: **if** ($P_{id} = id$) **then** // No suitable parent found
34: **Call** commuteToMyTree ( )
35: Neighs $\longleftarrow$ Neighs \($id, \_, \_, \_, \_$)

36: **Procedure** changeNeighbourStatus($id, stat$) **do:**
37: Neighs $\longleftarrow$ Neighs \ ($id, val, \_, tid, tlvl$)
38: Neighs $\longleftarrow$ Neighs $\cup$ ($id, val, stat, tid, tlvl$)

39: **Procedure** updateTreeStatus($tid, tlvl, tts, pid$) **do:**
40: $T_{id} \longleftarrow tid$
41: $T_{lvl} \longleftarrow tlvl$
42: $T_{ts} \longleftarrow tts$
43: $P_{id} \longleftarrow pid$

44: **Procedure** commuteToMyTree( ) **do:**
45: **Call** updateTreeStatus (N_{id}, 0, $now()$, N_{id})

Establishing the Dominating Tree:

When a node joins the system, it establishes itself as the root of its own tree. Since nodes exchange information regarding the tree to which they are connected, and since tree identifiers are unique, every node can compare the identifier of its current tree with the tree to which a neighbour is connected. This enables nodes to switch to a different tree if it dominates over their current tree, setting the node that sent information about the new dominating tree as their parent (Alg. 3 lines 12 and 18). This allows a single (correct) dominating tree to eventually be established among all nodes.

Additionally, and since each node only maintains a single parent, it is easy to ensure that no cycles exist in the tree. In particular, assume that a node a switches parent from node p to p'. The next message transmitted by a will report p' as being its current parent. This allows p to set the local status of a to `Passive`, effectively removing the link between these nodes from the tree (Alg. 3 line 20). Furthermore, p' will observe this message and set the status of a to `Active`, ensuring the link between them is now part of the tree (Alg. 3 line 6). This simple procedure allows a topology with no cycles and no redundant links to be established.

Notice however, that a will only set the status of p to `Passive` when it receives a new message from p. This is because the algorithm tries to reuse portions of the already established paths for the dominating tree. Consider that a switched parent form node p to p' because p' announced a tree that dominated over a's current tree (Alg. 3 line 16 – 18). a will switch to the tree announced by p' and transmit the change. When p receives the message from a reporting the change, if p is still in the same tree as a was, p will perform the same step as a (Alg 3 line 16 – 18). This is due to the fact that for p to be parent of a, a joined the tree to which p belonged to at some point in time. As such, if the tree that was adopted by a dominated over its previous tree, it will also dominate over the current tree of p, meaning that p will set a as its parent. This represents a possible case where the link between p and a does not change status (albeit both nodes switch to another tree).

Tree Repair/Recovery:

When a node fails, its neighbours will be eventually notified through the `NeighbourDown` notification (Alg. 3 lines 30 – 35). When a node a detects the failure of a peer p that was not the parent of a, no special measures are required except forgetting p. This is true since the tree topology, from the perspective of a, did not become compromised. However, if p was the parent of a, the tree topology has become incorrect and measures have to be taken to repair, or recover the tree.

In this case, a will attempt to locate a viable replacement for its parent (Alg. 3 lines 32 and 24 – 29). This is done by inspecting the state of all neighbours to find a suitable candidate that must be connected to the same tree, whose link is not currently part of the tree (i.e., status = `Passive`), and whose level in the tree is strictly below the level of a, as to avoid the creation of cycles. If a valid candidate is found, a updates its parent information

and its current level in the tree (this information will be propagated downwards in the tree on messages transmitted afterwards).

However, if no suitable candidate is found, a switches to the tree rooted on itself, triggering the process for establishing a new dominating tree as described above (Alg. 3 line 34) on node a. What this means is that a single fault may not affect the entirety of the network and only provoke a momentary localised instability, until a and the ones affected by a's state rejoin the dominating tree. The nodes that are affected by this are the ones that are in the subtree rooted in a that is currently a disconnected part of the tree due the fault of p.

When a transmits a message that reflects its new state, neighbours of a whose status is marked as `Passive` will not update their state as they are in the dominating tree. Neighbours of a whose status is marked as `Active` (children of a), will either adopt the tree of a (Alg. 3 line 12), effectively not changing links; or switch to the tree rooted on themselves, because their identifier is lower than a's identifier (Alg 3 line 14) and their tree is the one that should be the dominating tree, degenerating themselves to a similar state to the one of a. In that case, node a will eventually rejoin the dominating tree, by receiving information that the dominating tree is still up. By induction, so will the rest of the subtree rooted in a.

It is important to note that in the case of the failure of the root node the same mechanism explained before applies, as no other node can have a level of zero in that tree. This mechanism will, in this case, lead nodes to the process of establishing a new dominating tree. Furthermore, in these cases additional instability can happen and neighbouring nodes may detect direct cycles between them. In such scenarios both nodes will, similarly, switch to the tree rooted on themselves (Alg. 3 line 4). The failure of the root poses additional challenges that we discuss next.

Tree Stability:

When the root of the dominating tree fails, all nodes will have to converge to a new dominating tree. However, and since information does not propagate throughout the system instantaneously, it can happen that a node a discovers that the previous dominating tree is no longer correct, switches to a new tree, and afterwards receives a message from a different node that has yet to detect the failure of the previous tree. This can lead a to go back to the previous (incorrect) dominating tree. This happens because, with a high probability, the failed tree dominates over all other trees in the system (i.e., its identifier is lower than the identifiers of all other correct trees). This can generate instability in the process of establishing the tree, compromising the convergence of nodes to a new dominating tree.

To avoid this situation, we resort to the timestamps that are associated with each tree in the system. The timestamp of a tree is only incremented by the corresponding root node (Alg. 2 lines 22 – 23), and serves as a form of (multi-hop) heartbeat for that

tree. Messages exchanged among nodes carry the highest timestamp observed for the sender's current tree. Hence, observing increasing timestamps for a tree, indicates that the root node is still active. To avoid the situation described above, nodes only switch to a dominating tree if it is the first time they become aware of that tree (which is relevant for the bootstrap process) or if the message being processed by the node carries a timestamp greater than all previously observed timestamps for that tree.

Optional Optimisation:

Optionally, and to promote trees with lower heights, nodes can run an optimisation procedure to try and find a replacement for their current parent in a tree (Alg. 3 line 23). This is similar to the mechanism used to recover from faults, with the exception that the candidate must have a level lower than the current parent. This mechanism is not strictly required to ensure the correctness of the aggregation support tree.

4.4 Summary

In this Chapter we have presented the second main contribution of this work, the distributed aggregation protocol MiRAge. We have provided a brief overview of our protocol where we discussed the intuition of MiRAge and pointed out the main differences regarding similar solutions.

We presented the detailed operation of MiRAge that is divided into two parts: a first one that details how every node in the systems computes the aggregation result; and a second one that details how the tree topology employed by MiRAge is built and maintained by the collective efforts of the nodes in the system.

In the next Chapter we present the experimental evaluation of both Yggdrasil and MiRAge.

Chapter 5

Evaluation

In this Chapter we present our experimental work. At a higher level, we aim at demonstrating the applicability of our solutions, Yggdrasil and MiRAge, in realistic scenarios. To this end, we begin by describing our experimental methodology (Section 5.1), we follow this by presenting the tools that we have developed to help us in conducting our experiments (Section 5.2) and then, detail our experimental setup (Section 5.3).

Since our work presents two contributions, we also divide the rest of the Chapter into two parts. First we detail the experimental evaluation of Yggdrasil (Section 5.4). Then we present an extensive evaluation of MiRAge, where we compare it with relevant protocols from the state-of-the-art that serve as performance baselines to our own solution (Section 5.5).

5.1 Experimental Methodology

We intend to demonstrate the applicability of our solutions. To this end, we resort to implementations of our solutions and relevant baselines which execute in the physical devices that constitute the wireless ad hoc network. Our devices of choice are 24 Raspberry Pi's 3 - model B [73] equipped with an ARMv7 CPU, 1 GB of RAM, and a radio device capable of operating in ad hoc mode, executing a Linux-based operating system named Raspbian, in its 9.1 version with kernel version 4.9.41 – $v7+$. Each of the 24 devices is numbered from 1 to 24 to ease identification. These 24 devices form a wireless ad hoc network, over which, all communication is performed.

We use two experimental deployments of our fleet of 24 Raspberry Pis. A disperse deployment where we position each Raspberry Pi in different rooms across two hallways in our department building, such that each device can only communicate other nodes that are within the transmission range of their radio devices; and a dense deployment

where all nodes are placed in a single room, well within transmission range of each other.

The first deployments serves the purpose of showcasing the operation of our implemented protocols in a realistic environment. Although, there are some considerations to be made about this deployment. This scenario presents some adversary conditions, as there are multiple access points and other devices polluting the wireless medium as such, transmission ranges of each radio have been shown to vary. This means that some pairs of nodes, may be able to communicate with each other occasionally, producing instability in the operation of protocols. More importantly, our disperse deployment presents challenges regarding the controlled execution of experiences, namely the introduction of variable conditions, such as node/link failures or input value changes in the case of aggregation, in a surgical fashion. This is due to the fact that we have no knowledge of the pattern in which nodes communicate among them, as this may slightly vary from execution to execution.

These limitations are somewhat circumvented through the use of the dense deployment. Although in this deployment node are within transmission range of each other, we restrict the neighbouring relations of nodes by filtering out messages, effectively producing a logical multi-hop network that can be leveraged to perform more controlled experiences. This deployment also presents a high radio pollution in the wireless medium (since all nodes are clustered together) however, varying transmission ranges are not a problem.

We used Yggdrasil to implement all protocols that are considered in our experimental evaluation. Furthermore, to execute experiments in these two setting we developed two tools in Yggdrasil that we describe below.

5.2 Experimental Tools

The first challenge in executing experiments in a disperse deployment is to have all nodes start and stop the same experience (approximately) simultaneously. The solution relies on building a control network to start, stop, and introduce other dynamic features to the experience. Remember that nodes in the disperse deployment can only communicate through the established wireless ad hoc network as such, this control network must run atop this network. The second challenge is to enforce the topology in the dense deployment. However, with Yggdrasil's functionalities, achieving this second challenge is quite easy as we explain further ahead.

5.2.1 Yggdrasil Control Process

To address the first challenge, we developed the Yggdrasil Control Process, that is composed by three control protocols. A specialised *discovery protocol*, the *external input protocol* that allows commands to be issued by a client application (through TCP sockets), and the *core control protocol*. In addition to these protocols we have developed a set of simple

client applications that issue commands to the external input protocol and an application component that interprets the commands given by the Yggdrasil Control Process. We now explain each of these protocols in some detail.

