NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

VÍTOR HUGO MENINO

Bachelor's Degree in Computer Science and Engineering

A NOVEL APPROACH TO LOAD BALANCING IN P2P OVERLAY NETWORKS FOR EDGE SYSTEMS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
November, 2021

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

A NOVEL APPROACH TO LOAD BALANCING IN P2P OVERLAY NETWORKS FOR EDGE SYSTEMS

VÍTOR HUGO MENINO

Bachelor's Degree in Computer Science and Engineering

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

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
November, 2021

A Novel Approach to Load Balancing in P2P Overlay Networks for Edge Systems

This work was partially supported by FC&T through grants NOVA
LINCS (UID/CEC/04516/2013) and project NG-STORAGE (PTDC/CC-INF/32038/2017).

To life, the universe and everything.

Acknowledgements

I have received a great deal of support and assistance from my colleagues throughout the writing of this dissertation. Their impact in this work can not be underestimated. They may not have made things easier, but they surely made them better.

Especially, I would like to thank my supervisor, João Leitão, for giving me the chance of working under his supervision. Your expertise was invaluable and brought my work to a higher level.

I would also like to acknowledge Pedro Ákos Costa and Pedro Fouto, PhD students and researchers at NOVA LINCS. They were there in the hardest times, answering my distributed-systems-existential-questions. Thank you for your help in making my work a little better.

In addition, I would like to thank Nuno Morais, who just finished his own thesis. He developed the framework on which I did my first implementation of the solution prototype presented in this dissertation. Thanks for making the steep learning curve of Go a little less overwhelming.

Finally, I would like to thank everyone around me who had a somewhat positive impact in my life this last year—even those who have no idea about it. You made me who I am, and for that, I thank you with all my heart.

"For every complex problem there is an answer that is clear, simple, and wrong." (H. L. Mencken)

Abstract

Edge computing aims at addressing some limitations of cloud computing by bringing computation towards the edge of the system, i.e., closer to the client. There is a panoply of devices that can be integrated into future edge computing platforms, from local datacenters and ISP points of presence, to 5G towers, and even, multiple user devices like smartphones, laptops, and IoT devices. For all of these devices to communicate fruitfully, we need to build systems that enable the seamless interaction and cooperation among these diverse devices. However, creating and maintaining these systems is not trivial since there are numerous types of devices with different capacities. This resource heterogeneity has to be taken into account so that different types of machines contribute to the management of the distributed infrastructure differently, and the operation of the overall system becomes more efficient.

In this work, we addressed the challenges identified above by exploring unstructured overlay networks, that have been shown to be possible to manage efficiently and in a fully decentralized way, while being highly robust to failures. To that end, we devised a solution that adapts the number of neighbors of each device (i.e., how many other devices that device knows) according to the capacity of that device and the distribution of capacities of the other devices in the network, as to ensure that the load is fairly distributed between them and, as a consequence, improve the operation of other services atop the unstructured overlay network, for instance, reducing the latencies experienced when broadcasting information. This solution can be easily integrated into most existing peer-to-peer distributed systems, requiring just a slight adaptation to their membership protocol. To show the correction and benefits of our proposal, we evaluated it by comparing it with state of the art decentralized solutions to manage unstructured overlay networks, combining both simulation (to observe the performance of the solution at large scale) and prototype deployments in realistic distributed infrastructures.

Keywords: peer-to-peer systems, edge computing, unstructured overlay networks

Resumo

A computação de periferia visa abordar algumas limitações da computação em nuvem, trazendo a computação para mais perto do cliente. Há uma enorme variedade de dispositivos que podem ser integrados em futuras plataformas de computação de periferia, de *data centers* locais e pontos de presença de ISPs a torres 5G e até mesmo dispositivos de cliente, como *smartphones*, *laptops* e dispositivos IoT. Para que todos esses dispositivos comuniquem de forma proveitosa entre si, precisamos construir sistemas que possibilitem a interação e cooperação eficaz entre eles. No entanto, criar e manter esses sistemas não é trivial, uma vez que existem vários tipos de dispositivos com diferentes capacidades. Essa heterogeneidade de recursos deve ser levada em consideração para que diferentes tipos de máquinas contribuam para o gerenciamento da infraestrutura distribuída de forma distinta e a operação do sistema se torne mais eficiente.

Neste trabalho, enfrentámos os desafios identificados acima explorando redes sobrepostas não estruturadas, que se têm mostrado possíveis de gerenciar de forma eficiente e totalmente descentralizada, sendo altamente resistentes a falhas. Para tal, concebemos uma solução que adapta o número de vizinhos de cada dispositivo (ou seja, quantos outros dispositivos aquele dispositivo conhece) de acordo com a sua capacidade e a capacidade dos demais dispositivos da rede, de forma a garantir que a carga seja proporcionalmente distribuída entre eles e, como consequência, reduzindo as latências experienciadas por esses dispositivos. Esta solução pode ser facilmente integrada num sistema distribuído entre-pares existente, exigindo apenas uma ligeira adaptação ao seu protocolo de filiação. Avaliámos a nossa solução comparando-a com outras soluções descentralizadas de última geração, combinando simulação (para observar o desempenho da solução em grande escala) e emulação de protótipos em infraestruturas distribuídas realistas.

Palavras-chave: sistemas entre-pares, computação de periferia, redes sobrepostas não estruturadas

Contents

List of Figures

List of Tables

1 Introduction

Nowadays, doing all computation on the cloud, i.e., inside data centers distant from the end-user, is becoming insufficient mainly due to the high-latency access to these infrastructures and bandwidth limitations [30]. Edge computing [45] aims at addressing this and other limitations of cloud computing by bringing computation closer to the end clients.

There is a panoply of devices that can be integrated into future edge computing platforms, from local data-centers and ISP points of presence to 5G towers and even multiple user devices like smartphones, laptops, and IoT devices [30]. For all of these devices to communicate fruitfully, we need to build systems that enable the seamless interaction and cooperation among these diverse devices. However, creating and maintaining these systems is not trivial since there are numerous types of devices with different capacities. This resource heterogeneity has to be taken into account so that different types of machines contribute to the management of the distributed infrastructure differently, and the overall operation of these systems can become efficient [30].

In this work, we address this heterogeneity challenge by exploring unstructured overlay networks, that have been shown to be possible to manage efficiently and in a fully decentralized way, while being highly robust to failures. However, most of the work that has been conducted in this area [26, 52] deals with homogeneous networks, i.e., networks composed of nodes whose resources are considered mostly to be similar. Nonetheless, the system to manage devices with different capacities has to take into account their heterogeneous nature, therefore, it should be conceived thinking about heterogeneity.

1.1 Objective

In this dissertation, we address device resource heterogeneity in edge computing. More specifically, we address this problem by proposing a novel mechanism to dynamically compute and adjust the number of neighbors of each device at the overlay network layer, according to its local perception regarding its own capacity and an approximation of the distribution of capacities across all devices in the system. We do this at the overlay layer

since this is a component essential to the effective tracking of membership in large scale systems that has a significant impact across decentralized interactions that happen above it (both on other decentralized protocols/services or applications).

Devising an adequate solution as the one described above leads to some challenges. One of them is the need to keep the overlay proprieties sound when dynamically adapting (i.e., changing) the number of neighbors of each device. For example, we have to make sure that the network does not become partitioned when we adapt the number of neighbors of a given set of nodes. Another challenge that we face is the difficulty to infer the network resource distribution, given that each node does not have direct access (i.e., is not directly connected) to every other node and doing so would be unfeasible from a scalability standpoint.

1.2 Contributions

The main contributions generated from the work reported here are three-fold:

1. The design and implementation of ResEst—a decentralized algorithm whose objective is to estimate how powerful a node is in relation to the rest of the network. This is done through the decentralized computation of a histogram that approximates the distribution of capacities/resources across all existing nodes.

2. The design and implementation of Proteus—a decentralized algorithm whose objective is to compute the number of neighbors that the local node should strive to maintain, given its resources and an estimation of the distribution of the capacities of the nodes in the network (provided by ResEst).

3. An experimental comparison of our proposal with state of the art decentralized solutions by combining both simulation (to observe the performance of the solution at very large scale) and prototype deployments in realistic distributed infrastructures.

1.2.1 Research Context

This dissertation is done under the context of a research project conducted by NOVA School of Science and Technology in association with Protocol Labs [23]. Protocol Labs is a company that focuses on the research, development and deployment of decentralized distributed systems. Some of their work includes libp2p [22], Filecoin [21] and IPFS [4].

Part of the aforementioned research project consists on the design of a novel multi-level DHT. It is in this area that the contributions presented in this dissertation can provide value. By using an unstructured overlay optimized for heterogeneity, we can enable the creation of cliques of heterogeneous nodes in close proximity that can share load and replicate data while being minimally susceptible to failures and churn, due to the unstructured nature of the overlay that connects them.

1.2.2 Publications

Part of the work presented in this dissertation resulted in the following publication [37]:

- *ResEst — Algoritmo Distribuído para a Inferência de Recursos da Rede*

  Vítor Hugo Menino, Pedro Ákos Costa and João Leitão.

  Proceedings of the 12th Simpósio de Informática (INForum 2021), Lisbon, Portugal, September 2021.

1.3 Document Structure

The rest of the document is organized as follows:

- Chapter 2 starts by presenting the relevant concepts that lay ground for the work presented in the dissertation. More specifically, it explores the concepts of peer-to-peer, broadcast protocols, overlay networks and their key proprieties. It further delves into the problem that we are trying to solve and presents the currently available solutions, exploring their advantages, disadvantages, and discerning how they can contribute to our work. Although numerous solutions were found, none address the problem in the way that we aim to—by dynamically adjusting the number neighbors of each node.

- Chapter 3 starts by exploring the resource estimation problem, explaining the need for a resource estimation algorithm, and describing why it is not trivial to build one (in decentralized networks with partial views). Afterwards, we present our solution (ResEst) and validate it using simulation with one thousand and one million nodes. Finally, we conclude the chapter, summing up the main takeaways from our resource estimation contribution and presenting more use cases for it.

- Chapter 4 starts by presenting an overview of the devised solution to compute the number of neighbors that a node should have, displaying and describing the main behaviors of the algorithm (Proteus). Then, we delve into the first part of the solution, where we explore how to compute the percentile of the node (given a distribution of capacities of nodes). After that, we present and explain the second part of Proteus—the optimization function. Then, we expose the agnosticism and modularity of the presented solution, explaining how a membership protocol can adapt its number of neighbors upon receiving a notification from Proteus. And finally, with the base solution laid down, we propose an enhancement to our algorithm (in order to improve stability) and conclude the chapter with a summary of its contributions.

- Chapter 5 reports on the experimental evaluation of our work. It starts by describing the prototype that we built to evaluate our solution, as well as the prototype we

built to serve as a benchmark. Then, we describe the framework used to implement our prototype and cover the network constraints added to each node. Afterwards, we detail the experimental settings used—both on the hardware, and the software parameterization. Later on, we evaluate ResEst, and after evaluating ResEst in isolation, we evaluate Proteus using ResEst configured with the set of parameters that yield a better execution (found when testing ResEst in isolation). Finally, we deduce the parameter values that result in the best performance for our solution, exposing such results and summarizing the evaluation process, taking conclusions over the validity and performance of ResEst and Proteus in realistic scenarios.

- Chapter 6 concludes this dissertation. It starts by summarizing the work performed on this thesis and its applicability, and then closes-off with enumerating possible optimizations that can be done to our solution as future work.

2

Related Work

In this chapter, we are delving into the relevant concepts that lay ground for the elaboration of this dissertation, while presenting and analyzing related work. The chapter is structured as follows: in Section 2.1 we introduce key concepts of peer-to-peer systems; in Section 2.2 we present services that can be implemented on top of peer-to-peer systems; in Section 2.3 we delve into application level decentralized broadcast protocols, its problems and performance metrics; in Section 2.4 we introduce the concept of overlays, the different architectures, its proprieties and protocols that implement them; in Section 2.5 we use the concepts explained in previous sections to explore how the heterogeneity problem can be tackled; in Section 2.6 we explore the existing algorithms that could be, at first sight, used for resource estimation, arriving to the conclusion that they do not suffice our goals; finally, in Section 2.7 we reflect on the contributions and key insights that the solutions previously presented can provide to our work.

2.1 Peer-to-Peer Systems

There are two main architectures for distributed computer systems: centralized and decentralized. Both architectures have advantages and disadvantages and their usefulness depends on the use case of the system.

Centralized distributed systems rely on a single logical component (usually named server) that has a special role in the system and that provides services to the other nodes (i.e., computers, usually called clients in this context) of the system. In such architectures, clients usually only interact with the server and never directly among them. Usually, these systems manage to achieve great performance, since its nodes delegate all coordination tasks (e.g., deciding which node is the current leader) to the single central server. On the other hand, these systems are highly dependent on the availability of the coordinator, i.e., if the central server fails, the whole system stops—which is a big disadvantage of centralized systems [28]. In addition, centralized systems have scalability problems [28]—if the network is large, it might be impossible for the central server to manage every client in the network (i.e., every other node), since they can be numerous, distant from the central

node, and joining and leaving the network on a very frequent basis. These limitations led to the appearance of systems implementing a different and decentralized architecture, i.e., decentralized systems.

Decentralized systems, on the other hand, do not rely on a single component that has a special role on the operation of the whole system. In such architectures, nodes communicate between themselves directly to coordinate for executing the system logic: all information exchange and coordination is performed directly among the participants of the system. The biggest advantage of decentralized systems is that there is no single point of failure, which means that numerous nodes could fail and the system would still continue to operate [42]. In addition, they are more scalable and there is no single component that is in total control of the system [42]. However, decentralized systems often exhibit poorer overall performance than centralized systems. A popular example of decentralized systems are peer-to-peer systems [44], in which clients communicate between themselves and there is no central server.

One might ask—if centralized systems are theoretically faster, then why use peer-to-peer decentralized systems? As mentioned above, decentralized systems are usually more available (i.e., reliable against failures), more scalable, and make shared control over the system possible. There are several practical cases where these features might be important—specially the last one. One of these cases is a large-scale blockchain system (e.g., Bitcoin [38]) where, by design, the participants do not need to have to trust each other in order for the network to operate, and one node failing should not impact the validity of the system.

2.1.1 Peer-to-Peer Networks and Edge Computing

Peer-to-peer networks are regaining some popularity due to the emergence of edge computing [45]. Edge computing encompasses every computation done outside the cloud, i.e., outside the logical central computational unit, and closer to the fringe of the system where data is mostly produced and consumed. The rationale behind edge computing is moving the computation closer to the client so that, instead of all data generated by the clients being shipped to be processed in the cloud (data-centers), some of that data can be processed on intermediary hardware (possibly including the client's device), in order to provide better latency, reliability, and scalability [30], while avoiding saturating network links connecting clients to the cloud data-centers.