Discovery Protocol: The discovery protocol designed to support the Yggdrasil Control Process is very similar to the simple discovery protocol that we used as a showcase example of Yggdrasil's provided abstraction to implement protocols in Chapter 3. The key difference is that, since we use TCP connections in the design of the *core control protocol*, we have created a discovery protocol that also propagates the IP address of the wireless interface (which is generated by DHCP through a local process) on the announcement messages periodically issued by the protocol. Moreover, this protocol was also enriched with support for special disable and enable operations, that respectively deactivate and activate the transmission of announcements, which is relevant to ensure that during experiments we minimise interference due to the activity of the Control Process.

External Input Protocol: The external input protocol fundamentally waits for incoming TCP connections on any network interface and processes user operations issued through these connections (through a client application). These operations, as stated before, can be requests to start or terminate an experiment, simulate a link failure, recover from a link failure, etc. Some of these commands can be tagged with a set of nodes identifiers, in which case only those nodes execute the requested action. Otherwise, the command is executed by all processes. Independently of the targets of the command, whenever this protocol receives a request from the user, it issues the command to the core control protocol for dissemination and processing (for some commands it also waits for a reply from the core control protocol that is redirected to the client).

Core Control Protocol: The core control protocol is the main protocol of the Yggdrasil Control Process. This protocol has two main goals. The first is to disseminate commands to all other Yggdrasil Control Processes in the experimental testbed (which are discovered by the discovery protocol although, we should note that this solution allows for multi-hop network configurations). This is achieved by a broadcast protocol, that operates on top of TCP connections, whose design is inspired by the Plumtree [53] protocol described in Chapter 2. This broadcast protocol, currently, also offers a mechanism to gather responses from processes that execute disseminated commands to produce a reply for the client. This mechanism however, is not fully stable in the current prototype. The second goal of the core control protocol is the (local) execution of commands issued by users.

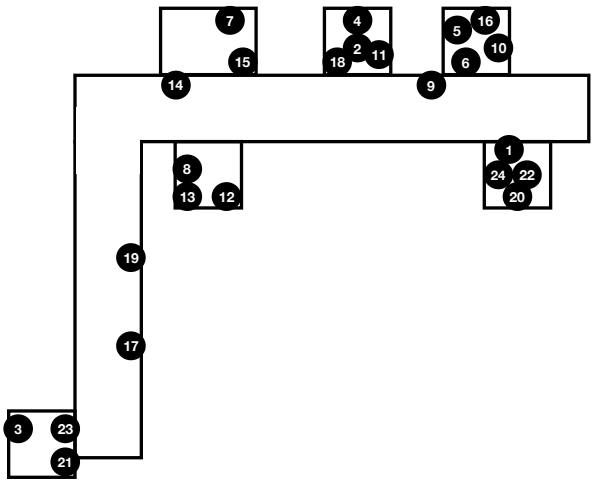


Figure 5.1: Distribution in Disperse Deployment.

5.2.2 Topology Control

To address the second challenge, we developed a protocol on Yggdrasil, called the Topology Control protocol. This is a very simple protocol given the abstractions and functionalities provided by Yggdrasil. The protocol operates as follows: it begins by reading two configuration files. The first file contains the MAC addresses of each device mapped to a number (the number of the device) and creates a MAC address database. The second file contains the numbers of the devices with which the local device will communicate. The protocol proceeds to send a request event type to the dispatcher protocol to ignore all incoming messages from the source nodes not contained in the second file. This enables the construction of a static multi-hop topology.

The protocol also has a request interface to change links. When the protocol receives one of these requests, it checks if the link (MAC address) is ignored or not by the dispatcher, and sends a request to dispatcher protocol either to stop ignoring or to ignore the source. This effectively enables the simulation of link failures and link recoveries.

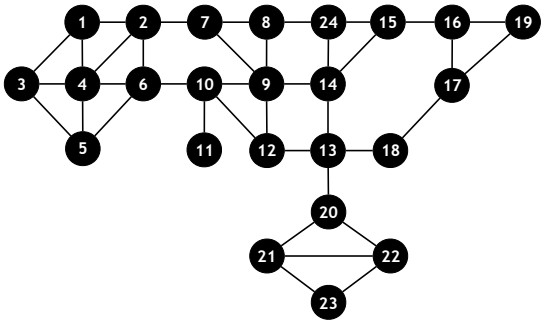


Figure 5.2: Overlay Topology in Dense Deployment.

5.3 Experimental Setup & Configuration

We now present how the devices are positioned in each setting. Figure 5.1 illustrates the distribution of nodes in space in the disperse deployment. Each of the hallways has approximately 30 meters. The overlay employed in the dense deployment is represented graphically in Figure 5.2.

Each device is configured to execute the Yggdrasil Control Process on boot. The Yggdrasil Control Process is then used to control all our experiments. This is done by having the Yggdrasil Control Process create a child process for the experience when the begin experience command is issued, and kill this process once the command to terminate the experience is issued. These processes log to disk all activity during experiments. Logs are processed offline.

We now detail our experimental work for each of our contributions that follow and make use of the previously described methodology and tools.

5.4 Yggdrasil: Experimental Evaluation

In this Section we report our experimental evaluation of Yggdrasil. This evaluation is divided in two parts. The first part provides additional insights on how useful and easy is to leverage Yggdrasil to implement distributed protocol for ad hoc networks. To this end, we discuss our implementation of the protocols used to motivate the design of Yggdrasil in Chapter 3: the broadcast protocol, B.A.T.M.A.N., and GAP. We also informally compare our B.A.T.M.A.N.'s implementation with a reference implementation of the protocol in C (that operates as a Linux daemon).

The second part is used to validate our implementations through practical evaluation. For this, we have conducted a performance evaluation of the implemented protocols using Yggdrasil to execute the protocols and simple test applications that exercise them in our disperse deployment.

5.4.1 Protocol Implementation

As to validate the advantage of leveraging Yggdrasil to produce implementations of wireless ad hoc distributed protocols, we have implemented the three case study protocols discussed in Chapter 3: a simple broadcast protocol based on a flooding mechanism, a simplified version of B.A.T.M.A.N. (V4) [68], and the aggregation protocol GAP [23]. With the exception of the broadcast protocol, which is the most simple, we have implemented the remaining protocols as described by the authors.

The broadcast protocol has about 200 lines of C code, that include a mechanism to garbage collect information about delivered messages. The protocol interacts with an application through request events, which hold a message to be disseminated throughout the network. This message is retrieved from the request's payload, delivered to the application and set for latter transmission with an arbitrary small delay, as to avoid

broadcast storms [69]. All delivered messages are stored in a list to ensure at most once delivery semantics. Messages that are in this list for a long enough period are eventually garbage collected by a periodic task dedicated to that purpose.

Our implementation of B.A.T.M.A.N. has less than 500 lines of C code. It has a number of simplifications regarding the original specification however, these do not affect the execution or correctness of the algorithm. In particular, our implementation of the protocol's sliding windows is represented as an array of shorts (rather than a bit array). Due to this, messages exchanged between processes are slightly larger than the original specified messages (however, they are still smaller than the maximum size of a network frame: 1,500 bytes).

An interesting aspect regarding this implementation is that we can compare it with a reference implementation, also in C, that operates as a Linux daemon. The code is publicly available at `https://www.open-mesh.org/projects/open-mesh/wiki`. This daemon implements version 3 of B.A.T.M.A.N. which, ignoring aspects related with the support for Internet gateways and external networks (that are ignored by our implementation), has the same logic as version 4 (that we implemented). We analysed the code of this implementation and discovered that it has approximately 7,000 lines. We inspected the code to identify and disregard all aspects related with interactions with the kernel, support for gateways and external networks, and other optimisations that we ignored in our implementation, and estimate that the core logic of the routing protocol has 2,000 lines of code. This represents an increase of about 4 times on the number of lines of code when considering our implementation. This difference is justified mostly due to the B.A.T.M.A.N. daemon having to deal with low level aspects such as, message serialisation, timer management, concurrency management, network interface management, among others. All of this is highly simplified by using Yggdrasil to implement the protocol.

Our implementation of GAP has less than 400 lines of C code. It implements the complete protocol as specified by the authors [23]. However, the authors propose different policies to deal with management of the values of neighbours. Our implementation uses the proposed default policy, which maintains all values from neighbours as long as they are not suspected of failure. Additionally, GAP assumes a companion discovery and fault detection mechanism. For this, we have implemented a protocol that provides both abstractions by piggybacking heartbeat messages on messages sent by GAP. This is the same protocol described in the previous Chapter for MiRAge.

All implementations took, independently, less than a week to be made, including debugging and verifying the correctness of the implementations. These observations, lead us to conclude that the Yggdrasil framework offers useful abstractions for the development of distributed protocols and applications for wireless ad hoc environments. Furthermore, and considering the difference in the size of our implementation of B.A.T.M.A.N. and the implementation of the B.A.T.M.A.N. daemon, we can also consider that these abstractions allow programmers to focus on aspects related with the design and operation of the protocol, involving less effort in the amount of code that has to be written (and

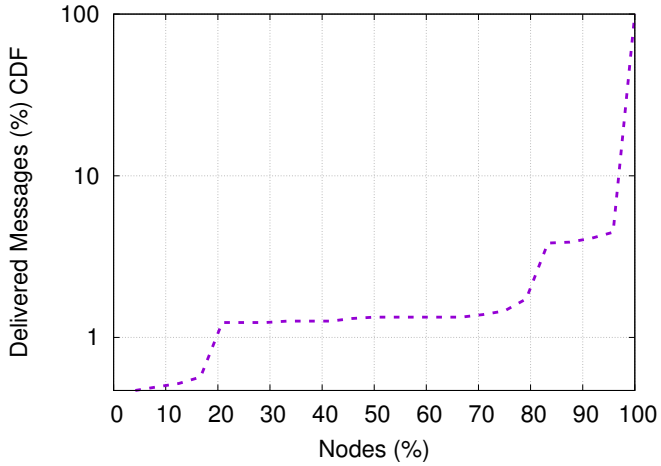


Figure 5.3: Broadcast Protocol: Delivery Ratio (CFD).

reviewed). Another interesting observation, is that this simplicity might be useful, for example, to use Yggdrasil as a support tool for lecturing advanced courses where wireless ad hoc networks and distributed protocols are discussed.

5.4.2 Performance Evaluation

Our experimental evaluation relies on simple demo applications which exercise the three classes of protocols, used to motivate the design of Yggdrasil. The goal is to assert that the implemented protocols have the expected (and) correct behaviour.

The experimental evaluation was conducted in our disperse deployment. The experiments for each protocol were executed for a period of 10 minutes. We collected logs in all devices for each experience, and then post-processed these logs after all experiences concluded. In the following we describe the experience and results obtained for each of the three case studies.

Broadcast Protocol:

For evaluating the performance of our developed broadcast protocol, which operates by flooding the network, we designed a simple test application that works as follows. Every two seconds, each process independently and randomly decides to broadcast a message with a probability of 50% by issuing a request to the broadcast protocol. Messages disseminated by the process carry the identifier of the process that generated the message and a unique monotonic identifier. All nodes register to a log all disseminated messages and all messages received.

In this evaluation we consider as performance metric the delivery rate of broadcast messages (i.e., the fraction of nodes that deliver a given message). Figure 5.3 reports our results in the form of a commutative distribution function (CDF) that shows the percentage of messages (in the y-axis, note that it is in logarithmic scale) in function of

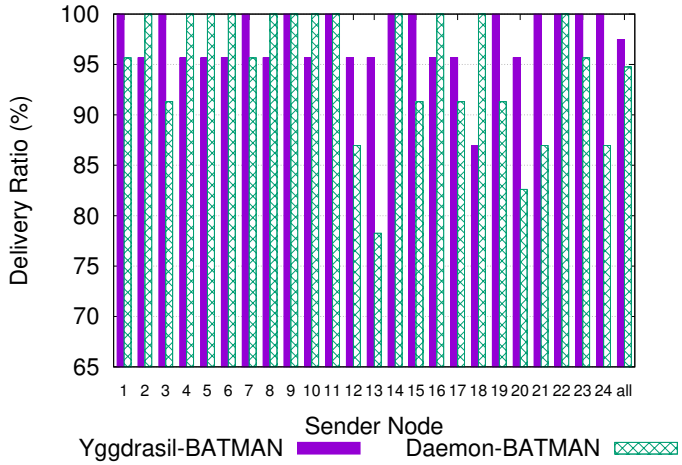


Figure 5.4: Routing Protocol: Comparison of Delivery Ratio per Node.

the fraction of nodes that delivered it (in the x-axis). The results show that the large majority of disseminated messages (in total there where approximately 3,500 messages disseminated) were delivered by every node. Only a small fraction, below 10%, of messages were delivered by few nodes. This however is not surprising, since collisions in the wireless medium still happen, despite our transmission delays that try to mitigate this effect. Overall, we consider that the protocol presents the expected performance and hence its implementation is correct.

Routing Protocol:

For validating the correctness of our implementation of the B.A.T.M.A.N. [68] protocol, we implemented a simple application akin to the one described above for evaluating the broadcast protocol. In this application each node decides to transmit a message with a probability of 50% every two seconds. When a node sends a message, it picks a destination for it randomly among the 23 possible destinations (we avoid nodes sending messages to themselves). Similarly to the previous experiment, we have logged all messages sent and received, and analysed the results offline. To further validate the behaviour of our implementation of the protocol, we wrote a second application (without resorting to Yggdrasil) that has exactly the same behaviour described previously, while using the B.A.T.M.A.N. daemon described previously. This allowed us to compare the performance of our own implementation with a reference implementation that does not resort to our framework.

In this experimental comparison we measure the fraction of messages sent by each node that were effectively received by their destination. Figure 5.4 reports the delivery ratio per individual node in our deployment, using each of the routing alternatives, as well as the average delivery ratio for all nodes (the final pair of columns labeled *all*). We note that in each experiment there were approximately 3,500 messages sent, with each node transmitting close to 150 messages. The results show that both implementations present

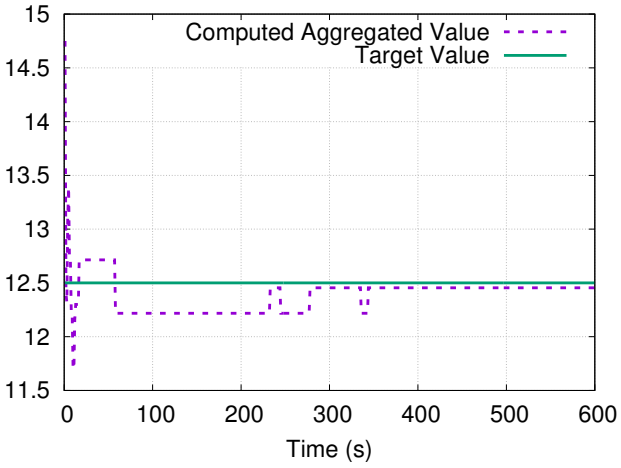


Figure 5.5: Aggregation Protocol: Precision of Aggregation Result.

very similar performance. In fact, when considering the average delivery ratio, our implementation surpasses the B.A.T.M.A.N. daemon by 3%. We note that this difference is most probably caused by interference in the wireless medium. In fact, certain nodes (such as node 20) have systematically lower delivery rates (i.e., messages sent by them reach their destination less frequently). This happens because in our disperse deployment, some nodes were positioned close to wireless access points or laptops that had WiFi radios active, and frequently polluting the wireless medium. Note that both implementations of the protocol route messages using unicast messages, which benefit from retransmission mechanisms implemented in the MAC layer (described in Chapter 2), which justifies why routing is less affected by collisions in the wireless medium when compared with the broadcast protocol.

Altogether, these results show that our implementation of B.A.T.M.A.N. has an equivalent behaviour to that of a reference implementation, with the difference that our implementation has 4 times less lines of code, being therefore easier to implement and maintain. This indicates that indeed, Yggdrasil offers the right and correct abstractions for developing distributed protocols for wireless ad hoc networks.

5.4.2.1 Aggregation Protocol:

Our experiments with the aggregation protocol (GAP [23]) are slightly different. We configured GAP to compute the average aggregation function. As input values to the aggregation process we used static values. In fact, we attributed as input value to each node its own identifier. This implies that the average of these input values is 12.5. Furthermore, we configured the GAP protocol to transmit periodic updates of its local estimate of the aggregated values (and tree management information) every two seconds (while the estimate is not stable). As discussed previously, GAP relies on a companion and generic protocol that discovers the neighbours of each node, and also emits suspect notifications when no message is observed from a particular node for a period of at least 10 seconds.

GAP requires a node to act as root node for the tree established by the protocol. We selected node 1 for this purpose. We wrote a simple application that exercises this protocol, where periodically (every two seconds), a request for the current aggregation estimate is issued to the aggregation protocol, waiting for a reply afterwards. This estimate was recorded to a log as soon as it was received.

Figure 5.5 reports the obtained results, only at the root node, where we depict the estimate of the aggregated value over time (we remind the reader that all experiments were conducted for a period of 10 minutes). For the convenience of the reader we also present a green solid line that represents the target (i.e., correct) value. This allows to infer the precision of the aggregation process in the root node. The results show that the protocol rapidly evolves to an estimate of around 12.7 units. Around the 50 second mark the root node starts to output a result of about 12.2 units until the 230 second mark. After this, the result slightly fluctuates for about 100 seconds, finally stabilising after the 350 second mark to a result of 12.45 units for the remainder of the experiment. Unfortunately, GAP, which is supposed to compute the precise aggregation result, is unable to do so in this setting. The reason is due to frequent loss of messages in a segment of the ad hoc network (as we discussed previously in the context of the previous protocols, some nodes where positioned in locations where there was prevalent wireless noise, or electromagnetic interference). This clearly shows the practical benefit of a framework such as Yggdrasil, that allows to run experiments on real settings. While the protocol is correct, noise in the environment makes it impossible for it to compute the correct value. This would be very hard to verify through simulation only.

In the next Section we present the experimental evaluation of MiRAge, where the implementation of GAP is also used. We will also provide further detail on the operation of GAP in the disperse deployment.

5.5 MiRAge: Experimental Evaluation

In this Section we present an extensive experimental evaluation of MiRAge comparing its performance against state of the art solutions for continuous aggregation. Furthermore, we evaluate MiRAge with and without the optimisation to promote trees with lower heights (the optimised version is labelled *MiRAge - Opt*).

All of the protocols presented here are implemented in Yggdrasil. In addition, all protocols make use of the same fault detector that also operates as a discovery protocol, described in the previous Chapter. The protocol is configured with $\Delta T = 1s$ and $K_{fd} = 10$. In practice this means that each node transmits an announcement with its own identifier every second, and that a node a is suspected to have failed by node b, when b is unable to receive an announcement from a for a period longer than 10 seconds.

We evaluate the protocols using the two deployments presented previously, a disperse deployment and a dense deployment with a logical network. We exercise the protocols in

varied conditions that include: fault-free, input value changes, and multiple node failures scenarios.

In our evaluation of MiRAge we use the baselines described in Chapter 2, which are the following:

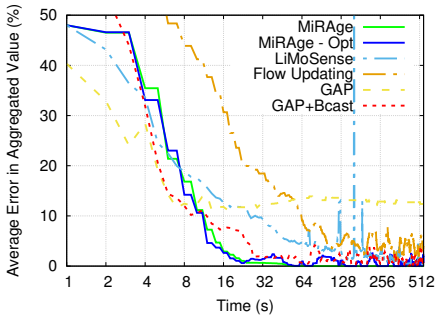
Flow-Updating [42], a protocol that computes the average function using an iterative approach; a version of **LiMoSense** [30] published by the authors, where counters maintained by nodes are never garbage collected. LiMoSense is a representative of the well known Push-Sum protocol [48] that can compute the sum, count, and average functions, enriched to ensure fault tolerance. In addition, LiMoSense requires a parameter q to be configured to a small arbitrary value so that weights are not excessively divided (i.e., $weight >= 2 * q$), we choose $q = 1/24 = 0.04166(6)$, which follows the authors guideline to be a small value; **GAP** [23] which is the protocol that mostly resembles MiRAge, being able to compute any aggregate function but unfortunately, unable to tolerate the failure of its static tree root. Since GAP does not enable every node in the network to obtain the computed aggregate result, we also developed a simple variant of GAP, that we named **GAP+Bcast** where the root of the tree broadcasts the currently computed result. This is achieved by having the result propagated in piggyback along the tree used by GAP, enabling all nodes to learn the result of the aggregation. Furthermore, both GAP and GAP+Bcast where configured to use the default policy to manage neighbours' values, as described previously. All protocols were configured to perform their periodic communication step every two seconds. In both GAP and GAP+Bcast experiments, the root was statically configured in all experiments to be node number 1, which will also be the root computed by MiRAge.