In the last years, two particular models emerged within the edge computing realm that have received significant attention—fog [33] and mist computing [15]. They are both seen as specific instances of, the more general, edge computing model [30]. Fog computing aims to improve the overall performance of IoT applications by installing servers physically close to sensors that generate large amounts of data. These servers can preprocess data before sending it to the cloud data-centers, resulting in less load on these and on the network infrastructure in general. Beyond that, the fog servers can also

reduce the latency observed by the clients by doing time-sensitive processing and quickly replying to the client with any relevant time-sensitive information (e.g., an alarm). The second model, mist computing, is an evolution of fog computing, where not only servers are installed nearby the clients and used for computation, but the clients themselves are also used to perform data-filtering over generated data. Using the clients' hardware to do some computational work alleviates the load imposed on edge and cloud servers (and on the network), resulting in a better overall system performance and resource usage.

Peer-to-peer networks can be useful in edge computing in different ways. Firstly, because there are a lot of different nodes (i.e., computers) in the edge environment, managing them centrally becomes a daunting task, which is hard to perform in a timely fashion [7]. This is difficult mainly because in a centralized management system, the central component would be saturated with messages from all the nodes in the system. In addition to that, it is also difficult because the network that encompasses those nodes can be wide, i.e., we can have nodes in remote locations, which makes it infeasible to timely manage changes in the configuration of the system. Using peer-to-peer networks to implement decentralized control solves these problems because the nodes usually only communicate with other nodes that are in their virtual vicinity (i.e., the set of nodes that they know) [28], thereby not saturating a coordinator nor presenting unnecessary delay in the management communications.

One might suggest that for not saturating the coordinator we could have different hierarchies of coordination. This also constitutes a centralized architecture, since there is also one top-level coordinator that manages the whole network through the other coordinators [55]. However, such a solution does mitigate both problems mentioned above—node saturation and communication delay. Nonetheless, even such an optimized centralized solution ends up suffering from other limitations. One of them is regarding the reliability—with a centralized system, we have a single point of failure (the top coordinator). This issue is avoided in most peer-to-peer decentralized systems because network management is a responsibility shared across all peers (i.e., nodes). The other problem is that in edge computing, nodes are usually not owned by the same entity—we can have nodes owned by different companies and individuals. That said, it is not wise to allow a single coordinator (from one of those companies or individuals) to manage the entire network, because that would involve having to trust that node for the whole system to keep operating.

2.1.2 Peer-to-Peer Architecture

When it comes to the system architecture of a typical peer-to-peer system, each of the nodes runs a stack of (usually) three protocols (which can be perceived as layers in an abstract peer-to-peer architecture): membership layer, service layer and application layer [28]. The layers within each node communicate with each other (to support the execution of the system logic) locally, and communicate with the remote equivalent layer

in other nodes using message passing mechanisms (i.e., by sending messages through the network). Figure 2.1 illustrates the presented abstract peer-to-peer architecture, which is explained in detail below.

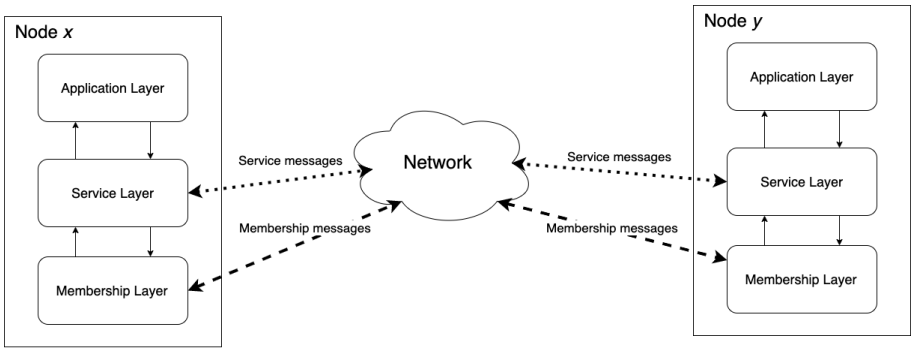


Figure 2.1: Peer-to-peer node layered architecture.

The membership layer is responsible for maintaining the neighbors of each node, i.e., the other nodes in the system that each node knows about and with whom it exchanges information directly. This layer communicates with other nodes' membership layers in order to maintain each node's local and partial view of the system membership (i.e., which neighbors each node has) [28]. This layer is going to be analyzed in depth on Section 2.4.

The service layer is responsible for providing higher level services to the application layer taking advantage of the membership layer. In other words, this layer uses the local view of the system provided by the membership layer to send messages to the node's neighbors in order to perform some function for the application layer (e.g., broadcast a message or locate some particular node) [28]. This layer is going to be analyzed in depth on Section 2.2.

The application layer uses the service layer (and potentially the membership layer) to implement a distributed application. It is in this layer that the application logic resides [28]. For example, if we were to implement a distributed file system where each node hosts part of the contents of the file system, this layer would be responsible for the application logic which involves exposing the operations over files and directories but not operations such as data/message dissemination (because that is typically implemented at the service layer) and membership management (because that is a responsibility of the membership layer).

2.2 Peer-to-Peer Services

There are numerous services that can be implemented at the level of the service layer [29, 54, 43, 14]. As mentioned above, these services provide functionalities for the application layer and usually involve the exchange of data (in the form of messages) between the nodes.

One of those services is data streaming [54]. In this type of service, a flow of data is sent by one node through the other nodes in order for the multiple receivers to be able to receive that flow of data and feed the application layer with it. Another common service is shared storage [14], which aims at sharing (e.g., splitting) data between the nodes for the client application to enable sharing the access to files by multiple parties. One concrete example of an application that can be built on top of it is a shared file system [21, 53].

The service layer can also be used to implement routing solutions [43]. These solutions aim at discovering and providing efficient and up-to-date routing paths for the messages to be sent through, among individual parties. This enables point-to-point exchange of information at the application layer.

Lastly, the service layer can implement application-level broadcast solutions [29, 40, 6], which aim at disseminating messages across all nodes of the network. We are going to focus on decentralized broadcast protocols in Section 2.3.

One of the building blocks that is used to implement many of the services mentioned above is gossip [28]. Gossip is a class of decentralized protocols that are many times employed to disseminate information through nodes with a configurable probabilistic assurance [29]. Due to their probabilistic nature, gossip protocols usually rely on parameters that enable the control over their probabilistic guarantees and cost.

In a nutshell, gossip-based dissemination works as follows. When a process (i.e., the software that a node runs) wants to disseminate a message, it picks t (the fanout) random nodes from its neighbors and sends the message to those nodes. When a process receives a message for the first time, it simply repeats the process (possibly avoiding sending the message back to the node from which it received the message) [28].

In their most simple form, gossip algorithms have two parameters: the fanout and the maximum rounds [29]. The fanout [29] consists on the number of neighbors that each neighbor disseminates a message to. The fanout has an impact on the probability that every node receives the message. For this probability to be high, the fanout should be $t >= ln(n)$, t being the fanout and n being the total number of nodes in the network [9]. That said, the higher the fanout, the more likely it is that all nodes receive (and then deliver to the application layer) the message being disseminated. On the other hand, a higher fanout results in more traffic on the network. Thus, the chosen fanout for the gossip implementation depends on the goals for the system, i.e., what is most important—high probability of total dissemination (i.e., all nodes receiving the disseminated message) or lowering the bandwidth consumption.

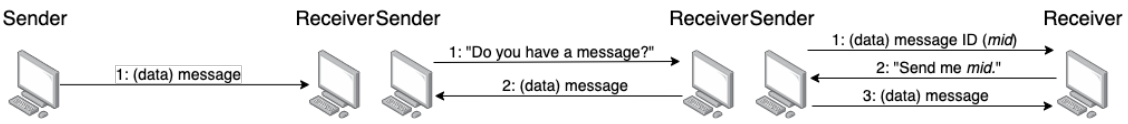
The maximum rounds parameter corresponds to the maximum number of times that a message can be retransmitted by nodes [29]. One way to control this is by using a round-number that is carried by messages. When a message is first transmitted (from the original sender), it carries a round-value of zero. This value is increased every time that a node retransmits the message. When a node receives a message for the first time and its round number is equal or higher than the maximum rounds parameter, the node does not retransmit the message. This round-value can be seen as a TTL (Time-to-live) of

the message being disseminated. Many simple implementations of gossip dissemination protocols set this value to infinite.

Gossip interactions among pairs of nodes can be conducted in different ways, being the most popular the following: eager push gossip (also known as push gossip), pull gossip, and lazy push gossip [29]. Eager push gossip [29] consists in the original behavior as described previously, where a node sends the payload message to t (fanout) random neighbors once it receives it for the first time. This is the faster communication mode, although it can be expensive in terms of generated traffic, specially if the payload of messages is big—due to the redundant nature of gossip.

An alternative to this is to employ pull gossip [29], where the nodes which do not yet have the payload message ask t random nodes for it. If one of those nodes has it, then it sends the payload message back to the node who asked. This solution produces less traffic than eager push gossip if the payload messages are big, because the number of redundant payload messages being transferred is not as high. However, if the payload messages are small (specifically, if they are smaller than the "ask" messages), then this solution ends up creating more traffic due to the fact that now we have—in practice—two messages being transferred between each pair of nodes (the payload message and the "ask" message). Pull gossip is also usually the slowest solution, since the nodes that do not have the payload message are the ones who have to ask for it to get it, and since they do not know when there are new messages, they end up setting a periodical timer to ask for them—a timer that cannot be very frequent because otherwise the traffic could become excessive, and it would not be worth it to use a pull approach.

Finally, the other—more sophisticated—communication method is lazy push gossip [29]. In this method, the nodes that receive the payload message for the first time send a payload message ID (*mid*) to t random nodes. If those nodes do not possess that payload message yet, they reply asking for the payload message. Then, the payload message is sent to them. This communication mode is faster than pull gossip and if the payload messages are big, it is faster than eager push gossip too. However, it is the communication mode with more communication steps, which adds complexity to the system. In Figure 2.2, we can see a diagram illustrating each of the communication modes explained above.



(a) Eager push gossip. (b) Pull gossip. (c) Lazy push gossip.

Figure 2.2: Diagrams of three different gossip communication modes.

There are numerous variants of gossip—all with different advantages and disadvantages. All these variants use different fanout and/or communication modes, some even combining multiple communication modes at once [25]. The simplest gossip strategy is

flood [28]. In this strategy, eager push is used as communication mode and the fanout is always the same as the number of neighbors of each node. Flood produces a lot of traffic, but it assures the highest probability (out of all strategies) for every node to receive the disseminated message (in fact it suffices that the graph denoted by the closure of the partial view of nodes is connected). Other commonly used strategy is Anti-Entropy [41]. It uses a fanout of 1 and pull as the communication mode (executed by every node in the system periodically). This strategy has been used; for instance, to replicate non time-sensitive data in distributed databases [41]. While being somewhat slow, it does not use a lot of bandwidth (due to the low traffic generated). The random-walk strategy [28] also uses a fanout of 1 but uses eager push as the communication mode (usually with a maximum number of retransmissions per message, i.e., with a maximum rounds parameter set to some value below infinite). This strategy is useful to search (somewhat popular) content in the network. For example, if a node wants to search for a file in the other nodes of the network, it can send a message in a random-walk so that if the message reaches a node that has that file, that node replies to the searcher node saying that it has the file.

Some of these variants, and their combinations, can be used to implement different gossip-based services that are going to be thoroughly studied in this work [6, 40]. In the next section, we are going to discuss some of these.

2.3 Broadcast

The broadcast problem can be synthesized as follows: a process needs to transmit the same message m to n other processes (where n is every process in the system including itself). There are several different formulations of this problem. Some of these have weaker guarantees, like the best-effort broadcast (e.g., IP Multicast [8]). Others have strong proprieties/guarantees, like the reliable broadcast problem [19].

2.3.1 Reliable Broadcast Problem

As the name suggests, the reliable broadcast problem [19] provides somewhat strong guarantees. It is, hence, a stronger version of the best-effort broadcast problem. In particular, the reliable broadcast requires that if a correct process i delivers message m, then every correct process j has to deliver message m at some point in time.

One solution that can solve this problem is to use flood [28]. As described above (in Section 2.2), when using flood, when a node receives a message for the first time, it sends that message to all of its neighbors, i.e., all the nodes that that node knows about. This way, we are flooding the network with the message, so that all nodes receive it (and, consequently, deliver it to the application layer). Although flooding the network is effective (and it is a very simple and easy to implement solution), it creates a lot of redundant messages in practice, which leads to a lot of traffic and, consequently, high bandwidth usage. For that reason, flood is not an optimal solution for broadcasting

messages in a real system. Furthermore, if a connected node is temporarily disconnected from the system (i.e., has no valid neighbor) it might not receive messages flooded in the network, and an additional mechanism, such as anti-entropy [41], is required, increasing the cost of the protocol. Since we cannot solve the reliable broadcast problem without such a high load of redundant messages, we have to relax the reliable broadcast problem so that we can implement solutions that work in practice.

2.3.2 Probabilistic Broadcast Problem

The probabilistic broadcast problem is a relaxed version of the reliable broadcast problem. In the probabilistic broadcast problem, if a message is delivered by some correct process, then it must be also delivered by every correct process with a configurable high probability [20]. This means that, when implementing a solution to this problem, we can choose how likely it is that all nodes in the network receive the broadcasted message, effectively controlling the trade-off between communication cost and reliability. This is relevant since there is a direct correlation between the likelihood of all nodes to receive the broadcasted messages and the number of redundant messages traversing the network during the process. That means that, if we relax the likelihood of all nodes receiving the message, then we can send less redundant messages and, consequently, generate less traffic.

Epidemic broadcast is a family of protocols that uses gossip to solve the probabilistic broadcast problem. The communication mode used is eager push and $t >= ln(n) \wedge (t < n)$, t being the fanout and n being the total number of nodes in the network [28]. It is called epidemic broadcast because the message is spread in an epidemic fashion, where each node "infects", i.e., sends the message to t (fanout) other nodes [26]. The probability of all nodes receiving the broadcasted message is configurable by changing the fanout used by the algorithm [20]. A higher fanout will result in a higher probability of all nodes receiving the broadcasted message but will also increase the number of redundant messages [29]. A lower fanout does the opposite. When implementing epidemic broadcast algorithms, the challenge is to find the harmony between the likelihood of all nodes receiving the broadcasted message versus generating a lot of redundant traffic, i.e., choosing the right fanout.