While MiRAge can easily be employed to compute any arbitrarily aggregate function, in our experiments every protocol was configured to compute the average. The initial input values of nodes are fixed, as described previously for GAP, being the numbers 1 to 24 attributed statically to each of the 24 Raspberry Pis. Hence, the average value, based on the initial input values, is 12.5. Each experiment was executed six times. Protocols were rotated between executions to amortise the effects of external and uncontrollable factors. Each execution was configured to have a duration of 10 minutes (600 seconds). To remove unexpected behaviours introduced by the environment, we chose the three best executions for each protocol. Results show averages of results obtained across the best executions. In the following, we discuss our experimental results.

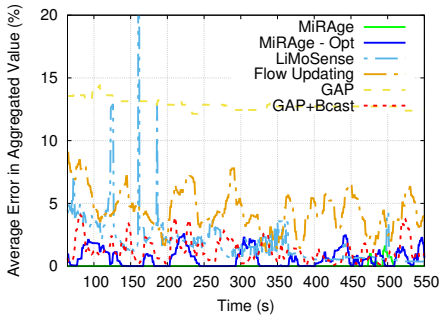
5.5.1 Experimental Results

In our experimental work we focus on the *Average Error in the Aggregated Value*, abbreviated *AvgErr*, that illustrates how far on average are all nodes from the correct aggregate result, being defined as:

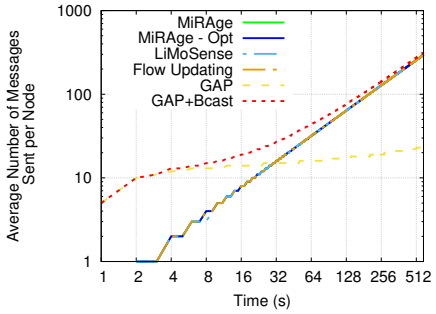
$$AvgErr = \frac{\sum_{i=1}^{n}(|Avg_{real} - Avg_i|)}{n \times Avg_{real}} \times 100$$



(a) Average Error in the Aggregated Value.



(b) Average Error in the Aggregated Value in the middle of the experiment.



(c) Total number of Messages sent by all nodes.

Figure 5.6: Disperse Deployment.

where n represents the total number of nodes, Avg_{real} is the current (and correct) aggregate result considering all input values, and Avg_i represents the current value computed by node i. We present this normalised for the real aggregation result. Intuitively, in a scenario where all nodes have computed the correct average, the *AvgErr* will be 0%, which is the ideal scenario. On the other hand, when the real average value is 12.5 and the computed average value by all nodes is 25, the *AvgErr* will be 100%; a computed average value of 50 would yield an *AvgErr* of 300%. Additionally, we also measure the total number of messages transmitted by all nodes in function of time. This is a measure of the communication overhead of each protocol.

5.5.1.1 Disperse Deployment

Figure 5.6 presents the results for our experiments in the disperse deployment. Where the nodes have been deployed as represented in Figure 5.1 previously.

Figure 5.6a reports the measured *AvgErr* across all nodes as the experience progresses. Note that the x-axis is in logarithmic scale, as the main point of this plot is to show the convergence of nodes towards the correct result, a process that is more noticeable at the start of the experiment. In addition, we provide the results obtained after the 64 second mark in Figure 5.6b to show the behaviour of protocols after they converged to a somewhat stable setting.

Results show that Flow-Updating is the protocol that converges more slowly towards the correct average, having an *AvgErr* lower than 10% only after more than one minute of execution has passed. Flow-Updating is also the protocol that suffers more from the instability of the environment, outputting results that have high fluctuations (around 5%) during the rest of the experiment, never being able to achieve the correct aggregation result. This is because Flow-Updating exchanges information with all neighbours. If the local neighbourhoods of nodes vary frequently, which they have shown to do so due to pollution in the wireless medium that cause variable transmission ranges, the local computed aggregates by each node will also vary significantly.

LiMoSense slowly converges towards a 5% *AvgErr* at the 70 second mark. After this point, it suffers from high spikes in the *AvgErr*, this is due to the detection of failures in some nodes, introducing high error in the aggregated value in compensation of the fault (we will elaborate more on this phenomena further ahead). Around the 200 second mark, the network stabilises, and the algorithm proceeds to slowly converge to a near perfect *AvgErr*, with the introduction of small amounts of variation. However, it never reaches the perfect (0%) *AvgErr*, this is because of the iterative nature of LiMoSense, and the fact that LiMoSense performs pairwise interactions, sending values to only one neighbour at each communication step. Hence, in order to reach the perfect *AvgErr*, the network would have to be stable for some amount of time.

GAP converges relatively fast towards an *AvgErr* close to 12% and stabilises around this value. This is expected, as GAP was not designed to provide all nodes in the system with the aggregate result. On the other hand, GAP+Bcast rapidly converges to an *AvgErr* of 2% unfortunately, it is never able to reach the correct result in all nodes. This is caused by instability in the network and because when GAP is able to converge to a stable tree, values still need to be propagated to the root. The root will propagate the computed result however, if the topology changes due to the loss of messages or the discovery of a new node, the root will start propagating different values leading to an asynchronous view of the result in the root and the rest of the network.

MiRAge shows the best performance in both variants, reaching an *AvgErr* of around 0.6% at the 30 second mark. MiRAge without the optimisation is able to reach the correct aggregation result around the 50 second mark however, MiRAge with the optimisation shows fluctuations of the *AvgErr* across the remainder of the experiment. This happens because the optimisation forces nodes to introduce additional instability trying to establish the lowest height tree. This will cause nodes to momentarily use connections that are less stable to propagate aggregated values.

Figure 5.6c shows the total number of messages sent over time for each protocol. The results show that both MiRAge variants, Flow-Updating, and LiMoSense have exactly the same cost. This is expected as all protocols were configured to exchange information at the same rate. GAP and GAP+Bcast issue more messages at the start of the experiment, as they send bootstrap messages to newly found nodes. GAP stops transmitting messages when values become stable, reducing GAP's communication overhead. However, this

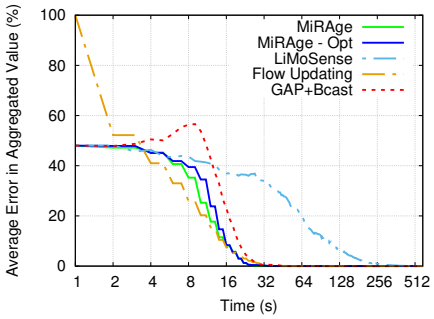


Figure 5.7: Average Error in the Aggregated Value in Fault-free Scenario.

could lead to missed updates due to message loss, and as such, GAP+Bcast was modified to cope with this issue by always transmitting messages, converging to a slightly higher cost than the remaining protocols.

In all experiments that we conducted, communication overhead always followed this pattern and hence, we omit those results from the following section.

5.5.1.2 Dense Deployment with Overlay

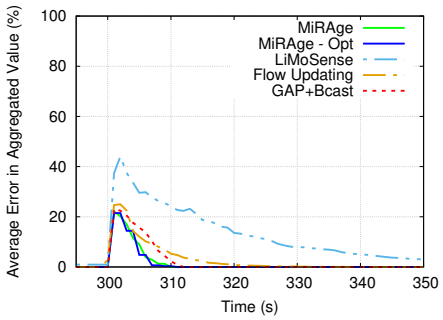
In this setting we have conducted multiple experiences. We note that while collisions in the wireless medium are highly probable, this setting is mostly shielded from other uncontrollable external factors as varying communication ranges that cause additional instability. We start by examining the behaviour of protocols in a fault free environment. Then we explore the effect of three dynamic aspects: *i*) input value change; *ii*) node failure; and *iii*) link failure. In experiments where we introduce dynamic aspects, these happen in the middle of the experiment (around the 300 seconds mark). Furthermore, we omit the results for GAP since it is unable to compute the aggregation result in all nodes of the system.

Fault-Free Scenario Figure 5.7 presents the *AvgErr* in a fault free execution for all protocols.

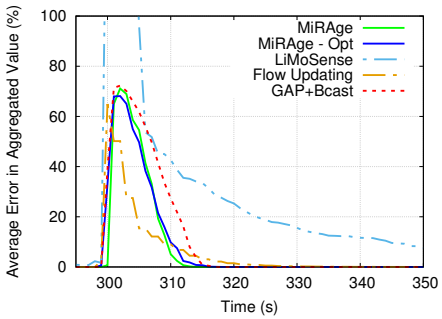
The results show that in this setting Flow-Updating quickly starts to converge towards a perfect aggregate value. Both MiRAge variants converge somewhat slower but reach an *AvgErr* of 0% slightly before. This happens due to the fact that MiRAge uses a deterministic tree topology to achieve convergence, whereas Flow-Updating relies on an iterative technique that iteratively converges towards the correct value. The optimised version of MiRAge has a slower start to convergence although, it reaches the perfect aggregate value at the same time as the non-optimised version. This again, is due to the establishment of the lowest height tree, introducing slightly more error in the beginning, but establishing a more efficient tree topology to propagate values.

GAP+Bcast converges towards the correct value across all nodes albeit, slightly slower than MiRAge and introducing a higher error in the process, this is because before the

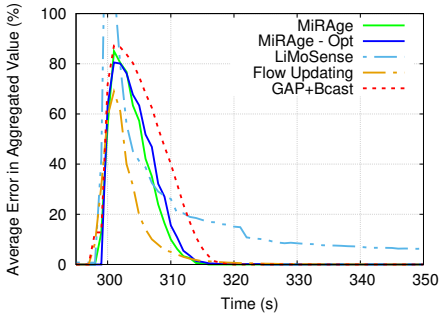
nodes have the result from the root, they will output their initial input values. LiMoSense is the slower protocol of our evaluation, converging to a good approximation of the value at the middle of the experiment. This is because, while the other protocols propagate values to multiple neighbours, LiMoSense, following the Push-Sum strategy, propagates values to only one neighbour at each communication step.



(a) 1 node.



(b) 12 nodes.



(c) 24 nodes.

Figure 5.8: Average Error in Aggregated Value with Dynamic Input Values at Different Number of Nodes.

Table 5.1: Input value change on 1 node, new aggregation result is 16.25 units.