According to [9], the probability p of all n nodes to receive the broadcasted message using an epidemic broadcast protocol with a fanout t is:

$$p = 1/(1 + n * e^{-t}) \tag{2.1}$$

2.3.2.1 The Beauty and the Beast of Epidemic Broadcast

Redundancy can be seen as the Beauty and the Beast of epidemic broadcast. On one hand, it can be good because since messages can be lost in the network (while in transit between two nodes), redundancy will make sure that—with a high probability—everyone will

receive the message, because it will organically compensate for the lost messages in the network.

On the other hand, redundancy has a dark side, because on average each process will receive the broadcasted message t (fanout) number of times (from different processes). This adds a lot of redundant traffic to the network and can have a big impact on a network composed by low bandwidth devices. Furthermore, processing such redundant messages also has a cost in terms of CPU usage. The total cost of messages of an epidemic broadcast protocol with a fanout t on a network with n nodes is, approximately, $t \times n$.

2.3.2.2 Performance Metrics

There are a series of metrics that can be used in order to evaluate and compare epidemic broadcast protocols. These metrics can also be generalized to evaluate the performance of other classes of gossip-based protocols. In this section we discuss them; since we will explore, further ahead, the impact of our proposed solution on the performance indicators of these protocols (that will serve as a case study).

Reliability Reliability in this context can be defined as the percentage of nodes of the system that received (and then delivered to the application layer) a broadcasted message [29]. A broadcast reliability of 100%—where all the nodes receive and deliver every broadcasted message—indicates that atomic broadcast was achieved [20]. Even though we usually aim at probabilities very close to 100%, unless flooding is used (i.e., the fanout is equal to the number of neighbors), with epidemic broadcast we can never be sure that we are going to achieve such a reliability—due to the probabilistic nature of epidemic protocols. A higher fanout usually leads to a higher reliability and a lower fanout usually leads to a lower reliability [20].

Relative Message Redundancy (RMR) The relative message redundancy is a metric that captures the message overhead in an epidemic broadcast (or other gossip-based broadcast mechanisms) [25]. It is equal to:

$$(m/(n-1)) - 1 \tag{2.2}$$

m being the total number of messages exchanged during the broadcast and n being the number of nodes that received and delivered the broadcasted message [25]. This metric is only applicable when $m >= 2$. Usually, the optimal RMR value is zero [25]. A value of zero indicates that there was exactly one payload message exchanged per receiver in the system. That would mean that we spent exactly the bandwidth necessary to propagate the message to all nodes—not wasting any more than necessary. A high value of RMR indicates that a lot of redundant traffic was generated and, therefore, a high amount of bandwidth was unnecessarily consumed. Higher fanouts usually lead to higher RMR and lower fanouts usually lead to lower

RMR [25]. When using pure gossip (such as simple epidemic broadcast) to disseminate a message, RMR tends to $t - 1$, t being the fanout [29].

Control messages (like the ones used on the pull gossip strategy) are not considered in this metric, i.e., they do not count towards the value of m. This is because control messages are usually significantly smaller than the actual payload messages—otherwise we would not be using a communication mode that relies on control messages. Therefore, in general, the control messages do not have a substantial impact on the total generated traffic and, consequently, can be ignored.

It is worth noting that it is easy to have a high reliability with a high RMR (by employing a very high fanout) and it is also easy to have a low reliability with a low RMR (by having a very low fanout) [25]. Nevertheless, it is challenging to have a high reliability with a low RMR—which is the ultimate goal. The challenge lies on finding the fanout that provides the best balance (according to our priorities) between a high-enough reliability and a low-enough RMR.

Last Delivery Hop (LDH) The last delivery hop metric [29] measures the maximum number of hops performed by a message that is delivered to the application (i.e., it is not redundant for the receiver), i.e., the maximum number of nodes that a message went through before being delivered to the application layer. This value is usually highly dependent of the network diameter, i.e., how wide the network can be in terms of paths between nodes [29]. For example, in a network where the shortest amount of links between two nodes is d, assuming an atomic broadcast started by one of those nodes, the LDH of that broadcast will never be smaller than d.

Latency The latency of an epidemic broadcast is given by the difference between the time upon which the last process delivers the message and the time upon which the first process (the original broadcaster) broadcasts the message [28]. Usually, we strive for low latency. However, the obtained latency is not solely dependent on the epidemic broadcast protocol—it is also dependent on the network itself (e.g., the latency between nodes, the node relative locations). If every path between the nodes has the same latency (which is very unlikely in practice), we can compute the broadcast latency by multiplying LDH by the link latency between two nodes (also known as *per node latency*) [29].

2.4 Overlay Networks

An overlay network (or simply overlay) is a logical network on top of another network (e.g., the physical network). In an overlay, nodes define (logical) neighboring relationships between them. These neighboring relationships can be materialized as links. This set of links and nodes constitutes an overlay, which can be seen as a graph, composed of nodes and links established between them. This overlay is then used by applications or

services to propagate messages between the nodes. The set of neighbors of each node (i.e., the nodes to whom a node shares an overlay link with) is called the partial view of the node [28]. In other words, the partial view of a node n can be seen as the set of nodes that are known by n.

An overlay (or graph) can either be undirected or directed, i.e., having asymmetric links or exclusively symmetric links, respectively. In a symmetric overlay, if node a has node b in its partial view (i.e., as neighbor), then node b has node a in its partial view. When using symmetric views, a node is always sure of the nodes that have him as neighbor and that can be useful in some cases. For example, if the overlay is symmetric and a node's partial view is not empty, then that node is sure that is not disconnected from the network, i.e., that other nodes have it in their partial views. However, symmetric partial views are usually harder and more expensive to maintain.

Contrarily to using a partial view, we could use a global view membership system. This would mean that each node would know every single node in the network [12]. Using this strategy, we would not need an overlay, since all nodes could interact with every other node (as long as they were connected to the physical network). That raises the question: why bother with creating and managing an overlay when we can have all nodes communicating with all other nodes? The answer is simple—when we are dealing with a large system (i.e., a system with a very high number of nodes), the set of nodes in the system is not static. Large systems are often dynamic because processes might be added at all times (to deal with additional load, for example) and other processes might leave the system anytime (e.g., if they crash). If every node (i.e., process) contains a global membership, that would mean that that node would have to keep up with every addition and removal to the system—which might not be realistic, because nodes can be far away from each other (thereby presenting high latencies) and that would lead to unacceptably high bookkeeping costs [31]. This could easily result in network saturation and total system inoperability.

There are a number of proprieties that overlays must own in order to support a high level of fault tolerance and fast message dissemination, i.e., for epidemic broadcast to be effectively implemented on top of them. These proprieties are useful to ensure the correctness of the overlay (when it comes to connectivity and accuracy) but also to measure the quality of the partial views—in terms of fault tolerance, message dissemination efficiency, etc. The most relevant of these proprieties are:

Connectivity The connectivity of an overlay indicates whether all nodes in the network are connected, i.e., if there is a path between every correct node a to every correct node b. For an overlay to be correct, it has to be connected. In other words, no node can be isolated from the other nodes. If a network is not connected, some nodes will not be able to communicate with the remaining elements of the system [29].

Accuracy The accuracy of a node is given by the number of correct nodes in its partial view divided by the total number of nodes in its partial view [26]. By correct, we

mean nodes that did not fail, i.e., that are running. Optimally, we would want that partial views would have an accuracy of 1 (100%) at all times, but that is obviously impossible because nodes can crash and nodes whose partial views had such a failed node as neighbor will naturally take some time to suspect such failure events. Therefore, we aim to have an accuracy as close to 1 as possible. The accuracy of an overlay (or graph) is defined by the average over the individual accuracies of all the correct nodes [26]. For an overlay to be correct, its accuracy has to eventually be equal to 1 (if no nodes are removed nor added for some amount of time).

Having a high overlay accuracy is very important for epidemic dissemination. If the accuracy is low, it means that nodes have a lot of faulty nodes in their partial views and when disseminating a message to t random neighbors, some of those t nodes will not contribute to the effective dissemination of the message, decreasing the broadcast reliability (unless we use a higher t in order to mask this issue) [29].

Degree Distribution The *out-degree* of a node is given by the number of nodes that are present in its partial view [29], i.e., its number of neighbors. The *in-degree* of a node is given by the number of nodes that have that node as neighbor [29].

The *out-degree* of a node is useful to infer the node's contribution to the membership protocol, i.e., how important that node is in maintaining the overlay connected [29]. For example, usually, a node with an *out-degree* of three is less relevant in maintaining the overlay than a node with an *out-degree* of ten, because the latter has the responsibility of managing and disseminating the messages to more nodes. If a node with a high *out-degree* crashes, it has an increased probability of affecting overall overlay proprieties such as the connectivity.

The *in-degree* of a node evidences how well known a node is in the network. It also has a direct correlation with the number of redundant messages that that node might receive—a higher *in-degree* results in a higher probability of that node receiving a high number of redundant messages [29], when messages are being broadcasted through the overlay network.

In the case of a homogeneous network (i.e., a network where all nodes have the same resources and links possess the same proprieties), all processes should have a similar number of *out-degree* (and *in-degree*), so that the traffic load is well distributed between them [26]. It is also worth mentioning that in a symmetric graph/overlay, a node's *in-degree* is equal to its *out-degree*.

Average Path Length A path from node a to node b is composed by the edges that a message goes through to travel from a to b. The length of the path from a to b is given by the number of such edges. The average path length of an overlay/graph is equal to the average of the shortest paths between all pairs of different nodes

in the overlay[1]. The average path length should be low, so that messages do not have to pass through a lot of nodes in order to reach their destinations, which adds latency [26]. This metric is closely related to the overlay diameter [29]—a wider network results in messages having to go through numerous edges to go from one side to the other of the overlay.

Clustering Coefficient The clustering coefficient of a node is equal to the number of edges between the neighbors of that node divided by the total number of possible edges between all of them [26]. It results in a number between zero and one. A clustering coefficient close to one means that a node shares a lot of the same neighbors with its neighbors. The clustering coefficient of the graph/overlay is equal to the average of the clustering coefficients of all nodes. An overlay with a high clustering coefficient is bad for two reasons: i) it will produce more redundant messages for nodes, since they have essentially the same neighbors as their neighbors; ii) it lowers the fault tolerance of the overlay because if most of the nodes share the same neighbors as their neighbors, then those nodes do not have a lot of neighbors outside their clusters, which means that the inter-cluster links are scarce—if they are broken, the overlay can more likely become disconnected. Moreover, a high clustering coefficient usually results on a high average path length [29], which is usually an undesirable propriety.

As mentioned in Section 2.1.2, the membership layer is responsible for managing the partial view. This partial view has to be frequently updated so that the nodes that left the network are removed from it (to improve accuracy) and the nodes that entered the network become known by (some) other nodes.

Overlay networks can be divided in two groups: structured and unstructured. They both have different characteristics and have been employed for different use-cases. In the following sections we are going to delve into them.

2.4.1 Structured Overlays

Even though this dissertation focuses on unstructured overlays, it is worth discussing their structured counterparts. A structured overlay is an overlay whose topology has proprieties known *a priori* [28]. Each node in the overlay is usually identified by a unique identifier. The neighbors of a node are defined (at least partially) considering the identifier of nodes, such that the overlay topology has a global organization that can be exploited. For instance, Chord [48] structures nodes in a ring where they are organized in increasing order of their identifiers (assuming that the identifier space is circular). This ring topology is then exploited to easily navigate the overlay to locate particular nodes considering their identifiers.

[1] This propriety is only worth measuring if the network is connected. Otherwise, at least one node will have an infinite shortest path length to all other nodes—which will result in an infinite average path length, by definition.

Knowing some topology proprieties allows us to implement mechanisms that operate on top of the overlay in a much more efficient way. Thereby, we can use structured overlays to implement specific abstractions and functionalities more efficiently than with unstructured ones [48].

One example of a problem that can be very efficiently solved with a structured overlay is the exact location problem [28]. This problem can be defined informally as finding an object (i.e., a resource, such as a file) by its unique ID. If we used an unstructured network, we would be blindly searching for the object across random nodes. However, with a structured network, we can take advantage of a scheme to attribute the responsibility of hosting that resource to a particular node, and then find it faster and more efficiently, i.e., with fewer messages. This can be done by hosting each object in the nodes (or node) whose identifiers are closest to the object ID.

On the other hand, structured overlay networks have an important drawback, which is the slower convergence when compared to unstructured overlays. Due to having a strict structure, structured overlays are not as flexible when membership changes happen, since nodes that join or leave the network trigger a mandatory restructuring of the overlay for it to enforce its topological proprieties [28]. Besides this, structured overlays are not as fault-tolerant as unstructured overlays—a node can quickly become isolated if a relevant part of its neighbors crash [28]. However, most implementations try to minimize these problems (specially the lack of fault tolerance) with some algorithm tweaks, as we are going to discern below.

There are many protocols to build and manage a structured overlay. Chord [48] and Kademlia [35] are two of the most well known, which we will discern in more detail, for completeness of the survey of relevant works.

2.4.1.1 Chord

The Chord [48] protocol main functionality is—given an object identifier, find the node responsible for managing that object. To achieve this, Chord structures nodes in a ring–like topology, each node being responsible for an interval of object identifiers. Each of the nodes has a unique ID, and they are organized based on that ID. Each node is responsible for the data objects whose identifiers fall in the interval between that node's predecessor on the ring and its own identifier.

A simple and naive routing solution would be for each node to have the next node in the ring in its view (i.e., node n would know the IP address and port of $successor(n)$). That way, each node would only know one other node (its successor). Hence, whenever a node would receive an operation to find an object with a specific identifier, it would check if that node was the one that managed it, and if it was not, it would send the operation to its successor. When the message finally arrived to the node that manages the object, that node would reply to the client.

Chord proposes a more complex routing solution in order to deal with the fault tolerance and efficiency problems generated by the simple solution above. Essentially, it works by each node n maintaining in its view a set of nodes whose IDs are $neighID = n + 2^{i-1} \wedge i > 0 \wedge neighID < maxID$, $neighID$ being the ID of neighbor and $maxID$ being the maximum possible identifier. In this solution, in case the node receiving the search query has in its view a node with ID equal to the identifier being searched, it redirects the query to that node. Otherwise, the node receiving the search query redirects that query to the successor of the known closest preceding node of the node that manages the object. Effectively, this halves the distance to the target node at each routing step.

With this solution, each node stores information about m nodes when the total number of nodes in the network is 2^m. This number is not very high, as it increases logarithmically with the total number of nodes in the overlay. However, it is high enough to make the overlay robust against nodes failing and broken links. The fact that each node has neighbors well spread through the overlay also helps with quickly locating the searched objects (since a low number of hops is often sufficient to find the node that manages the target object).

It is also worth mentioning that Chord proposes a periodical stabilization process with the goal of converging the overlay to deal with cases where nodes recently joined or left the network. In this process, nodes update their successors and predecessors in order to match the correct structure of the overlay. This cyclic stabilization process presents a trade-off between spending a lot of bandwidth (by doing very frequent stabilization processes) to have a correct overlay at (almost) all times or spending little bandwidth and risk having an often inconsistent overlay.

2.4.1.2 Kademlia

Kademlia [35] is a protocol that builds and maintains a structured overlay that is widely used. Although the number of expected hops for each search query is $O(log(n))$ (similar to Chord), Kademlia presents some clear advantages when compared to Chord. Some of these advantages are the fact that configuration information spreads automatically as a side effect of search queries. Besides that, Kademlia uses parallel asynchronous search queries in order to be fault-tolerant to nodes that have possibly crashed mid-execution.

This protocol has a lot of optimizations compared to simpler protocols (like Chord), but perhaps the most important one is that fact that it uses an XOR metric to measure distances between nodes. Because of that, Kademlia is able to use symmetric links between the nodes, which leads to a greater flexibility on the search queries redirection because—unlike Chord—Kademlia can redirect the search query to any node that is close to the node that should have the object. In fact, it can even redirect the search query to multiple appropriate nodes in parallel. This concurrency can be controlled by a concurrency parameter α that trades-off low bandwidth usage for low-latency and increased fault-tolerance.

2.4.2 Unstructured Overlays

An unstructured overlay is an overlay where the neighbors that each node has are selected at random [28]. Therefore, unlike structured overlays, an unstructured overlay has a random topology, i.e., we cannot infer proprieties about the topology of the overlay before it is actually built.

This type of overlay has a lower maintenance cost than structured overlays because there is no structure to be enforced [28]. Besides that, they tend to have better fault tolerance than structured overlays (as explained thoroughly in Section 2.4.1), mainly due to the fact that no strong topological proprieties have to be maintained when a node updates its partial view, i.e., any node can have any random node in its view, unlike structured overlays, where each node has to have a very specific set of nodes in its partial view.

Due to its dynamic and low-cost adaptability and high tolerance to failures, unstructured overlays are useful for message dissemination (e.g., epidemic broadcast) [9]. They are also useful for replication of data across numerous nodes [32]. Another use case is system monitoring, where nodes monitor each other (e.g., their load, if they have not crashed) [52]. A different problem where using unstructured overlays is useful is the (generic) resource location problem, where a node has to find some resource in the network (e.g., files) matching a given set of proprieties (e.g., $size >= 256$) [32]. These are all problems where unstructured overlays are highly advisable (instead of structured ones) because solving these problems would not benefit from an existing structure, i.e., a structured overlay does not make solutions to these problems much more effective or efficient.

When it comes to building unstructured overlays, there are two strategies: cyclic and reactive. A cyclic strategy uses periodic timers to shuffle the partial views of nodes between them [28]. From the point of view of a node, it sets a periodic timer t and then every t units of time it will perform an operation where its neighbors may change (and it may also lead to updating the contents of some of its neighbors' partial view). This strategy is called cyclic because the only trigger for maintaining the overlay is given by periodic (i.e., cyclic) actions, i.e., it is not reactive upon some other event (like nodes joining the network).

2.4.2.1 Cyclon

Cyclon [52] is a cyclic, gossip-based membership management protocol (also named as peer-sampling protocol) that builds graphs (i.e., overlays) with low diameter, low clustering, highly symmetric node out-degrees, and that are highly resilient to massive node failures. The protocol is also capable of restoring randomness when numerous nodes fail, specially when compared to other (simpler) shuffling protocols. In [52], the authors presented an experimental analysis of a basic shuffling protocol for large networks (to

demonstrate that shuffling is indeed a promising technique to build these protocols), and then presented an enhanced version of that basic shuffling protocol—Cyclon.

Cyclon employs an enhanced version of shuffling. Shuffling consists on (generally) two nodes switching neighbors between themselves, so that at the end of the shuffle operation, they both end up with a different partial view than they had when they started it. The number of neighbors shuffled by each node is called the shuffle length. Enhanced shuffling follows the same model as basic shuffling, with the difference that nodes do not randomly choose which neighbor to shuffle views with. Instead, upon shuffling, each node selects the neighbor whose identifier was created earliest. This is achieved by every node having, in its view, not only a list of nodes, but, instead, a list of pairs (node, age). This way, each time a node shuffles, it increases the age of each neighbor identifier and chooses the neighbor with the highest age to shuffle with (instead of a random one). Note that the age of an identifier is left unchanged when an identifier is sent to a peer during the shuffle. The other difference between Cyclon and basic shuffling protocols is in the *join* operation. Cyclon implements a sophisticated way of a node joining a network by only knowing one neighbor (contact node) without disrupting the randomness of the network, which is key to maintain the proprieties of unstructured overlays (such as high fault-tolerance).

In [52], the authors suggested a time interval between shuffles of 10s. However, for some applications, this might be too long, since the network can take minutes to recuperate from node failures (depending on the number of nodes that fail concurrently). That said, there is clear correlation between the bandwidth used and the speed of the protocol to detect and act upon failures. This trade-off between low-bandwidth usage and fast recovery might be one of the biggest draw-backs of this algorithm. This happens because the algorithm is exclusively cyclic, having no reactive strategy to failed nodes.

2.4.2.2 SCAMP

Unlike Cyclon, SCAMP [12] is a reactive gossip-based membership management protocol. A reactive strategy changes the membership of nodes in the overlay only when specific events are triggered—like nodes joining or leaving the network, instead of using periodic behaviors. SCAMP is also self-organizing, meaning that the size of the partial views is dynamically and locally adapted according to the number of nodes in the system, as to ensure reliability in large scale settings.

SCAMP works by having each new neighbor sending a subscription (i.e., presenting itself) to an arbitrary node that it already knows (contact node). This means that a node that enters the system starts with one node in its (partial) view. The node that receives the subscription, forwards it to all nodes on its partial view and then sends the subscription again to more c nodes of its partial view (c being a design parameter that determines the proportion of failures tolerated). When a node receives a forwarded subscription, it adds the subscriber to its partial view (if it is not already there) with a probability computed

based on the number nodes already on its view. Else, it forwards the subscription to one of its existing neighbors. This probability makes the system configure itself towards (partial) views of average size $(c+1)*log(n)$, n being the total number of nodes in the system.

For failure detection, the protocol suggests two different techniques (that can be used simultaneously). The first one is for each node to save a different view (*in-view*) that contains the set of nodes that have that node in their partial views. That way, each node knows which nodes have it as neighbor. Then, when a node wants to leave the system, it just has to send a *leave* message to all nodes in its *in-view*. When a node receives a *leave* message, it removes the sender of the message from its view. The second technique is aimed to solve the problem of nodes that crash without warning (i.e., without sending the *leave* messages). In this technique, nodes periodically send a heart-beat message to their neighbors in order to make sure that they are still alive. If a node does not answer the heart-beat for x times in a row, it is removed from the partial view of the node that sent the heart-beat. Both of these techniques are effective ways of maintaining a high overlay accuracy.

Besides being proven mathematically, the algorithm was also tested against a full membership protocol system in [12]. Using SCAMP, the view size of each node converged to an average size of $(c+1)*log(n)$. Because of this, when using the gossip algorithm with SCAMP, the fanout should be equal to the partial view size of each node, since a fanout $>= log(n)$ is necessary to achieve reliability. In terms of resilience to failures, SCAMP proved to be very reliable, with results very close to global membership knowledge—even when almost half of the nodes failed.

On the other hand, Scamp presents a significant limitation—if a node was the last one to enter the system (and no nodes enter the system after it), it will ever only know one other node, i.e., its (partial) view size will be 1 forever. Even if a node was not the last one to enter, it can stay with a low view size for a long time if not many nodes enter the system right after it did, specially if the nodes that enter the system after that node do not choose a contact node nearby it. This means that this algorithm works well only if a lot of nodes are always entering/exiting the system. This happens due to the reactive nature of the algorithm—if there are no external changes to the overlay (i.e., no nodes joining or leaving) then the overlay will not change. Cyclon does not have this problem because no matter how many nodes enter or leave the network, it will always shuffle neighbors periodically.

2.4.2.3 HyParView

HyParView [26] is a membership protocol that supports gossip-based broadcast, which ensures high levels of reliability even when the rates of node failure are very high. This protocol uses two distinct partial views to ensure that high reliability is achieved, even when using a lower-than-average fanout size on the gossip protocol being used on top of

it.

Many membership protocols with healing properties see the reliability of message broadcasts be seriously affected after heavy failures [26]. Even for protocols that are able to totally recover, it takes a long time to restore their desirable view properties when a very high number of nodes fail [26]. Furthermore, when using usual membership protocols, we often have to use high fanouts if we want to achieve high reliability, due to the lack of fault tolerance of these protocols [26]. HyParView claims to solve these problems.

The protocol works as follows. Each node maintains two views: i) small symmetric active view of size $t+1$, t being the fanout used in the upper (service) layer for the epidemic broadcast; ii) a bigger passive view of backup nodes that can be promoted to the active view when one of the nodes in the active view fails. Failures are detected using TCP as an (unreliable) fault detector. The active view is maintained reactively and the passive view is maintained cyclically (by performing shuffles with other nodes, like Cyclon). When a node sends to its neighbor a set of nodes to be shuffled, it includes in the set some nodes from its active view as well. This is done to increase the probability of nodes having passive views with active nodes. By using these two views and TCP as a reliable transport, it is possible to use smaller fanouts, while maintaining a very high reliability, resulting in more cost-effective gossip protocols. Moreover, by using TCP as a reliable fault detector, this membership protocol has fast healing properties, quickly recovering from a high number of failures.

On a negative note, a lot of messages are used to maintain symmetric views (at least two times the number of needed messages to maintain asymmetric views). This might cause some unnecessary overhead. On the other hand, since the protocol provides us with the capability of using smaller epidemic broadcast fanouts (while providing the same levels of reliability), perhaps the number of messages spent to guarantee symmetric views ends up compensating the number of messages saved on the service layer (due to the lower fanout)—which is likely—specially if the overlay is fairly stable (i.e., if it has low churn[2]).

2.4.3 How to Adjust/Bias the Network Topology

Unstructured overlay networks—while being highly reliable and adaptable—can have performance caveats due to the randomness of the links that are established between the nodes [28]. This means that some connections between nodes might end up being inefficient (in terms of latency or other metrics) and that is not good, because it leads to an undesirable impact on the network efficiency. This is the problem that T-Man [16] and X-Bot [27] try to solve—how to optimize these networks, while maintaining some key proprieties.

[2]Churn is defined by the frequency to which nodes join and leave the network/overlay. An overlay with high churn is an overlay with a lot of nodes frequently joining and leaving, while an overlay with low churn is a stable overlay where nodes join and leave infrequently.

Both of these protocols bias the topology of unstructured overlays according to some chosen criterion. The protocols are agnostic to the criterion chosen, i.e., any criterion can be used as long as it allows nodes to rank other nodes (in the case of T-Man) or links (in the case of X-Bot) based on it.

2.4.3.1 T-Man

T-Man [16] is an algorithm that can be applied on top of a random graph in order to bias its topology according to some criterion. Therefore, T-Man is capable of emerging structure out of a random graph, i.e., out of a pure unstructured overlay.

The protocol operates between the membership layer and the service layer, i.e., it uses the membership layer, and it presents itself as the membership layer to the service layer above. It runs in a cyclic way. Every time interval t, each process picks the highest ranked node from its view (according to the ranking function/criterion provided) and sends to that node a set of nodes that includes its current view, itself and a sample from the random view on the membership layer. When a process receives a set of nodes, first, it replies with a set containing its current view, itself and a sample from the random view on the membership layer. Then it merges the received set with its own view. So that each process has always c nodes in its view, the merge operation merges both sets based on the ranking function, i.e., it ranks the nodes from both sets and chooses the highest ranked c nodes.

T-Man also proposes some optimizations to the solution described above. One of them is to forbid each node to communicate (in the context of T-Man) more than twice per interval t. That way, if a node tries to communicate with other that currently cannot communicate, it skips that node and tries with the next one. The authors in [16] affirm that this technique strikingly improves the convergence speed of T-Man.

Another proposed optimization is to reduce the payload of messages sent between nodes. This can be achieved by reducing the number of nodes sent in messages through inferring what nodes would definitely be discarded by the receiver of the set.

The periodic optimization of each node's views by applying the T-Man protocol makes the network topology more efficient (according to the criterion/ranking function used) over time. In [16], the authors construct a torus out of a random graph with 2500 nodes in just 15 cycles (with a view size of 20)—which is remarkable. On the other hand, although T-Man ensures that the topology of the overlay improves in a fast manner, it does so at the risk of allowing the topology to break (and the overlay to lose some of its desired proprieties) [7].

2.4.3.2 X-Bot

The goal X-Bot [27] is to optimize the overlay by minimizing the cost of the links between the nodes, while striving to maintain the same number of links as the original overlay network. The link cost function is parameterized and encapsulated in a companion oracle.

X-Bot allows us to use any efficiency criteria X, i.e., we can choose the oracle that we want for the costs of the links between neighbors. This means that we can choose latency as our cost indicator or any other characteristic of the path (like distance, monetary cost of infrastructure, etc.)—X-Bot is completely agnostic to the oracle. Besides this, X-Bot preserves several key proprieties of the overlay, like a low clustering coefficient and low overlay diameter.

The protocol is decentralized and relies on a 4-node coordinated optimization technique in order for those 4 nodes to switch links between them so that the least link cost is achieved. By always switching links between 4 nodes, the algorithm makes sure that most node degrees remain the same. This switch is made periodically, making sure that the overlay never stabilizes at some local minimum. In this technique, each node starts optimization rounds in which it tries to switch one node of its active view with one (better) node in its passive view, and that node communicates with another node that communicates with another node. To avoid breaking some key proprieties of the overlay network (like the low clustering coefficient, low average path length, or connectivity), the algorithm keeps some unbiased neighbors that it never tries to switch/bias. These "unbiased neighbors" should be the y neighbors with the "highest-cost" in each node's initial partial view, y being a parameter.

This protocol was tested and compared with T-Man, Araneola [36] and GoCast [50], which are other protocols that try to bias the network in any way to achieve better efficiency. Two testing scenarios were used: Cartesian scenario and Planet-Lab scenario. The oracle used measured latency between links (as cost). X-Bot managed to have the lowest latency in a Cartesian scenario while maintaining a reliability of 100%. In the Planet-Lab scenario, X-Bot had the 2nd lowest latency, but T-Man—that achieved the lowest latency—presented a reliability of only 16%, while X-Bot maintained a reliability of 100%.