Node	1	2	3	4	5	6	7	8	9	10	11	12
Old Value	1	2	3	4	5	6	7	8	9	10	11	12
New Value	1	2	3	4	5	6	**97**	8	9	10	11	12
Node	13	14	15	16	17	18	19	20	21	22	23	24
Old Value	13	14	15	16	17	18	19	20	21	22	23	24
New Value	13	14	15	16	17	18	19	20	21	22	23	24

Dynamic Input Values In these experiments we have introduced variations on the input value of different amounts of nodes in the system. We have conducted experiments where we modify the input value of 1, 12, and 24 nodes concurrently. The values to be changed were randomly generated between 1 and 200 units and attributed randomly to nodes. Table 5.1 represents the distribution of values for experiments with 1 input value

Table 5.2: Input value change on 12 nodes, new aggregation result is 51.833(3) units.

Node	1	2	3	4	5	6	7	8	9	10	11	12
Old Value	1	2	3	4	5	6	7	8	9	10	11	12
New Value	**186**	**52**	**79**	4	**111**	6	7	8	9	**5**	11	**103**
Node	13	14	15	16	17	18	19	20	21	22	23	24
Old Value	13	14	15	16	17	18	19	20	21	22	23	24
New Value	**46**	14	**134**	**125**	17	**87**	19	**137**	21	**16**	23	24

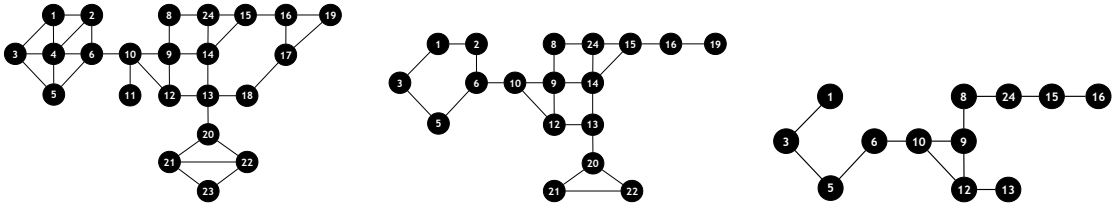
Table 5.3: Input value change on 24 nodes, new aggregation result is 112.125 units.

Node	1	2	3	4	5	6	7	8	9	10	11	12
Old Value	1	2	3	4	5	6	7	8	9	10	11	12
New Value	**39**	**69**	**132**	**84**	**87**	**135**	**88**	**136**	**112**	**161**	**119**	**53**
Node	13	14	15	16	17	18	19	20	21	22	23	24
Old Value	13	14	15	16	17	18	19	20	21	22	23	24
New Value	**179**	**101**	**186**	**176**	**37**	**88**	**57**	**103**	**152**	**154**	**161**	**82**

change. In these experiments, node number 7 (in grey) changed its input value from 7 units to 97 units, resulting in a new aggregation result of 16.25 units. Tables 5.2 and 5.3 represent the distribution of values for experiments with 12 and 24 input value variations, respectively.

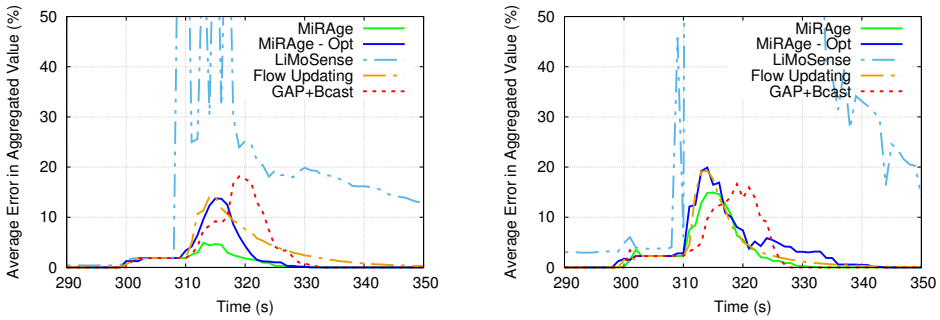
Results are summarised in Figure 5.8 and show consistent results for all experiments. LiMoSense is the protocol that is more susceptible to input value variations, whereas MiRAge, Flow-Updating, and GAP+Bcast all present somewhat similar results, being able to converge to the new aggregation result in less than 20 seconds. The reason why only these three protocols are able to cope in a timely fashion with the change of input values is nuanced. While LiMoSense must apply a transformation to new input value to compensate for the already transferred values that reflected the old input value, the remaining protocols' computation of the aggregation result directly depends on the (original) input value. This implies that as soon as the input value changes, nodes start propagating aggregation information that completely reflects the input value variation. Therefore, it suffices that messages propagate through the system to ensure that all nodes' result reflect the change.

Node Failures In these experiments we introduced a variable number of (concurrent) node crash faults and measured the impact on the *AvgErr*. In more detail, we introduced a number of concurrent node crashes that vary from 1, 6, and 12 nodes around 300 seconds in the experience. In these, we made sure that node number 1 was not selected to become faulty, as it was the appointed root node of GAP and GAP+Bcast and its failure would not be tolerated by the protocols. The resulting overlay topologies after the node crashes are depicted in Figure 5.9. In experiments with one node crash, node number 7 was assigned

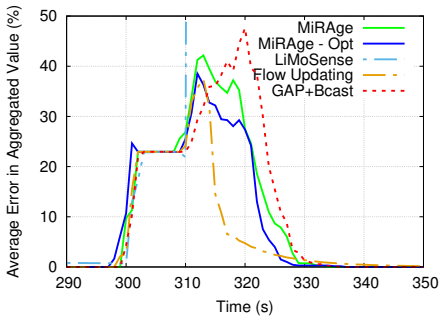


(a) 1 node. (b) 6 nodes. (c) 12 nodes.

Figure 5.9: Overlay Topology after node failures.



(a) 1 node. (b) 6 nodes.



(c) 12 nodes.

Figure 5.10: Average Error in Aggregated Value with Variable Number of Node Failures.

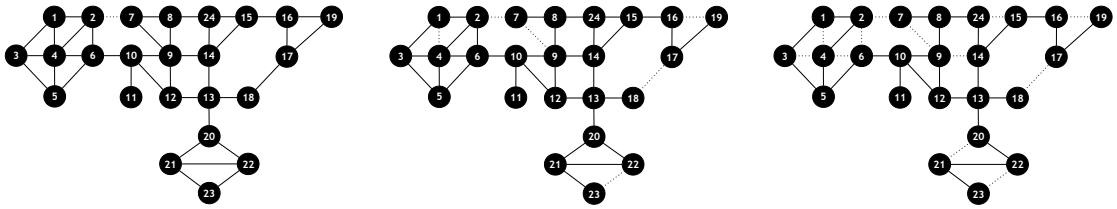
to become faulty resulting in the topology represented in Figure 5.9a. In experiments with more node crashes, we gradually increased the number of nodes assigned to become faulty, resulting in the topologies represented in Figures 5.9b and 5.9c for 6 and 12 node failures respectively.

Figure 5.10 reports the obtained *AvgErr* in these scenarios. In these experiments, results show that from the 300 to the 310 second mark all protocols present an increase in the *AvgErr*. This happens due to the implication of a node crashing. When a node crashes, its input value is no longer consider to the aggregation result hence, during this period the remaining nodes will still consider the previous aggregation value as being the correct one until the fault(s) is (are) detected. Once faults are detected, all protocols suffer from an increase in the *AvgErr* as they try to adapt to the changes.

LiMoSense is the protocol that suffers more from faulty nodes. This is because when a node is suspected of having failed, LiMoSense must apply a compensation mechanism to

remove the input values of the faulty nodes from the rest of the system. This compensation mechanism will introduce very high *AvgErr* as to effectively remove the values that the faulty node transferred, and to reintroduce the values that were transferred to the faulty node in the system. As we have mentioned before, this is a very slow process when compared to the other protocols.

The remainder protocols are able to adapt to the change in around 20 seconds, and present similar increases in the *AvgErr* upon the detection of the faults. MiRAge (without the optimisation) is the protocol that shows the lower increase of *AvgErr* in scenarios with less variation. This is due to fact that nodes that crashed have been considered to be leaf nodes of the established tree in some of the runs. This means that there are less topological changes to be made, introducing a lower *AvgErr*.

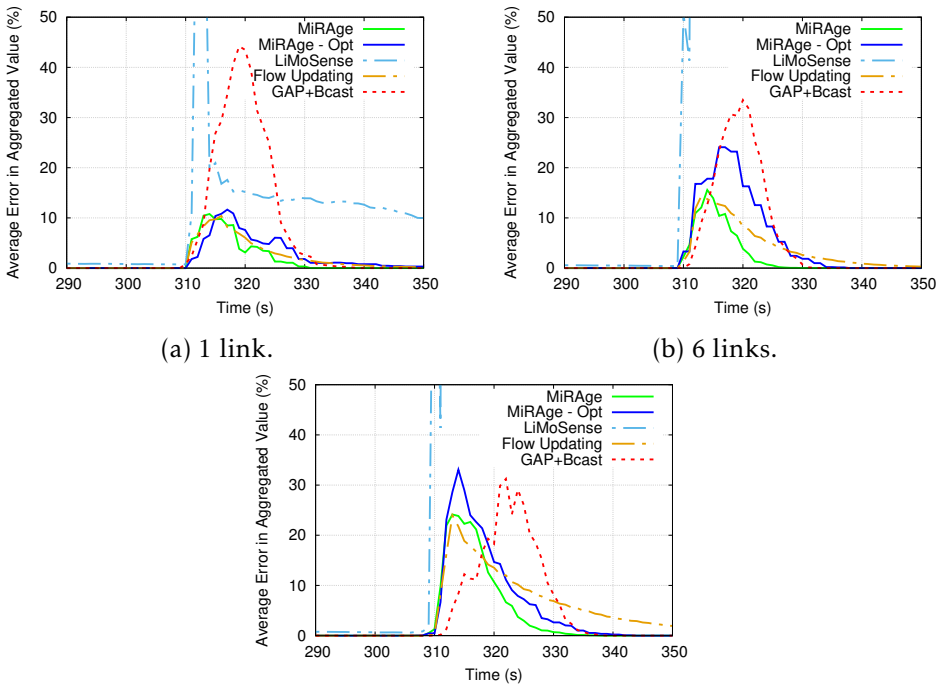


(a) 1 link. (b) 6 links. (c) 12 links.

Figure 5.11: Overlay Topology after link failures.

Link Failures In these experiments we introduced 1, 6, and 12 concurrent link failures around 300 seconds in the experience, where pairs of nodes become permanently unable to communicate. This experiment simulates the existence of obstacles or other external factors that might force the topology to permanently change. The resulting overlay topologies after the link failures are depicted in Figure 5.11 (dotted lines represent failed links). In experiments with one link failure, node number 2 and 7 were made unable to communicate, effectively removing their link from the topology in Figure 5.11a. For experiments with more link failures, we gradually increased the number of pairs of nodes that were made unable to communicate, resulting in the topologies represented in Figures 5.11b and 5.11c for 6 and 12 link failures respectively.

Figure 5.12 reports the measured *AvgErr* in these scenarios. The results show somewhat consistent results for LiMoSense with the previous experiments with node crashes, the key difference is that there are less nodes affected by the failure of the link and as such, the introduced variation is lower as there are less nodes applying the compensation mechanism. The remainder protocols show similar increases in the *AvgErr* with the exception of GAP+Bcast in the experiments with lower amounts of variation (1 and 6 link failures), specially in the experiments with 1 link failure. This is because in this experiment we failed a crucial link in the topology, forcing values to be propagated to the root by node 6 in the topology. As GAP+Bcast must broadcast the value from the root node to the rest of the system, the root will propagate the incorrect value, that considers the loss of a subtree



(a) 1 link.

(b) 6 links.

(c) 12 links.

Figure 5.12: Average Error in Aggregated Value with Variable Number of Link Failures.

(rooted in node number 7), until the correct values are propagated to the root through a different path. This phenomena does not happen in both versions of MiRAge, because nodes are not dependent on the root to propagate the aggregation result.

5.6 Summary

In this Chapter we have presented our experimental work over Yggdrasil and MiRAge. Regarding Yggdrasil, we have provided an informal comparison in terms of effort required to implement protocols. We also validated these implementations through an experimental evaluation. Overall, Yggdrasil has shown the capacity to save time regarding the implementation of multiple protocols that perform as well as reference implementations.

Regarding the evaluation of MiRAge, we have conducted an extensive evaluation with the relevant protocols from the state-of-the-art. This evaluation covered multiple scenarios, namely real-like scenarios by using our disperse deployment, and scenarios with dynamic aspects, such as node crashes, input value changes, and topological changes due to link failures. Our results show that MiRAge is able to be more robust and precise than the alternatives for continuous aggregation. We further reinforce that both GAP and GAP+Bcast are unable to sustain failures of the root node being therefore, less fault-tolerant than MiRAge.

Chapter 6

Conclusion and Future Work

Conclusion

The edge computing paradigm has emerged as a strategy to address the existing limitations of cloud infrastructures. As the number of devices increases, and as the digital data being produced and in need of processing scales to unprecedented numbers, cloud infrastructures are rapidly becoming unable to process and produce responses in reaction to all input data in a timely fashion. Edge computing can take various forms, as we have discussed in this work, and in this thesis we focussed on the challenging edge scenario where edge devices interact through wireless ad hoc networks hence, we strived to design solutions that enable computations to be performed in this particular scenario.

To properly understand how distributed protocols performed in wireless ad hoc networks, we intended to develop real implementations of distributed protocols that could effectively execute in real devices. However, as the study of the state-of-the-art revealed, there is a lack of suitable tools to allow efficient implementation and execution of protocols and applications in these settings. As such, we proposed the first main contribution of this work, the Yggdrasil framework. Yggdrasil proved to be a powerful tool that enabled us to develop and test distributed protocols operating on wireless ad hoc networks. As our experimental work revealed, Yggdrasil is capable of providing the adequate tools to develop correct protocols in a timely fashion.

To achieve our goal of performing computations in wireless ad hoc networks, we studied distributed data aggregation. We found that the state-of-the-art mostly focussed on achieving aggregation results in a single specialised node in the system and that input values where considered to be static. We however, focussed on a more challenging variant of the aggregation problem where input values may change over time. Hence, we studied protocols that could also cope with this dynamic aspect, i.e., solutions for the continuous

aggregation problem. This led to the second contribution of this work, the Multi Root Aggregation protocol, or simply MiRAge.

The concept of MiRAge is to establish a self-healing spanning tree to perform continuous aggregation. This concept is not entirely new however, MiRAge brings the advantage of being able to recover from the failure of the root of tree and enabling each node to compute the aggregation function, something that competitive alternatives based on tree topologies are not able to do. We have shown an extensive evaluation that shows that MiRAge is able to achieve faster convergence across all nodes in the network, while being robust to failures (both node crashes and link failures).

Future Work

As future work we are currently aiming for three main venues regarding Yggdrasil. The first venue we aim for is to improve the interfaces provided by Yggdrasil, as to provide improved intuitive programming interfaces and abstractions. The second venue we aim for is to further explore the dynamic management of protocols, designing solutions and applications that leverage this feature The last venue is related to the meaning of the name Yggdrasil. The name Yggdrasil comes from norse mythology and it symbolises the tree of life that connects the various realms that exist in the mythology. Consequently, we aim at generalising Yggdrasil to support more types of networks, namely IP networks, enabling the coexistence of distributed protocols that operate over IP networks and that operate over ad hoc networks. To reach such goal, we will integrate new low level libraries that can be used by the Dispatcher protocol to correctly identify in which interface should a message be sent. For this, the Dispatcher protocol must also be extended.

In relation to MiRAge, and distributed aggregation protocols in general, we also aim at two main venues for future work. The first is that our experimental evaluation showed that the optimisation of promoting lower height trees did not prove to bring many advantages. Specially in scenarios of high instability, the optimisation proved to lower the performance of the algorithm. Another more viable approach would be to promote trees with more reliable communication links to propagate the aggregated values. This could be achieved by employing similar techniques as employed by the routing protocol B.A.T.M.A.N..

A second venue to pursue is to reduce the communication overhead of MiRAge, having the algorithm stop sending messages under stable conditions (i.e., no changes in the input values). This however, compromises key properties of the algorithm that require messages to be continuously propagated, namely to deal with the failure of the root of the tree. A starting point would be to infer local stability in a node (e.g., the values that the neighbours of a node are sending are not changing), and to integrate fault detection in the algorithm itself, allowing for a more natural form of fault detection. Such efforts might lead to the design of a novel algorithm that builds on some of the key ideas and strategies introduced by MiRAge.

Bibliography

[1] I. Aad and C. Castelluccia. "Differentiation mechanisms for IEEE 802.11." In: *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*. Vol. 1. Apr. 2001, 209–218 vol.1. DOI: 10.1109/INFCOM.2001.916703.

[2] Y. Afek and A. Bremler. "Self-stabilizing Unidirectional Network Algorithms by Power-supply." In: *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '97. New Orleans, Louisiana, USA: Society for Industrial and Applied Mathematics, 1997, pp. 111–120. ISBN: 0-89871-390-0. URL: http://dl.acm.org/citation.cfm?id=314161.314193.

[3] Y. Afek, S. Kutten, and M. Yung. "Memory-efficient self stabilizing protocols for general networks." In: *Distributed Algorithms*. Ed. by J. van Leeuwen and N. Santoro. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 15–28. ISBN: 978-3-540-47405-0.

[4] M. Akter, A. Islam, and A. Rahman. "Fault tolerant optimized broadcast for wireless Ad-Hoc networks." In: *2016 International Conference on Networking Systems and Security (NSysS)*. Jan. 2016, pp. 1–9. DOI: 10.1109/NSysS.2016.7400690.

[5] I. F. Akyildiz and X. Wang. "A survey on wireless mesh networks." In: *IEEE Communications Magazine* 43.9 (Sept. 2005), S23–S30. ISSN: 0163-6804. DOI: 10.1109/MCOM.2005.1509968.

[6] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. "A survey on sensor networks." In: *IEEE Communications Magazine* 40.8 (Aug. 2002), pp. 102–114. ISSN: 0163-6804. DOI: 10.1109/MCOM.2002.1024422.

[7] I. F. Akyildiz, D. Pompili, and T. Melodia. "Challenges for Efficient Communication in Underwater Acoustic Sensor Networks." In: *SIGBED Rev.* 1.2 (July 2004), pp. 3–8. ISSN: 1551-3688. DOI: 10.1145/1121776.1121779. URL: http://doi.acm.org/10.1145/1121776.1121779.

[8] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications." In: *IEEE Communications Surveys Tutorials* 17.4 (Oct. 2015), pp. 2347–2376. ISSN: 1553-877X. DOI: 10.1109/COMST.2015.2444095.

[9] C. Ameixieira, A. Cardote, F. Neves, R. Meireles, S. Sargento, L. Coelho, J. Afonso, B. Areias, E. Mota, R. Costa, R. Matos, and J. Barros. "Harbornet: a real-world testbed for vehicular networks." In: *IEEE Communications Magazine* 52.9 (Sept. 2014), pp. 108–114. ISSN: 0163-6804. DOI: 10.1109/MCOM.2014.6894460.

[10] D. P. Anderson. "BOINC: A System for Public-Resource Computing and Storage." In: *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. GRID '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–10. ISBN: 0-7695-2256-4. DOI: 10.1109/GRID.2004.14. URL: http://dx.doi.org/10.1109/GRID.2004.14.

[11] N. Anwar and H. Deng. "Ant Colony Optimization based multicast routing algorithm for mobile ad hoc networks." In: *2015 Advances in Wireless and Optical Communications (RTUWO)*. Nov. 2015, pp. 62–67. DOI: 10.1109/RTUWO.2015.7365721.

[12] F. Araujo, L. Rodrigues, J. Kaiser, C. Liu, and C. Mitidieri. "CHR: a distributed hash table for wireless ad hoc networks." In: *Distributed Computing Systems Workshops, 2005. 25th IEEE International Conference on*. IEEE. 2005, pp. 407–413.

[13] Y. Azar, A. Z. Broder, A. R. Karlin, N. Linial, and S. Phillips. "Biased Random Walks." In: *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*. STOC '92. Victoria, British Columbia, Canada: ACM, 1992, pp. 1–9. ISBN: 0-89791-511-9. DOI: 10.1145/129712.129713. URL: http://doi.acm.org/10.1145/129712.129713.

[14] C. Baquero, P. S. Almeida, R. Menezes, and P. Jesus. "Extrema Propagation: Fast Distributed Estimation of Sums and Network Sizes." In: *IEEE Transactions on Parallel and Distributed Systems* 23.4 (Apr. 2012), pp. 668–675. ISSN: 1045-9219. DOI: 10.1109/TPDS.2011.209.

[15] M. Bawa, H. Garcia-Molina, A. Gionis, and R. Motwani. *Estimating Aggregates on a Peer-to-Peer Network*. Technical Report 2003-24. Stanford InfoLab, Apr. 2003. URL: http://ilpubs.stanford.edu:8090/586/.