2.5 Addressing Heterogeneity in Unstructured Overlays

All the protocols presented above—either for building and maintaining unstructured overlays or for biasing their topology—assume that the network is homogeneous, i.e., they assume that every node (and many times each network link) have similar resources. However, in real world scenarios that is not true—specially in the edge [7]—where we can have very powerful machines (e.g., datacenter servers) and very weak machines (i.e., machines with low resources), such as IoT devices, smartphones, etc.

For this reason, in most real-world scenarios, it is not wise to evenly distribute load on the nodes and network links, since they are heterogeneous in nature, which would make some nodes be extremely saturated with work while others would be operating below their capacities. This leads to low performance on services, resulting in high latency (due to some saturated nodes, for example) and poor bandwidth allocation [6]. Therefore, there are some techniques (i.e., protocols/algorithms) that aim to tackle this problem.

There are a lot of ways to tackle resource heterogeneity. Some solutions aim at solving the heterogeneity problem at the service layer by optimizing epidemic broadcast (or other gossip) algorithms for heterogeneous networks. Other solutions tackle the problem at the membership layer by trying to optimize the overlay for heterogeneity so that when some service (like epidemic broadcast) uses it, it benefits from the already optimized overlay.

The main goal of the protocols and algorithms that aim at addressing the resource heterogeneity in unstructured overlays is usually to minimize latency. This is because it is the metric that often has the bigger impact in a real world scenario [7]. Besides that, by optimizing latency we are also indirectly optimizing other metrics—like bandwidth, distance between nodes, etc. For example, if an algorithm strives to achieve low latency, it will probably do so by fairly distributing bandwidth among resources. There are also solutions that—instead of trying to optimize latency—have parameterized optimization functions that allow us to directly optimize whatever metric/aspect we want (including machine operation cost, for example).

2.5.1 Tackling Heterogeneity at the Service Layer

As mentioned above, there are several solutions that try to optimize services for heterogeneous resources. In this case, we are interested in protocols that adapt epidemic broadcast for heterogeneous settings. Some of the most relevant solutions are presented below.

2.5.1.1 Emergent Structure in Unstructured Epidemic Multicast

The work presented in [6, 40] proposes a probabilistic broadcast protocol (commonly named adaptive gossip) for heterogeneous networks. The protocol works by lazily deferring message transmission according to a configurable policy. It combines two push gossip communication modes (eager and lazy) in an epidemic broadcast protocol.

The proposed gossip algorithm mixes eager and lazy push. In summary, before a node broadcasts a message, it runs a *split* function that—depending on the sender, the neighbors to which the message is being sent, and the message itself—decides to which neighbors the message should be sent via eager push and to which neighbors the message should be sent via lazy push. For example, if node a was going to broadcast message m to nodes $\{b, c, d, e, f\}$, it would first run a *split* function that would return something like $\{eager : \{c, e\},\ pull : \{b, d, f\}\}$; then, node a would send m via eager push to $\{c, e\}$ and via lazy push to $\{b, d, f\}$. It is worth noting that the broadcasting fanout (t) of node a does not change, since the function *split* always returns two sets (x and y), in which $|x| + |y| = t$ and $x \cap y = \emptyset$.

Being the goal of the protocol to achieve the best balance between lazy push and eager push communication modes, the authors in [6] presented 4 different strategies to implement the *split* function: i) deciding lazy/eager push based on a probability; ii) deciding lazy/eager push based on the number of hops that a message has traveled; iii) deciding lazy/eager push based on the radius between the sender and receiver nodes; iv)

deciding lazy/eager push based on whether the sending node is one of the super-nodes (i.e., most powerful nodes) or not. The protocol uses monitors in order to extract the metrics necessary for using the strategies above. The authors in [6] suggest that hybrid strategies (mix-ups between the 4 alternatives above) are possible and recommended (depending on the case).

By testing adaptive gossip using a hybrid strategy, the authors in [6] have shown that by scheduling the transmission of payload in a combined eager/lazy push epidemic broadcast protocol, one can reduce the bandwidth while keeping a low latency. In terms of reliability, the tests confirmed that the proposed protocol does not impact reliability (compared to the normal gossip protocol). They also tested inputting noise in the metrics obtained by the monitors, and even in the worst case (when noise was the highest), the worst thing that happened was that the lazy push mode was used all the time or that the eager push mode was used all the time. Also, note that even though the best results are achieved when all the nodes run the same strategy, correctness is still assured if they do not.

2.5.1.2 Low Latency Probabilistic Broadcast in Wide Area Networks

The work presented in [39] proposes a probabilistic broadcast protocol (i.e., a variant of epidemic broadcast) for heterogeneous networks that reduces the average end-to-end latency by dynamically adapting to the network topology and traffic conditions. It works by ensuring that a virtual backbone-like node structure of faster nodes is created, so that when a message is first sent, the protocol tries to route it first to the faster nodes which will then use all their available bandwidth to quickly and reliably disseminate the message to the remaining nodes.

The protocol is based on epidemic broadcast, but with two differences in the first w rounds of a message being forwarded among nodes, and then it adopts an equivalent behavior to it from round $w+1$ until the message is delivered to every node. The two main differences (in the first w rounds) are: i) each node broadcasts to a fanout tm, which is greater or equal to t ($log(n)$, with n being the total number of nodes)—that depends on the node's capacity; ii) each node broadcasts to a non-random set of nodes, which is generated based on the latency as perceived by the sender, their advertised bandwidth, and the sender's bandwidth. This technique results in the more powerful nodes receiving (and therefore sending) more messages, while the less powerful nodes receive (and send) fewer messages. Note that, although the function to extract the non-random set of nodes to broadcast probabilistically chooses more powerful nodes, it also chooses some weak nodes—to make sure that every node has a high probability of receiving the message.

The reported experiments in [39] proved that this approach effectively reduces the latency of the dissemination protocol, not only in heterogeneous environments, but also when the system is subject to high load. Another positive aspect of the protocol is that, by limiting the node selection bias to the early (w) rounds of the gossip procedure, the

authors effectively prevent this mechanism from affecting the overall reliability of the broadcast. According to the experiments, this was proved true even when the network is homogeneous.

On a negative note, the authors claim that each node would obtain the estimated (bandwidth) resources of its neighbors through the membership protocol being used. If so, then that means that the membership layer has to be modified to accommodate this operation (to send each neighbor's resources), which results in this protocol not being able to be used with existing membership solutions—unless they are adapted. Adapting the current membership protocol solutions to work with this broadcast protocol would add overhead to the membership protocols. That said, since we would need to adapt the membership protocol anyway, it would be more effective to solve the whole problem on the membership layer, i.e., make the membership layer responsible for constructing the optimized overlay considering the heterogeneous capacity of nodes, so that the broadcast layer could just receive the (already partially-biased set of) neighbors and simply broadcast to them.

2.5.2 Tackling Heterogeneity on the Membership Layer

We have presented solutions that tackle the resource heterogeneity problem on the service layer (through adapting epidemic broadcast to heterogeneous resources). However, that is not the only (nor possibly the best) way to address the resource heterogeneity problem.

Another interesting alternative is to address the challenges imposed by heterogeneity in the membership layer. By doing so, we are providing more flexibility to the services being implemented because the services just have to use the membership layer as before, with no change to the service internal logic. In practice, this means that we could use the same services as before on top of the already optimized overlay. Besides that, optimizing the overlay for heterogeneity (in the membership layer) is often cheaper than addressing the problem in the service layer because the service layer would have to gather data from the other nodes either through the membership protocol or through extra messages exchanged between nodes (in order to have a view of the network, similar to what the work reported in [39] does). Directly changing the overlay also lets us have more precision on the process, by biasing its topology in exactly the way we want—leading to a greater effectiveness in achieving the desired results.

We thereby present two solutions that bias/change the topology of the overlay in order to deal with resource heterogeneity.

2.5.2.1 Unstructured Overlays Based on Super-peers

In an unstructured overlay we can have specific nodes denominated super-peers [5]. These nodes are more powerful than average and are usually connected to each-other, forming a back-bone link between them.

The super-peer nodes can then be used to propagate more information than the other (non super-peer, i.e., regular) nodes, since they are more powerful. For a broadcast service, we could first disseminate the message between super-peers, and then they would disseminate the message to the regular nodes. This way, we would not saturate the network because of less powerful nodes having to propagate the message to many nodes.

This strategy is particularly useful for search query dissemination in unstructured overlays [5]. Regular processes connect to a super-peer and transmit to it the index of their resources. Search queries are then forwarded to the (closest) super-peer and then disseminated among super-peers to find the owner of the resource being queried.

There are some challenges with using super-peers, however. One huge difficulty is to find out which processes should be super-peers. Since processes/nodes do not have a global view of the network, they do not know if they are more powerful than average or not, i.e., since each node can only see the surrounding nodes, it can be easily fooled if it happens to be in an area with a lot of powerful nodes or a lot of weak nodes. For example, if a node has a lot of weak nodes around it, it may think that it should be a super-peer, when in reality that node is just an average node when considering the whole system. Other (less critical) difficulty that can come up is what to do when a super-peer fails—how can its connected regular nodes find another super-peer in a timely fashion.

2.5.2.2 Biased Layered Tree

Biased Layered Tree [7] is a recent, decentralized membership protocol which takes into account the computational and network capacity available in each node, manually encoded in a numerical value associated with each node and a proximity criteria based on IP prefix commonality. The protocol has the purpose of building a robust hierarchical tree topology that connects and allows managing large numbers of nodes across the cloud and edge.

The protocol assumes that each node contains a "level" value that is manually set. This value indicates whether the node is closer to the cloud or closer to end-devices—and consequently, it encodes whether the node has more or less resources (e.g., a node that has a level 0 is a cloud server and has, therefore, a lot of resources, a node that has a level 4 might be a 5G tower and has moderate resources, and a node that has a level 8 might be a smartphone and has few resources). The algorithm builds a tree–like structure, in which the more powerful nodes are closer to the root and the less powerful nodes are closer to leave positions. This hierarchy is achieved by using the levels. By using this tree–like structure, the nodes with more resources will send/receive more messages and the nodes with less resources will send/receive fewer messages, resulting in a higher efficiency and faster broadcast, minimizing bottlenecks. The authors also try to make sure that nearby devices end up being close to each other in the overlay structure by using the IP address common prefix as a distance criterion, which also helps on minimizing latency. Despite this, the resulting overlay structure is not a pure tree-structure since the protocol uses

some redundant links that make the overlay more robust to failures.

The solution (Bias Layered Tree) was tested in a real-world heterogeneous edge network emulation alongside HyParView [26], X-Bot [27], Cyclon [52], and T-Man [16]. Bias Layered Tree provided the lowest latency of all evaluated solutions, due to the structure of the network, that positions nodes with higher capacity at higher points in the tree—which allows to mitigate the queuing effect due to network saturation. It also proved to have a reliability of 100% when no nodes failed.

On the other hand, when more than 25% of the nodes failed, Bias Layered Tree proved to be less reliable than all the other solutions. This happens due to the structured nature of the overlay, which makes it less robust. Another negative aspect of the solution is the fact that it uses IP address common prefixes as a distance criterion. This is not very reliable, since modern ISPs networks have access to pools of IP addresses that have small common prefixes [11].

2.6 Algorithms for Resource Estimation

In the previous section we explored ways to tackle node resource heterogeneity in the membership layer. However, none of the explored algorithms directly adapts the number of neighbors of a node according to its capacity relative to the distribution of resources across the network, which is what we want to achieve. Therefore, in order for us to do that, we first need to find a way for a node to estimate the distribution of resources of the network, so that it can then adjust its number of neighbors according to how powerful itself is according to that distribution. In this section we explore algorithms that could, at first sight, aid us in computing an estimation of the resource distribution of the nodes across the network in a decentralized way.

Defining properties of the participants of a system has been subject of great interest, leading to the appearance of network partitioning algorithms [10, 17] (i.e., *network slicing*) and population protocols [1, 49]. Network partitioning algorithms aim to create network partitions in order to organize network resources on a global scale, usually considering a particular property of the nodes. However, this aim, although apparently similar to the aim of this thesis, is fundamentally different, as the challenge we address is to allow for a node to decide for itself how to best contribute to the operation of the system, identifying the relationship between its capacity with that of the rest of the network. On the other hand, even though population protocols also try to characterize the system, they suffer from the absence of a deterministic stop condition.

That said, for a node to be able to infer the capacities of the other nodes in the network (and to be able to estimate where it is on this scale), it is necessary to use some type of data aggregation mechanism in the network. As such, in this section we review several aggregation protocols that exist in the literature.

One of the most relevant aggregation algorithms is Extrema Propagation [2]. This

algorithm works through the use of vectors generated by the nodes and employs gossip-based communication among nodes of the system. Despite being able to be used in large-scale systems (due to the estimation error being dependent on the sizes of the vectors exchanged between nodes, which is a parameterizable aspect of this algorithm), this solution only computes an estimate of the sum aggregation function, making it impossible for a node to infer properties related to its contribution to the network.

On the other hand, Q-Digest [46] allows us to compute more complex aggregation functions, such as mode and median. In this algorithm, each node computes a data structure (called *q-digest*). These data structures are propagated through a tree to a specific collector node. Despite the complex aggregation functions that the algorithm offers, it is dependent on a structured network (a tree), which makes it quite fragile in large networks with a significant degree of *churn*.

Another relevant aggregation algorithm is Randomized Reports [3], which works through making a probabilistic survey of the network, in order to count the total number of nodes. In this algorithm, each node sends—through flood dissemination—a message to the other nodes in the network. When one of these nodes receives the message, it replies to the original sender depending on a parameterizable probability of the algorithm. This way, the original node estimates (probabilistically) how many nodes are in the network without all of these having replied to the polling message. Although this algorithm tries to avoid overloading the network with large amounts of messages by minimizing the number of nodes that reply to the original sender, it has limitations in its functionality as it can only estimate the number of nodes in the network. On top of that, even if we could enhance the algorithm to capture more information (other than just estimating the total number of nodes in the network), the fact that it uses flood dissemination makes the algorithm expensive from a communication point of view.

On the other hand, there are aggregation algorithms that offer us a more complex node count and that are lighter in the communication mechanisms that they use. One such example is Random Tour [34]. Although the original aim of this algorithm is to estimate the number of nodes in the network, it can also be used (as stated by the authors) to count only the nodes that satisfy a certain criterion (e.g., total RAM greater than 1024MB). As such, it can be used by a node to infer (in an elementary way) the other nodes' resources.