[16] K. Birman and R. Cooper. "The ISIS Project: Real Experience with a Fault Tolerant Programming System." In: *Proceedings of the 4th Workshop on ACM SIGOPS European Workshop*. EW 4. Bologna, Italy: ACM, 1990, pp. 1–5. DOI: 10.1145/504136.504153. URL: http://doi.acm.org/10.1145/504136.504153.

[17] K. P. Birman. "Replication and Fault-tolerance in the ISIS System." In: *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*. SOSP '85. Orcas Island, Washington, USA: ACM, 1985, pp. 79–86. ISBN: 0-89791-174-1. DOI: 10.1145/323647.323636. URL: http://doi.acm.org/10.1145/323647.323636.

[18] K. P. Birman. "Replication and Fault-tolerance in the ISIS System." In: *SIGOPS Oper. Syst. Rev.* 19.5 (Dec. 1985), pp. 79–86. ISSN: 0163-5980. DOI: 10.1145/323627.323636. URL: http://doi.acm.org/10.1145/323627.323636.

[19] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. "Fog Computing and Its Role in the Internet of Things." In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. MCC '12. Helsinki, Finland: ACM, 2012, pp. 13–16. ISBN: 978-1-4503-1519-7. DOI: 10.1145/2342509.2342513. URL: http://doi.acm.org/10.1145/2342509.2342513.

[20] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. "Design and Implementation of a Routing Control Platform." In: *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 15–28. URL: http://dl.acm.org/citation.cfm?id=1251203.1251205.

[21] J.-Y. Chen, G. Pandurangan, and D. Xu. "Robust Computation of Aggregates in Wireless Sensor Networks: Distributed Randomized Algorithms and Analysis." In: *IEEE Transactions on Parallel and Distributed Systems* 17.9 (Sept. 2006), pp. 987–1000. ISSN: 1045-9219. DOI: 10.1109/TPDS.2006.128.

[22] Cisco. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update*. https://tinyurl.com/zzo6766. 2016.

[23] M. Dam and R. Stadler. "A generic protocol for network state aggregation." In: *self* 3 (2005), p. 411.

[24] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[25] I. Demirkol, C. Ersoy, and F. Alagoz. "MAC protocols for wireless sensor networks: a survey." In: *IEEE Communications Magazine* 44.4 (2006), pp. 115–121. ISSN: 0163-6804. DOI: 10.1109/MCOM.2006.1632658.

[26] T. Dillon, C. Wu, and E. Chang. "Cloud Computing: Issues and Challenges." In: *2010 24th IEEE International Conference on Advanced Information Networking and Applications*. Apr. 2010, pp. 27–33. DOI: 10.1109/AINA.2010.187.

[27] S. Dolev, A. Israeli, and S. Moran. "Self-stabilization of dynamic systems assuming only read/write atomicity." In: *Distributed Computing* 7.1 (1993), pp. 3–16. ISSN: 1432-0452. DOI: 10.1007/BF02278851. URL: https://doi.org/10.1007/BF02278851.

[28] S. C. Ergen and P. Varaiya. "TDMA scheduling algorithms for wireless sensor networks." In: *Wireless Networks* 16.4 (May 2010), pp. 985–997. ISSN: 1572-8196. DOI: 10.1007/s11276-009-0183-0. URL: https://doi.org/10.1007/s11276-009-0183-0.

[29] S. C. Ergen and P. Varaiya. "TDMA scheduling algorithms for wireless sensor networks." In: *Wireless Networks* 16.4 (2010), pp. 985–997. ISSN: 1572-8196. DOI: 10.1007/s11276-009-0183-0. URL: https://doi.org/10.1007/s11276-009-0183-0.

[30] I. Eyal, I. Keidar, and R. Rom. "LiMoSense: live monitoring in dynamic sensor networks." In: *Distributed computing* 27.5 (2014), pp. 313–328.

[31] E. C. Eze, S. Zhang, and E. Liu. "Vehicular ad hoc networks (VANETs): Current state, challenges, potentials and way forward." In: *2014 20th International Conference on Automation and Computing*. Sept. 2014, pp. 176–181. DOI: 10.1109/IConAC.2014.6935482.

[32] B. Ferreira. "Privacy-preserving efficient searchable encryption." Doctoral dissertation. Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, 2016.

[33] T. K. Forde, L. E. Doyle, and D. O'Mahony. "Ad hoc innovation: distributed decision making in ad hoc networks." In: *IEEE Communications Magazine* 44.4 (2006), pp. 131–137. ISSN: 0163-6804. DOI: 10.1109/MCOM.2006.1632660.

[34] F. C. Gaertner. *A Survey of Self-Stabilizing Spanning-Tree Construction Algorithms*. Tech. rep. Swiss Federal Institute of Tecnology (EPFL), 2003.

[35] C. Gomez, J. Oller, and J. Paradells. "Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology." In: *Sensors* 12.9 (2012), pp. 11734–11753.

[36] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN: 3540288457.

[37] G. T. C. Gunaratna, P. V.N. M. Jayarathna, S. S. P. Sandamini, and D. S. D. Silva. "Implementing wireless Adhoc networks for disaster relief communication." In: *2015 8th International Conference on Ubi-Media Computing (UMEDIA)*. Aug. 2015, pp. 66–71. DOI: 10.1109/UMEDIA.2015.7297430.

[38] I. Gupta, K. Birman, P. Linga, A. Demers, and R. Van Renesse. "Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead." In: *International Workshop on Peer-to-Peer Systems*. Springer. 2003, pp. 160–169.

[39] Z. J. Haas, J. Y. Halpern, and L. Li. "Gossip-based ad hoc routing." In: *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*. Vol. 3. 2002, 1707–1716 vol.3.

[40] P. Jacquet, P. Muhlethaler, T. Clausen, A. Laouiti, A. Qayyum, and L. Viennot. "Optimized link state routing protocol for ad hoc networks." In: *Multi Topic Conference, 2001. IEEE INMIC 2001. Technology for the 21st Century. Proceedings. IEEE International*. IEEE. 2001, pp. 62–68.

[41] P. Jesus, C. Baquero, and P. S. Almeida. "A Survey of Distributed Data Aggregation Algorithms." In: *IEEE Communications Surveys Tutorials* 17.1 (Jan. 2015), pp. 381–404. ISSN: 1553-877X. DOI: 10.1109/COMST.2014.2354398.

[42] P. Jesus, C. Baquero, and P. S. Almeida. "Fault-Tolerant Aggregation by Flow Updating." In: *Distributed Applications and Interoperable Systems*. Ed. by T. Senivongse and R. Oliveira. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 73–86. ISBN: 978-3-642-02164-0.

[43] P. Jesus, C. Baquero, and P. S. Almeida. "Flow updating: Fault-tolerant aggregation for dynamic networks." In: *Journal of Parallel and Distributed Computing* 78 (2015), pp. 53 –64. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2015.02.003. URL: http://www.sciencedirect.com/science/article/pii/S0743731515000416.

[44] D. Johnson, N. Ntlatlapa, and C. Aichele. "Simple pragmatic approach to mesh routing using BATMAN." In: *2nd IFIP International Symposium on Wireless Communications and Information Technology in Developing Countries (WCITD'2008)*. IFIP. 2008.

[45] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein. "Energy-efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet." In: *SIGARCH Comput. Archit. News* 30.5 (Oct. 2002), pp. 96–107. ISSN: 0163-5964. DOI: 10.1145/635506.605408. URL: http://doi.acm.org/10.1145/635506.605408.

[46] J. Kangasharju, J. Roberts, and K. W. Ross. "Object replication strategies in content distribution networks." In: *Computer Communications* 25.4 (2002), pp. 376 –383. ISSN: 0140-3664. DOI: https://doi.org/10.1016/S0140-3664(01)00409-1. URL: http://www.sciencedirect.com/science/article/pii/S0140366401004091.

[47] B. Karp and H. T. Kung. "GPSR: Greedy Perimeter Stateless Routing for Wireless Networks." In: *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*. MobiCom '00. Boston, Massachusetts, USA: ACM, 2000, pp. 243–254. ISBN: 1-58113-197-6. DOI: 10.1145/345910.345953. URL: http://doi.acm.org/10.1145/345910.345953.

[48] D. Kempe, A. Dobra, and J. Gehrke. "Gossip-based computation of aggregate information." In: *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* Oct. 2003, pp. 482–491. DOI: 10.1109/SFCS.2003.1238221.

[49] O. Kennedy, C. Koch, and A. Demers. "Dynamic approaches to in-network aggregation." In: *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. IEEE. 2009, pp. 1331–1334.

[50] H.-S. Kim, M.-S. Lee, Y.-J. Choi, J. Ko, and S. Bahk. "Reliable and Energy-Efficient Downward Packet Delivery in Asymmetric Transmission Power-Based Networks." In: *ACM Trans. Sen. Netw.* 12.4 (Sept. 2016), 34:1–34:25. ISSN: 1550-4859. DOI: 10.1145/2983532. URL: http://doi.acm.org/10.1145/2983532.

[51] J. Leitão. "Gossip-Based Broadcast Protocols." Master's thesis. Faculdade de Ciências da Universidade de Lisboa, 2007.

[52] J. Leitão, P. Á. Costa, M. C. Gomes, and N. Preguiça. *Towards Enabling Novel Edge-Enabled Applications*. Tech. rep. 2018. URL: https://arxiv.org/abs/1805.06989.

[53] J. Leitão, J. Pereira, and L. Rodrigues. "Epidemic broadcast trees." In: *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. IEEE. 2007, pp. 301–310. DOI: 10.1109/SRDS.2007.27.

[54] J. Leitão, J. Pereira, and L. Rodrigues. "HyParView: A membership protocol for reliable gossip-based broadcast." In: *Proc. of DSN'07*. IEEE. 2007.

[55] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. "TinyOS: An Operating System for Sensor Networks." In: *Ambient Intelligence*. Ed. by W. Weber, J. M. Rabaey, and E. Aarts. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 115–148. ISBN: 978-3-540-27139-0. DOI: 10.1007/3-540-27139-2_7. URL: https://doi.org/10.1007/3-540-27139-2_7.

[56] T. Liu and M. Martonosi. "Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems." In: *SIGPLAN Not.* 38.10 (June 2003), pp. 107–118. ISSN: 0362-1340. DOI: 10.1145/966049.781516. URL: http://doi.acm.org/10.1145/966049.781516.

[57] T. Liu, C. M. Sadler, P. Zhang, and M. Martonosi. "Implementing Software on Resource-constrained Mobile Sensors: Experiences with Impala and ZebraNet." In: *Proceedings of the 2Nd International Conference on Mobile Systems, Applications, and Services*. MobiSys '04. Boston, MA, USA: ACM, 2004, pp. 256–269. ISBN: 1-58113-793-1. DOI: 10.1145/990064.990095. URL: http://doi.acm.org/10.1145/990064.990095.