Random Tour works as follows. The node that wants to know the number of nodes in the network sends a message with a counter to a randomly chosen neighbor. This neighbor increments the message counter by $1/d$ (where d represents the number of neighbors of that node) and sends the message to another randomly chosen neighbor. This process is repeated until the message is returned to the original sender. When it finally returns, the original sender computes an estimate of the number of nodes in the network by multiplying the counter value by its number of neighbors.

In a simple way, this algorithm manages to count the number of nodes in the network that satisfy a certain criterion. However, it does not enable a node to collect enough information in order to easily identify what its contribution to the system as a whole

should be. In addition, Random Tour has another critical limitation, which is the fact that, for the estimate to be made, the random-walk message must return to the original sender. This being the case, as messages are exchanged between nodes at random, if the network is large enough, the probability of a message returning to its originator is very low, making the algorithm slow to obtain estimates. In Chapter 3, we will take advantage of Random Tour's intuitions (more specifically, its communication pattern) to devise ResEst—a distributed algorithm for estimating the distribution of resources of the nodes across a network with partial views.

2.7 Discussion and Summary

In this chapter, we began by explaining the concepts needed to understand the following work. We started by describing the notion of peer-to-peer systems and what services can be implemented with it, then we delved deeply into broadcast, focusing on epidemic broadcast. After that, we explained the concept of overlays, the different types of overlays available, their advantages and disadvantages, the protocols that build (and maintain) them, and two different algorithms for optimizing unstructured overlays. Having introduced these concepts, we delved in the topic of addressing heterogeneity in unstructured overlays—both on the service and on the membership layer. Finally, we explored some of the existing resource estimation algorithms that can aid us upon devising the first part of our solution.

In conclusion, the available tools to tackle heterogeneity in unstructured overlays do not suffice to achieve our goal—none of them directly adapts the number of neighbors of a node according to its capacity relative to the network. We believe that we can, therefore, contribute to optimizing heterogeneous networks by exploring this new area of the design space.

In the following chapter, we will discuss the resource estimation part of our work. In that chapter, we start by presenting the motivation to why we need a resource estimation algorithm, then we present our proposed solution—ResEst, and finally we evaluate the solution through the use of simulation, assessing its validity and (expected) performance.

3

ResEst

In this chapter, we explore the resource estimation problem and present our own proposed solution to it—ResEst. ResEst is a decentralized algorithm whose objective is to estimate how powerful a node is in relation to the rest of the network. This is done through the computation of a histogram that approximates the distribution of capacities/resources of all existing nodes. The algorithm is parameterizable, allowing us to adjust the intended confidence in the computed estimation, in order to allow for some control between the consumed resources by the algorithm (and its running time) and the quality of the obtained estimate.

The chapter is structured as follows: in Section 3.1 we explain why we need a resource estimation algorithm and describe why it is not trivial to build one (in decentralized networks where nodes only have access to partial views); in Section 3.2, we propose our solution; then, in Section 3.3, we validate and evaluate ResEst through the use of simulation; finally, in Section 3.4, we conclude this chapter, summing up our resource estimation contributions, more uses to it (other than in the context of this thesis), and introduce the next steps.

3.1 Motivation for a Resource Estimation Algorithm

As mentioned in Chapter 1, the goal of this thesis is to improve load balancing on heterogeneous decentralized systems by exploring how to adapt the number of neighbors of a node, according to its (relative) capacity—in order for nodes with more resources to have more neighbors, and nodes with less resources to have fewer neighbors. However, in order to do this, we need to be able to estimate the distribution of resources of the other nodes in the network, in order for a node to know where it stands on that scale.

In systems with a central coordinator, this is a more trivial problem to be solved. In such centralized systems, a node can trivially obtain an estimate of the capacities of the remaining nodes in the network—since there is a central entity that knows all nodes and, consequently, can easily track their resources, sharing it with nodes that require access to that information. Something similar can be achieved when considering decentralized

distributed systems where all the nodes have global views of the system [20, 49] (i.e., every node knows every other node), since each node can ask every other node for their capacities—trivially obtaining the capacities of all the nodes in the network.

However, as discussed on Chapter 2, decentralized systems with global views present very high maintenance costs when considering large scale systems. For this reason, large scale systems usually resort to partial views, with each one only knowing a fraction of the totality of the nodes of the system—with each node usually knowing $ln(n)$ other nodes, in which n is the total number of nodes in the network [24].

In these systems (in which nodes only have a partial view of the network), it is not trivial to estimate (locally) the resource distribution of all the participants in the network, for two reasons: i) one node does not know all other nodes in the network; ii) due to the inherent churn of peer-to-peer networks. Consequently, it is difficult for a node to identify the relation between its locally available resources and the rest of the network, in order to locally adjust its contribution for the system, avoiding that less powerful nodes get overloaded (and, therefore, possibly saturated), and enabling more powerful nodes to contribute more to the system—in order to promote a better operation of the system as a whole [7]. Therefore, we need to devise a solution that enables a node (operating only with a partial view) to obtain an estimation of the resource distribution of the system, in order for that node to know where it stands in that scale and adapt its number of neighbors accordingly, as aforementioned.

3.2 ResEst Solution

The devised solution—*ResEst*—consists of using messages that follow a communication pattern based on random walks, in which each message collects data each step to build a histogram with the capacities of the nodes it passes through. The computation of these histograms is inspired by the way Random Tour [34] counts nodes with a certain property. In our solution, for each node that processes a message, the node checks in which histogram class it identifies itself and increments the counter of that class by one. Note that, in Random Tour, the counter increment is done by $1/d$ (where d is the number of neighbors of the node), unlike our solution, where we increment the counter by 1. This is because we assume that all nodes of the overlay network on which ResEst operates have (mostly) the same number of neighbors, which can be guaranteed by using a membership management protocol such as HyParView [26]. It is also worth noting that the number of histogram classes can be parameterizable, i.e., it can go up to u classes, where u is the number of different resource values[1] that were found. Due to the inherent lightweightness of the algorithm (evidenced in Section 3.3.4), we opted to use the maximum number of histogram classes possible, i.e., a histogram class for every different resource value found.

[1] A *resource value* (also called *node resource* or *capacity*), is a static number intrinsic to each node. That said, this value can be anything, as long as it is static. For example, it can represent the total RAM memory of that machine, or its cost of operation.

In order for a node to actually be able to compute the estimate, the message originated by that node must be able to return to itself. However, to avoid incurring in the limitations of other solutions (like Random Tour), we use a stopping criterion for the collection of data, which causes the message to be returned to the node that originated it—*the maximum margin of error* of the average of the values collected by the message, given a parameterizable confidence level [51].

The computation of the margin of error for an obtained estimated average given a confidence level is performed using the Equation 3.1. This computation depends on three variables: *i*) n—the number of samples taken; *ii*) σ—the standard deviation of the samples obtained; and *iii*) z—the confidence coefficient for the desired confidence level and for the number of samples obtained so far.

$$marginErr = z * (\sigma/\sqrt{n}) \tag{3.1}$$

The intuition of this solution is that, as more samples are collected in the random-walk, a high standard deviation from the average of the capacities (which will result in larger error margins) will be amortized by the number of samples. In systems where the node capacities are uniform, a message will need fewer hops to achieve a lower error margin than the parameterized maximum limit (the stopping criterion), because the standard deviation will be small. On the other hand, in a system that has nodes with more heterogeneous capacities, the message will need to take more samples to offset the weight of the high standard deviation. In short, this stopping criterion does not depend on the size of the system, but on the quality of the sample.

Note that the algorithm benefits from a uniform distribution of the node capacities. For networks where this is not the case, in order to increase the reliability of the algorithm, the messages must be able to collect a more representative (i.e., larger) sample from the network. In addition, in order to increase the probability of good quality samples, the network over which the algorithm operates must be unstructured with randomly created links between nodes, where the nodes have a similar in-degree, thus distributing the probability upon which a node is chosen as the target of the random walk evenly. An example of a membership protocol that guarantees these properties is HyParView [26].

In Algorithm 1, we present the simplified pseudocode of ResEst, simplifying the component of the algorithm that starts the random walk process. Note that this may be a periodic process to recompute estimates. Regardless, in Algorithm 1, we present the fundamental part of the algorithm, which is the processing of messages (Alg. 1, line 1). A message contains the arithmetic *mean* (i.e., the average) of the resource values of the nodes that it passed through. This average is computed by storing the node count and the sum of the obtained capacities in the messages. The message also contains the *histogram* containing the number of nodes that have a specific resource value. Finally, it also contains the address of the node that originated the random-walk.

The Algorithm receives two parameters—$maxMarginOfError$ and $confInt$, which

Algorithm 1: ResEst

```
//Parameters
    maxMarginOfError //Maximum allowed margin of error
    confInt //Confidence level

1. Upon ReceiveMessage(m, originNode) from sender do:
2.     if AcceptableMargin(m) then:
3.         trigger send(m) to originNode
4.     else:
5.         randNeighbor ⟵ PickRandomNeighbor()
6.         trigger send(m) to randNeighbor

7. Procedure AcceptableMargin(m):
8.     m.mean ⟵ ComputeAndUpdateMean(m)
9.     m.histogram ⟵ UpdateHistogram(m)
10.    error ⟵ ComputeMarginOfError(m,confInt)
11.    errorRate ⟵ error/m.mean
12.    return errorRate ≤ maxMarginOfError
```

represent, respectively, the maximum acceptable margin of error of the computed average of the acquired sample (for the stopping criterion) and the confidence level to be used.

When a node processes a message (Alg. 1), it computes the new error margin (already taking into account its own resource value) and checks if this number is lower than $maxMarginOfError$. If that is the case, it halts the random-walk and sends the message back to the original sender. If not, the random-walk is continued.

In detail, the procedure AcceptableMargin(m) (Alg. 1 line 7) starts by updating the sample average present in the message by calling the function ComputeAndUpdateMean(m) (Alg. 1 line 8). Then, the message histogram is also updated (function UpdateHistogram(m) in Alg. 1 line 9. Finally, the margin of error of the average of the samples present in the message is computed using the Formula 3.1 described above and encoded by the function ComputeMarginOfError(m, $confInt$) (Alg. 1 line 10). The rate of this error is then computed (Alg. 1 line 11) so that it can be compared with the parameter $maxMarginOfError$ (Alg. 1 line 12), which encodes the maximum acceptable value that the computed margin of error can have.

The result of that comparison is then used as the stopping criterion for the random walk. If the error rate is less than the maximum acceptable error margin, the message is sent to its originator (Alg. 1 line 3). Otherwise, the message is propagated to another random neighbor of the local node—continuing the random-walk (Alg. 1 lines 5 and 6).

Having explained in detail the algorithm, in the next section we are going to validate our solution using simulations while also assessing its performance.

3.3 ResEst Evaluation by Simulation

3.3.1 Simulator Implementation

Our solution aims to operate over large networks (for example, with a million nodes). As such, to validate the mathematical part of our algorithm, we developed a simulator[2]. This simulator ignores network complexities (i.e., network properties, latency and bandwidth between nodes, communication failures, etc.), focusing on executing the ResEst algorithm described in Chapter 3. In this simulator each node only tries to obtain one estimate of the system's resources, that is, each node only generates a random walk.

From a node's point of view, the simulator captures the movement of the message between nodes and calculates the margin of error at each step (for a confidence level, that is a parameter of the experiment). When the margin of error is smaller than the maximum margin of error parameter, the simulator presents the computed histogram for that message, thus terminating its execution flow. The simulation ends when all message execution flows finish. In each step of simulating the communication of a message, the simulator executes the operation of a peer sampling protocol [18] in which it randomly chooses 7 or 14 nodes (respectively to the number of nodes in the experiment—1000 or 1000000—the size of the partial views is the natural logarithm of the total number of nodes in the system) between all nodes in the experiment, thus obtaining a partial view for the node. Then, the simulator chooses a target for the message at random from that partial view of the node. Note that, in this simulator, nodes have a capacity value that has been assigned according to a configuration file that contains the capacities of all nodes. In our experiments we used two distributions for the amount of resources available to each node, a uniform and a lognormal distribution. The node capacities for each of these distributions were generated once (as detailed further ahead on Section 5.3.2.1 of Chapter 5) and saved in different configuration files.

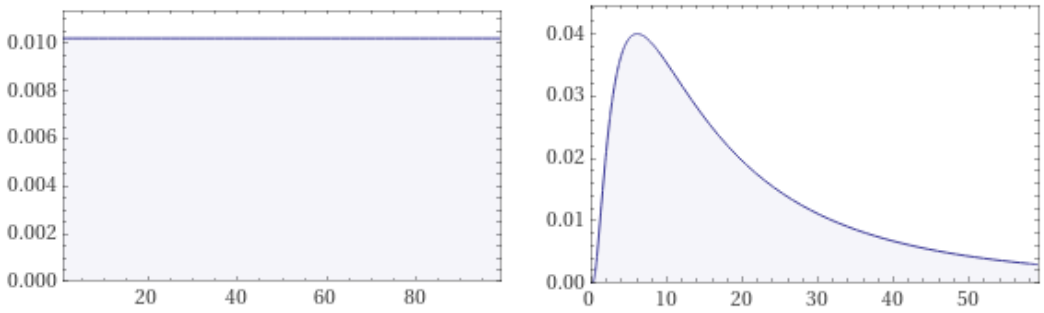
3.3.2 Experimental Settings

In our simulations, we used four parameters to control the experiments: *i*) *Maximum Error Margin*, which represents a percentage of the maximum acceptable error rate of the average capacities for the random walk to stop (i.e., to activate the stopping criterion); *ii*) *Confidence Level*, which consists on a percentage that represents the existing confidence in the estimated value of the average of the collected samples; *iii*) *Distribution*, which represents the distribution of capacities of the different nodes in the network; *iv*) *Number of nodes*, which denotes the number of nodes that are simulated.

We used the following values for the parameters of our simulations. For the margin of error, we used 25% and 15%. For the confidence level, we used 90% and 95%. The simulations were executed with one thousand and one million nodes. Finally, we used two different resource distributions for our experiments—uniform and lognormal. The

[2] https://github.com/hisetip/resest-sim

uniform distribution is controlled by two values—min and max—which limit the range of possible values. The values used to generate the uniform distribution were $min = 1$ and $max = 99$. The lognormal distribution is controlled by two values—μ and σ—which respectively encode the mean and the standard deviation of the variable's natural logarithm. The values used to generate the configuration of lognormal capacities were $\mu = 2.8$ and $\sigma = 1$. In Figure 3.1(a), we present the probability distribution function of the uniform distribution, where the average of the node capacities is 50, with the number of nodes weaker than the average approximately equal to the number of nodes more powerful than this. In Figure 3.1(b) we present the probability distribution function of the lognormal distribution, where a more realistic network is represented. The selection of these specific parameters ($\mu = 2.8$ and $\sigma = 1$) was carried out experimentally, in order to simulate a network where there are many moderately weak nodes and few powerful nodes—similar to a realistic scenario [7].



(a) Uniform distribution.

(b) Lognormal distribution.

Figure 3.1: Distribution of probability functions.

Our experiments were performed using all combinations of parameters described above (resulting in 16 different experiments) and their results are presented in Tables 3.1 and 3.2, for experiments with a uniform and lognormal distribution for the capacities of the network nodes, respectively. The collected metrics in each experiment were: i) the number of hops needed to compute an estimate; and ii) the calculated histogram error. These metrics are explained in detail in the following section (Section 3.3.3). Each different experiment was performed 10 times to remove possible experimental noise. The results show the average of the 10 runs. In the next sections we are going to interpret and analyze those results.

3.3.3 Results

Our results focus on the number of hops needed to obtain an estimate and the histogram error calculated in each experiment. In Figure 3.2 we present the average number of communication hops (rounded to the units), for the various combinations of possible parameters. These plots compare the number of hops between experiments with different

Parameters			Average Results of 10 Experiments	
Max. Margin of Error	Confidence Level	Number of Nodes	Number of Hops	Histogram Error
25%	90%	1.000	14	6.28%
25%	90%	1.000.000	14	7.17%
15%	90%	1.000	37	5.14%
15%	90%	1.000.000	38	4.41%
25%	95%	1.000	21	4.89%
25%	95%	1.000.000	20	5.18%
15%	95%	1.000	51	3.46%
15%	95%	1.000.000	49	3.38%

Table 3.1: ResEst simulation results for uniform distribution.

Parameters			Average Results of 10 Experiments	
Max. Margin of Error	Confidence Level	Number of Nodes	Number of Hops	Histogram Error
25%	90%	1.000	38	3.42%
25%	90%	1.000.000	37	3.38%
15%	90%	1.000	107	1.70%
15%	90%	1.000.000	107	1.93%
25%	95%	1.000	56	2.53%
25%	95%	1.000.000	55	2.31%
15%	95%	1.000	153	1.15%
15%	95%	1.000.000	151	1.13%

Table 3.2: ResEst simulation results for lognormal distribution.

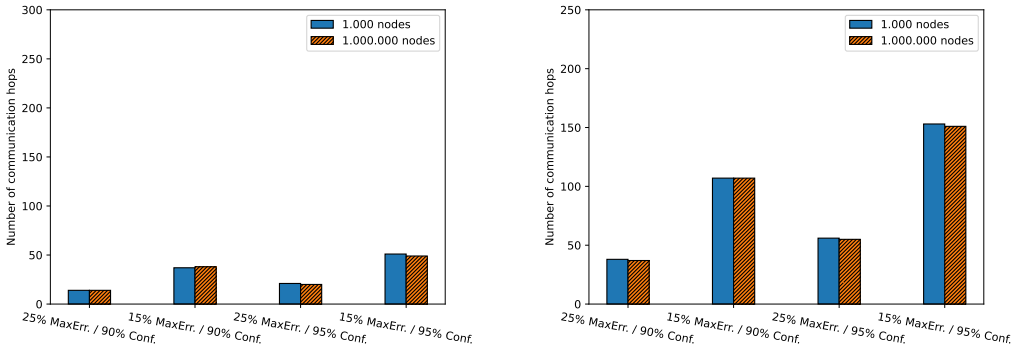
network sizes. From what can be seen, the number of hops the algorithm needs to obtain a histogram does not depend on the size of the system. Another relevant aspect is the observation that when the distribution of node capacities in the network is not uniform, the algorithm needs many more hops to obtain an estimate, as shown in Figure 3.2(b).

In Figure 3.3 we present the error obtained in the estimated histogram (on the y axis) for all experiments (on the x axis). Note that in these graphs we aim to compare the computed histogram error given the parameters used by ResEst (maximum error margin and confidence level). To compute the histogram error, we first reduce both the obtained estimated histogram and the real capacity histogram (given the distribution used) to 5 classes. Then, we compare both histograms, calculating the sum of the differences of each class in both histograms. It is worth noting that the histogram error is higher in experiments where the maximum error margin is 25% compared to experiments with a maximum error margin of 15%. This is because the random walk collects less information, which can be seen in Figure 3.2, where experiments with maximum error margins of 25% have fewer communication hops. In addition, ResEst can obtain estimates with similar errors regardless of the number of nodes in the system.

The Tables 3.1 and 3.2 summarize our experimental results for the two resource distributions used. In both tables, the first three columns refer to the algorithm parameters and their values in the experiments. The last two columns show the (average of) results obtained for each experiment. These are the average number of communication hops in a random-walk and the error of the computed histogram.

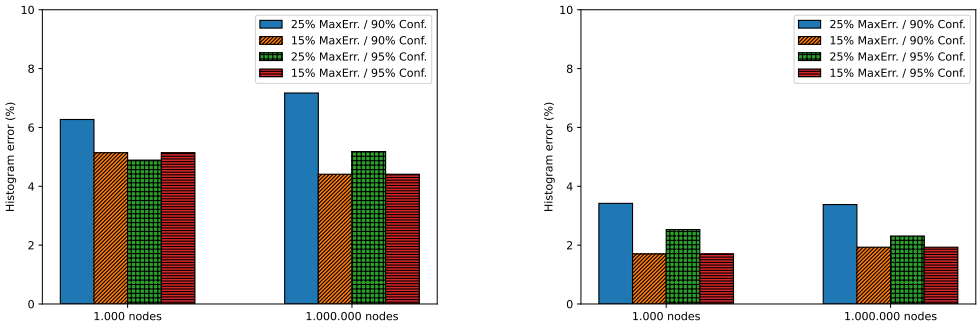
3.3.4 Results Analysis

It should be noted that, regardless of the distribution of the nodes' relative capacities, our algorithm is able to calculate estimated histograms whose error is much smaller than



(a) Uniform distribution. (b) Lognormal distribution.

Figure 3.2: Number of communication hops for uniform and lognormal distributions.



(a) Uniform distribution. (b) Lognormal distribution.

Figure 3.3: Histogram error for uniform and lognormal distributions.

the maximum error margin configured in the algorithm. This validates our solution by the fact that, at confidence levels of 90% and 95%, the admitted histogram error could exceed 10% and 5% of the maximum margin of error, respectively. Still, our results show that in no instance did the histogram errors exceed the parameterized maximum margin of error. These very positive results are due to the fact that the histogram errors are much smaller than the estimated mean errors and the stopping criterion is only dependent on the estimated mean error.

One interesting aspect of ResEst that can be seen in the presented experimental results is that the number of hops needed to obtain the estimate does not depend on the total number of nodes in the network. That is, in a network with 1000 nodes or in a network with 1000000 of nodes, in which both networks follow the same distribution of resources, the number of hops needed to obtain the estimate is essentially the same (for the same parameters of maximum error margin and confidence level).

Thus, as can be seen in our results, the number of hops needed to obtain an estimate

depends on three parameters: *i*) the maximum error margin used by the algorithm—what we noticed is that the smaller the maximum error margin, the more hops will be needed to get the correct estimate; *ii*) the desired confidence level—the higher, the more communication hops needed; and *iii*) the distribution of resources of the nodes in the network—for uniform distributions less hops are needed, for lognormal distributions more hops will be needed because there is less uniformity in the distribution of capacities.

An interesting feature of ResEst is that the processing spent by each node to obtain a very reliable estimate of the network is very low. For example, for a maximum margin of error of 15% with a confidence level of 90% and a uniform distribution of resources, if the algorithm is used by each node every 5 minutes, each node will send and receive between 37 to 38 messages, on average, every 5 minutes. As discussed earlier, this number does not depend on the number of nodes in the network. In other words, in this case, a given node would send and receive 37 to 38 messages whether the network has 1000 nodes or 1000000, making this algorithm a both cheap and scalable solution.

3.4 Summary

In this chapter, we presented a decentralized algorithm (ResEst) to estimate how powerful a node is in relation to the other nodes in the network. This is achieved through the (distributed) construction of a histogram that encodes an estimation of the distribution of capacities of all the existing nodes in the network.

In this chapter we also evaluated ResEst using simulation (with up to one million nodes). The experiments performed while evaluating the algorithm proved its validity, demonstrated its low cost per node, and its reliability. In Chapter 5, we are going to evaluate it in emulation, with one thousand nodes, in order to validate our solution in a realistic scenario.

As mentioned in the beginning of this chapter, the goal of ResEst is to provide a node with the capability of knowing how its own resources compare to the resources of the other nodes in the network, so that it can then adjust its number of neighbors accordingly (which is the challenge that we will tackle in the next chapter). Nonetheless, there are other uses for ResEst. One of them can be to solve one of the greatest limitations of unstructured overlays based on super-peers [5], which is the fact that one node does not know whether it should be a super-peer or not, because it does not possess information about the resources of the other nodes in the network—therefore not knowing how powerful it is. Given the potential of ResEst, this is a path that is worth exploring in future work, in order to unleash the until-now under-allocated potential of super-peer unstructured overlays.

In summary, we have now tackled the first challenge of this thesis—a node can now estimate the resources of the other nodes in the network, being able to infer where it fits in that scale. With this information, that node can now try to adapt its number of

neighbors in order to better contribute to the network. It is exactly how to do this that we are going to explore in the next chapter.

4

Proteus

In the last chapter, we devised an algorithm for a node to acquire an estimation of the distribution of the resources of the nodes in the network. We now need a way to use that information to achieve the main goal of this thesis—each node adapting its number of neighbors according to its capacity in relation to other nodes'.

Therefore, in this chapter, we present Proteus, which is a decentralized algorithm whose objective is to compute the number of neighbors that a given node (the node that is running the algorithm) should have, given its resources and an estimation of the distribution of capacities of the nodes in the network (provided by ResEst, for example). This algorithm then sends the computed number of neighbors to the membership management protocol, in order for that protocol to perform the adaptation.

Proteus computes the desired number of neighbors through the calculation of the node's percentile in the received distribution and using that value to compute a function that returns the appropriate number of neighbors for that specific node. Such function is parameterizable, so that, depending on the application, one can decide what the maximum number of neighbors of the most powerful node can be.

This chapter is structured as following: in Section 4.1, we present an overview of the devised solution, displaying and describing the flow chart of Proteus; in Section 4.2, we delve into the first part of the solution, explaining how we managed to compute the percentile of the node that is running Proteus; then, in Section 4.3, we present the second part of our solution—the optimization function; having the base solution explained, in Section 4.4, we propose an optimization to Proteus, in order to improve stability; then, in Section 4.5, we expose the agnosticism and modularity of the proposed solution, explaining how a membership protocol can adapt its number of neighbors upon receiving a notification from Proteus; finally, in Section 4.6, we conclude this chapter, summing up our contributions and introducing the next steps—experimental evaluation.

4.1 Proteus Solution

Having received the histogram with the estimation of the distribution of capacities of the nodes, Proteus uses that information to compute the appropriate number of neighbors that the node running the algorithm should have. Proteus receives three parameters: i) the default number of neighbors ($defaultNumberOfNeigh$)—the default node degree of the running overlay (usually $ln(n)$); ii) the minimum number of neighbors ($minNumberOfNeigh$)—the minimum number of neighbors that a node can have; iii) the rate of optimization ($rateOfOptimization$)—a scalar that controls what the maximum number of neighbors of the most powerful node can be (this parameter is going to be explained in depth on Section 4.3).

Note that it is necessary to have a lower bound on the number of neighbors of the nodes in order to avoid that the overlay becomes disconnected. Therefore, a number of 4 minimum neighbors is recommended [47].

In Figure 4.1, we present the flow of execution of Proteus. The algorithm starts by receiving a notification from ResEst. This notification contains a histogram representing the estimation of the capacity distribution of the nodes (h), and the capacity of the node that is running the algorithm ($myRes$).

After receiving that data, the node computes its percentile (p) in the estimation of the distribution of capacities. This percentile represents an estimation of how powerful the node is in relation to the other nodes in the network. For example, if the percentile is 90%, it means that that node is more powerful than ±90% of all the nodes in the network.

With the percentile computed, we then execute the optimization function, which receives as parameters the default number of neighbors ($defaultNumberOfNeigh$), the recently-calculated percentile (p) and the rate of optimization ($rateOfOptimization$). This function (explained in Section 4.3) returns an integer (v), which represents the appropriate number of neighbors that the node should have.

If v is higher than the minimum number of neighbors ($minNumberOfNeighbors$), we have to set v to the minimum number of neighbors, as to prevent disconnectivity in the overlay (as mentioned above). Afterwards, we send v—representing the desired number of neighbors—to the operating membership protocol, so that this protocol can remove or add neighbors as required.

To sum up the protocol overview, we can divide Proteus in two parts: the calculation of the percentile of the node and the computation of the appropriate number of neighbors for that node (using the optimization function). In the next two sections we are going to delve into each of these mechanisms.