[58] F. Longo, D. Bruneo, S. Distefano, G. Merlino, and A. Puliafito. "Stack4Things: An OpenStack-Based Framework for IoT." In: *2015 3rd International Conference on Future Internet of Things and Cloud*. 2015, pp. 204–211.

[59] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. "TAG: A Tiny AGgregation Service for Ad-hoc Sensor Networks." In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2002), pp. 131–146. ISSN: 0163-5980. DOI: 10.1145/844128.844142. URL: http://doi.acm.org/10.1145/844128.844142.

[60] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. "TinyDB: an acquisitional query processing system for sensor networks." In: *ACM Transactions on database systems (TODS)* 30.1 (2005), pp. 122–173.

[61] R. Mahmud, R. Kotagiri, and R. Buyya. *Fog Computing: A Taxonomy, Survey and Future Directions*. Ed. by B. Di Martino, K.-C. Li, L. T. Yang, and A. Esposito. Singapore, 2018. DOI: 10.1007/978-981-10-5861-5_5. URL: https://doi.org/10.1007/978-981-10-5861-5_5.

[62] L. Massoulié, E. Le Merrer, A.-M. Kermarrec, and A. Ganesh. "Peer Counting and Sampling in Overlay Networks: Random Walk Methods." In: *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*. PODC '06. Denver, Colorado, USA: ACM, 2006, pp. 123–132. ISBN: 1-59593-384-0. DOI: 10.1145/1146381.1146402. URL: http://doi.acm.org/10.1145/1146381.1146402.

[63] H. Miranda, A. Pinto, and L. Rodrigues. "Appia, a flexible protocol kernel supporting multiple coordinated channels." In: *Proceedings 21st International Conference on Distributed Computing Systems*. Apr. 2001, pp. 707–710. DOI: 10.1109/ICDSC.2001.919005.

[64] H. Miranda, S. Leggio, L. Rodrigues, and K. Raatikainen. "A Power-Aware Broadcasting Algorithm." In: *2006 IEEE 17th International Symposium on Personal, Indoor and Mobile Radio Communications*. Sept. 2006, pp. 1–5. DOI: 10.1109/PIMRC.2006.254191.

[65] J. Mocito and L. Rodrigues. "Run-Time Switching Between Total Order Algorithms." In: *Proceedings of the Euro-Par 2006*. LNCS. Dresden, Germany: Springer-Verlag, Aug. 2006, pp. 582–591.

[66] G. Montenegro, C. Schumacher, and N. Kushalnagar. *IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals*. RFC 4919. Aug. 2007. DOI: 10.17487/RFC4919. URL: https://rfc-editor.org/rfc/rfc4919.txt.

[67] S. Motegi, K. Yoshihara, and H. Horiuchi. "DAG based in-network aggregation for sensor network monitoring." In: *International Symposium on Applications and the Internet (SAINT'06)*. Jan. 2006, 8 pp.–299. DOI: 10.1109/SAINT.2006.20.

[68] A. Neumann, C. Aichele, M. Lindner, and S. Wunderlich. *Better Approach To Mobile Ad-hoc Networking (B.A.T.M.A.N.)* Internet-Draft draft-openmesh-b-a-t-m-a-n-00. Work in Progress. Internet Engineering Task Force, Mar. 2008. 24 pp. URL: https://datatracker.ietf.org/doc/html/draft-openmesh-b-a-t-m-a-n-00.

[69] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu. "The Broadcast Storm Problem in a Mobile Ad Hoc Network." In: *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*. MobiCom '99. Seattle, Washington, USA: ACM, 1999, pp. 151–162. ISBN: 1-58113-142-9. DOI: 10.1145/313451.313525. URL: http://doi.acm.org/10.1145/313451.313525.

[70] C. E. Perkins and E. M. Royer. "Ad-hoc on-demand distance vector routing." In: *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA '99. Second IEEE Workshop on*. Feb. 1999, pp. 90–100. DOI: 10.1109/MCSA.1999.749281.

[71] C. M. Ramya, M. Shanmugaraj, and R. Prabakaran. "Study on ZigBee technology." In: *2011 3rd International Conference on Electronics Computer Technology*. Vol. 6. Apr. 2011, pp. 297–301. DOI: 10.1109/ICECTECH.2011.5942102.

[72] A. Raniwala and T. cker Chiueh. "Architecture and algorithms for an IEEE 802.11-based multi-channel wireless mesh network." In: *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.* Vol. 3. Mar. 2005, 2223–2234 vol. 3. DOI: 10.1109/INFCOM.2005.1498497.

[73] *Raspberry Pi 3 Model B - Raspberry Pi*. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. URL: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ (visited on 01/28/2018).

[74] M. G. Rubinstein, I. M. Moraes, M. E. M. Campista, L. H.M. K. Costa, and O. C.M. B. Duarte. "A Survey on Wireless Ad Hoc Networks." In: *Mobile and Wireless Communication Networks*. Ed. by G. Pujolle. Boston, MA: Springer US, 2006, pp. 1–33. ISBN: 978-0-387-34736-3.

[75] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. "Edge Computing: Vision and Challenges." In: *IEEE Internet of Things Journal* 3.5 (Oct. 2016), pp. 637–646. ISSN: 2327-4662. DOI: 10.1109/JIOT.2016.2579198.

[76] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. "Medians and beyond: new aggregation techniques for sensor networks." In: *Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM. 2004, pp. 239–249.

[77] B. Sidhu, H. Singh, and A. Chhabra. "Emerging wireless standards-wifi, zigbee and wimax." In: *World Academy of Science, Engineering and Technology* 25.2007 (2007), pp. 308–313.

[78] G. Simon, M. Maróti, A. Lédeczi, G. Balogh, B. Kusy, A. Nádas, G. Pap, J. Sallai, and K. Frampton. "Sensor Network-based Countersniper System." In: *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*. SenSys '04. Baltimore, MD, USA: ACM, 2004, pp. 1–12. ISBN: 1-58113-879-2. DOI: 10.1145/1031495.1031497. URL: http://doi.acm.org/10.1145/1031495.1031497.

[79] M. P. Spertus, S. Kritov, D. M. Kienzle, H. F. Van Rietschote, A. T. Orling, and W. E. Sobel. *Efficient backups using dynamically shared storage pools in peer-to-peer networks*. US Patent 7,529,785. May 2009.

[80] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A scalable peer-to-peer lookup service for internet applications." In: *ACM SIGCOMM Computer Communication Review* 31.4 (2001).

[81] A. Z. Tomsic, T. Crain, and M. Shapiro. "Scaling Geo-replicated Databases to the MEC Environment." In: *2015 IEEE 34th Symposium on Reliable Distributed Systems Workshop (SRDSW)*. Sept. 2015, pp. 74–79. DOI: 10.1109/SRDSW.2015.13.

[82] M. Torrent-Moreno, S. Corroy, F. Schmidt-Eisenlohr, and H. Hartenstein. "IEEE 802.11-based One-hop Broadcast Communications: Understanding Transmission Success and Failure Under Different Radio Propagation Environments." In: *Proceedings of the 9th ACM International Symposium on Modeling Analysis and Simulation of Wireless and Mobile Systems*. MSWiM '06. Terromolinos, Spain: ACM, 2006, pp. 68–77. ISBN: 1-59593-477-4. DOI: 10.1145/1164717.1164731. URL: http://doi.acm.org/10.1145/1164717.1164731.

[83] R. Van Renesse, T. M. Hickey, and K. P. Birman. *Design and performance of Horus: A lightweight group communications system*. Tech. rep. Cornell University, 1994.

[84] R. Van Renesse, K. P. Birman, and W. Vogels. "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining." In: *ACM transactions on computer systems (TOCS)* 21.2 (2003), pp. 164–206.

[85] P. Verissimo and L. Rodrigues. *Distributed Systems for System Architects*. Norwell, MA, USA: Kluwer Academic Publishers, 2001. ISBN: 0792372662.

[86] N. Wang, B. Varghese, M. Matthaiou, and D. S. Nikolopoulos. "ENORM: A Framework For Edge NOde Resource Management." In: *IEEE Transactions on Services Computing* (2017), pp. 1–1. ISSN: 1939-1374. DOI: 10.1109/TSC.2017.2753775.

[87] G. Werner-Allen, K. Lorincz, M. Ruiz, O. Marcillo, J. Johnson, J. Lees, and M. Welsh. "Deploying a wireless sensor network on an active volcano." In: *IEEE Internet Computing* 10.2 (Mar. 2006), pp. 18–25. ISSN: 1089-7801. DOI: 10.1109/MIC.2006.26.

[88] B. Wong and S. Guha. "Quasar: A Probabilistic Publish-subscribe System for Social Networks." In: *Proceedings of the 7th International Conference on Peer-to-peer Systems*. IPTPS'08. Tampa Bay, Florida: USENIX Association, 2008, pp. 2–2. URL: http://dl.acm.org/citation.cfm?id=1855641.1855643.

[89] J. Yeo, H. Lee, and S. Kim. "An efficient broadcast scheduling algorithm for TDMA ad-hoc networks." In: *Computers & Operations Research* 29.13 (2002), pp. 1793–1806. ISSN: 0305-0548. DOI: https://doi.org/10.1016/S0305-0548(01)00057-0. URL: http://www.sciencedirect.com/science/article/pii/S0305054801000570.

[90] S. Yi, C. Li, and Q. Li. "A Survey of Fog Computing: Concepts, Applications and Issues." In: *Proceedings of the 2015 Workshop on Mobile Big Data*. Mobidata '15. Hangzhou, China: ACM, 2015, pp. 37–42. ISBN: 978-1-4503-3524-9. DOI: 10.1145/2757384.2757397. URL: http://doi.acm.org/10.1145/2757384.2757397.

[91] J. Yick, B. Mukherjee, and D. Ghosal. "Wireless sensor network survey." In: *Computer Networks* 52.12 (2008), pp. 2292 –2330. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2008.04.002. URL: http://www.sciencedirect.com/science/article/pii/S1389128608001254.

[92] E. Ziouva and T. Antonakopoulos. "CSMA/CA performance under high traffic conditions: throughput and delay analysis." In: *Computer Communications* 25.3 (2002), pp. 313 –321. ISSN: 0140-3664. DOI: https://doi.org/10.1016/S0140-3664(01)00369-3. URL: http://www.sciencedirect.com/science/article/pii/S0140366401003693.