4.2 Computing the Percentile

As mentioned in the previous section, the first part of Proteus consists in the calculation of the percentile of the node that is running the algorithm, using a histogram which

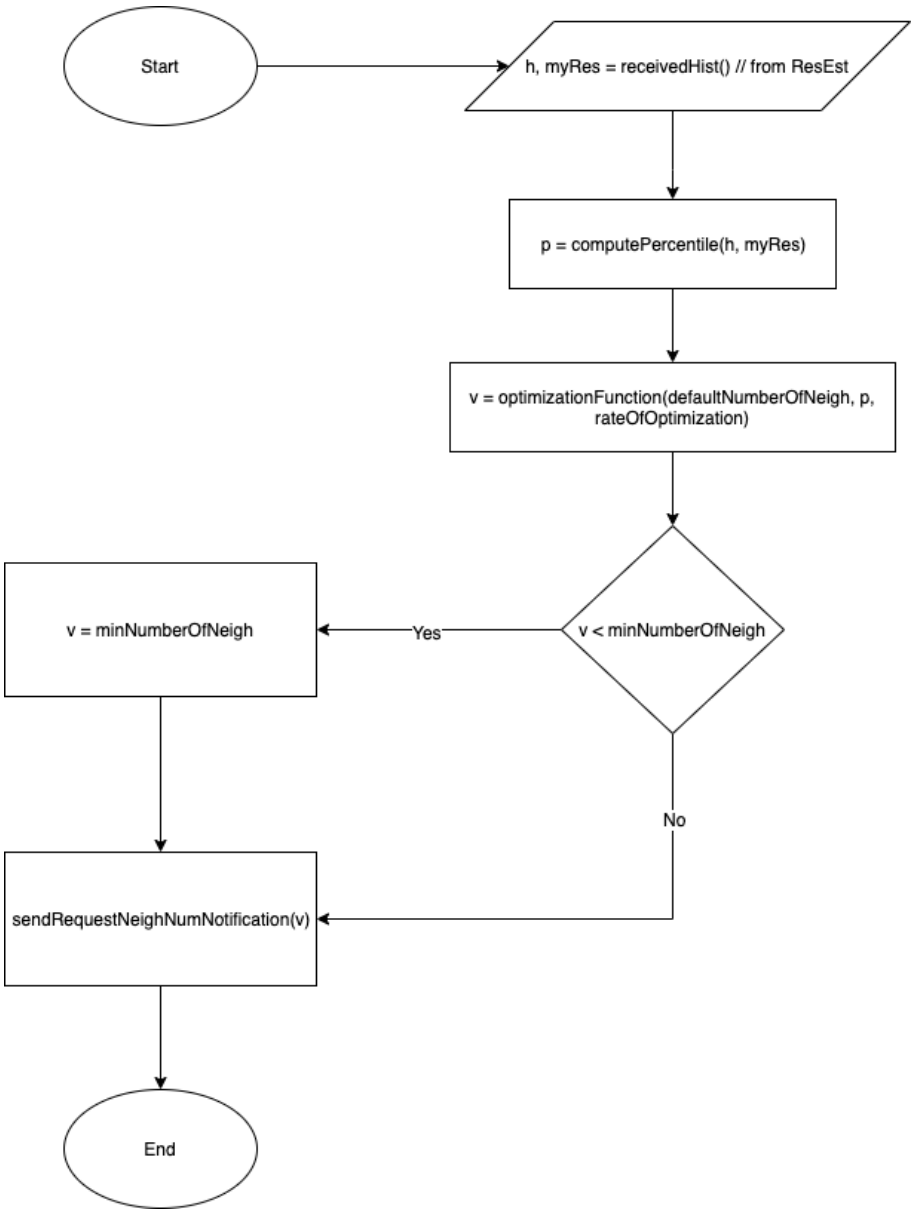


Figure 4.1: Flowchart of Proteus.

Algorithm 2: Proteus — Compute Percentile

```
1. Procedure computePercentile(h, myRes):
2.     belowMyResources ⟵ 0
3.     totalSamples ⟵ 0
4.     for (k, v in h) :
5.         if k ≤ myRes then:
6.             belowMyResources ⟵ belowMyResources + v
7.         totalSamples ⟵ totalSamples + v
8.     return belowMyResources/totalSamples
```

represents an estimation of the distribution of capacities of all the nodes in the network. To do so, the function responsible for this process (computePercentile(h, $myRes$)) receives the resource of the node running the algorithm ($myRes$) and the said histogram (h) as a map.

In Algorithm 2, we present the pseudocode for the computePercentile(h, $myRes$) function/procedure. We start by initializing a variable ($belowMyResources$) representing the number of nodes that have a lower capacity than the capacity of the node running the algorithm (Alg. 2 line 2). Then, we initialize the variable $totalSamples$ (Alg. 2 line 3), which represents the total number of samples contained in the histogram h.

Afterwards, we go through every entry in the histogram h (Alg. 2 line 4) and we check if the key (k) for that entry is lower than $myRes$ (Alg. 2 line 5). If so, it means that all the samples present in that entry have lower capacities than the node running the algorithm. Hence, if that is the case, we add the value of that entry (v) to $belowMyResources$ (Alg. 2 line 6). Then, before we leave that iteration, we add v to the total number of samples (Alg. 2 line 7).

Finally, after completing the loop, we calculate the percentile (Alg. 2 line 8), dividing the number of samples that are lower than $myRes$ ($belowMyResources$) by the total number of samples contained in the histogram ($totalSamples$), and then return that value.

4.3 Optimization Function

Now that the node knows an estimation of its percentile in the distribution of capacities of all the nodes, i.e., how powerful it is in relation to the other nodes in the network, we need to use that information to compute the number of neighbors that that node should have. We do this with a function that we called *optimization function*, since the objective of the function is to return the optimal number of neighbors for that node.

This function receives three parameters: i) the default number of neighbors ($defaultNumberOfNeigh$); ii) the percentile (p), calculated as described in the previous section; iii) and the rate of optimization ($rateOfOptimization$). The function then

returns an integer representing the appropriate number of neighbors for that node (disregarding the lower bound on the number of nodes, which is applied after, as seen in Figure 4.1).

The rate of optimization is the parameter that controls the slope of the optimization function. Hence, it controls what the maximum number of neighbors of the most powerful node (i.e., the node with a percentile of 1) will be. In more detail, a rate of optimization of x results in a function of optimization where the average node has $defaultNumberOfNeigh$ neighbors and the most powerful node will have $defaultNumberOfNeigh + x \times defaultNumberOfNeigh$ neighbors.

Therefore, being d the $defaultNumberOfNeigh$, p the percentile, and r the $rateOfOptimization$, in mathematical terms, the optimization function is denoted by the following equation:

$$optimizationFunction(d,p,r) = 2 \times d \times r \times p + d - d \times r \mid 0 \leq p \leq 1 \wedge d > 0 \wedge r > 0 \quad (4.1)$$

As an example, for a $defaultNumberOfNeigh$ of 7 and a $rateOfOptimization$ of 1, the optimization function would take the form displayed on Figure 4.2, only being dependent on the percentile (p) of the node. As we can see in the figure, the higher the percentile of a node (i.e., the more powerful a node is), the higher the number of neighbors to be kept by that node.

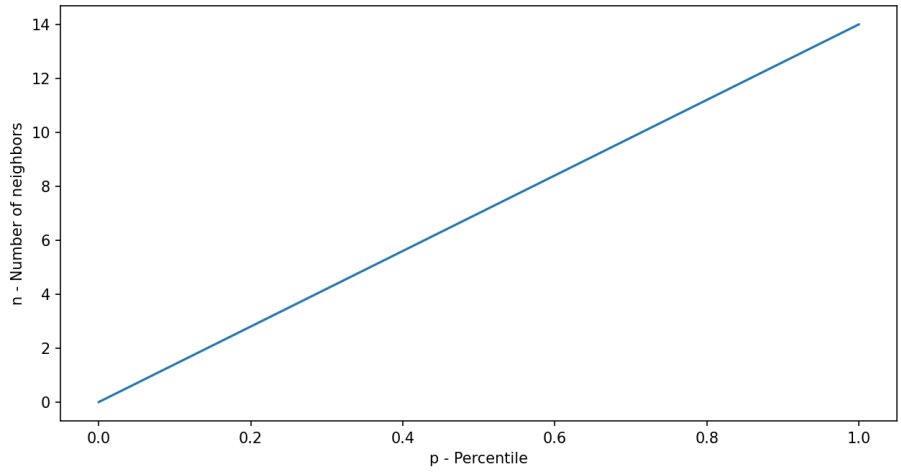


Figure 4.2: Optimization function — $f(p) = 14 \times p$.

It is worth noticing that for the average node ($p = 0.5$), the function would return 7 (which is the default number of neighbors). For the less powerful node ($p = 0$), the optimization function would return 0. As for the most powerful node ($p = 1$), the optimization function would return 14, which is equal to $defaultNumberOfNeigh + x \times defaultNumberOfNeigh$, as expected. Although, as said before, we do not allow any node to have a number of neighbors below the minimum number of neighbors allowed.

4.4 Improving Stability

With the Proteus base solution explained, in this section we are going to introduce an improvement to the algorithm. Even though this base solution of Proteus demonstrated good results in the experimental evaluation (Chapter 5), it can present some instability in the decided appropriate number of neighbors as a node collects samples from the network over time, especially when the histogram that is fed to Proteus (by ResEst, for example) is not very accurate.

Hence, especially when ResEst uses a low confidence level or a high acceptable maximum margin of error, Proteus might receive slightly different histograms, which result in different decisions about what the number of neighbors of that node should be. This is not good because it might result in Proteus sending different numbers of neighbors to the membership protocol over the time. For example, one time it might send a notification with a number of neighbors of 5, and then the next time sends 4, then 5 again, etc.

This situation ends up creating instability in the membership protocol, since it will have to add and remove neighbors frequently. This is especially adverse on HyParView [26], due to its symmetric views, because each time a node removes a neighbor, it causes that (ex) neighbor to remove it as neighbor as well, making that ex-neighbor search for a new neighbor—which ends up creating instability, as this chain event makes many nodes add and remove neighbors each time that Proteus runs.

Therefore, in order to minimize this effect, we developed an enhancement to Proteus, in which the algorithm saves the last five computed number of neighbors and sends an average of those five results to the membership protocol. That way, the neighbor number sent to the membership protocol will fluctuate less over time, as it is the average of the last five computed targets based on different samples collected by ResEst.

In the flowchart of Figure 4.3, we can see the simplified flow of execution of Proteus with this enhancement. Note that, even though we omitted that from the flowchart, on the first five iterations of Proteus (when there are not yet 5 computed values), Proteus sends to the membership protocol the average of the values that have already been computed. For example, in the second iteration, it sends the notification to the membership protocol with the average of the first and the second computed values.

4.5 Adapting membership protocols with Proteus

The way that the membership protocol performs the removal or addition of neighbors is not in the scope of Proteus, as it depends on which membership protocol is being employed and managed by Proteus. Proteus itself is agnostic to what membership protocol is managing the overlay, as long as it is properly adapted to receive a notification from Proteus with the desired number of neighbors, and performs the appropriate addition or removal of neighbors (upon receiving that notification).

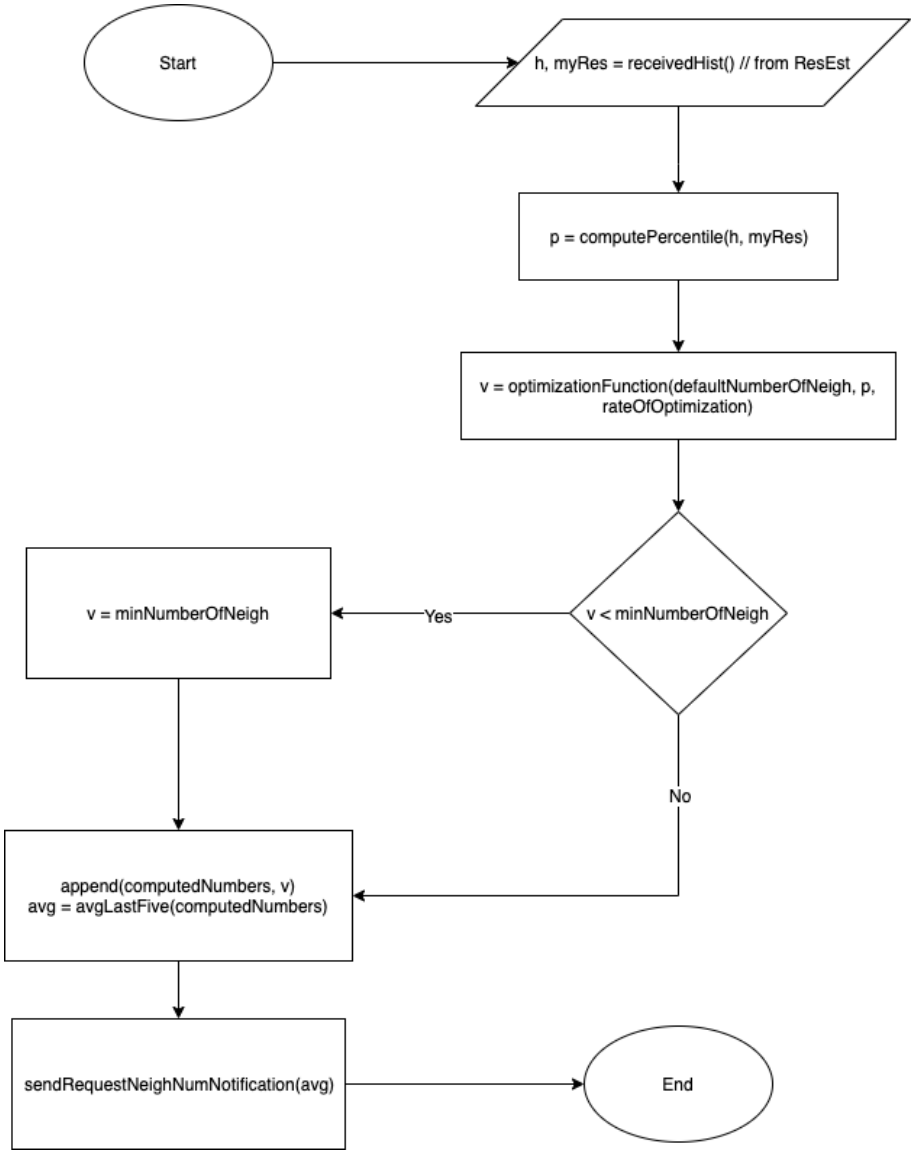


Figure 4.3: Flowchart of Proteus with stability improvement enhancement.

That said, we devised Proteus in this modular and agnostic way so that it can be easily added to a system running any membership protocol, with very small changes to that protocol. In Algorithm 3, we present the generic pseudocode for what actions the membership protocol should take upon receiving a notification from Proteus with the desired number of neighbors. The concrete implementation might be slightly different from protocol to protocol, but it should derive from the abstraction of the presented algorithm.

In Algorithm 3 we start by declaring one parameter—$removalMode$—which represents the criteria upon which neighbors might be removed from the view, (e.g., *random*, *by least powerful*, or *by most powerful*). Following that, we declare two variables—one representing the overlay view of the node (i.e., its neighbors) and the other one representing

Algorithm 3: Membership protocol adaptation upon receiving information from Proteus

```
//Parameters
    removalMode //The criteria for nodes to be removed from the view
//Variables
    view //Membership view
    maxNumberOfNeighbors //Maximum number of neighbors

1. Upon RequestNumberOfNeighborsNotification(n) do:
2.     maxNumberOfNeighbors ⟵ n.maxNrNeigh
3.     if maxNumberOfNeighbors < view.size then:
4.         TrimNeighbors(view.size – maxNumberOfNeighbors)

5. Procedure TrimNeighbors(toTrim):
6.     for (i in [0..toTrim]) :
7.         view ⟵ view.RemoveNeighborByCriteria(removalMode)
```

the maximum number of neighbors that that node can have.

When the membership protocol receives a notification from Proteus (Alg. 3, line 1), it first sets its maximum number of neighbors to the value contained in the notification (Alg. 3, line 2). Then, if that value is lower than the current size of the view (Alg. 3, line 3), we call a procedure to remove neighbors from that node (Alg. 3, line 4) so that the size of its view becomes equal to the maximum allowed number of neighbors. In the opposite case where the maximum number of neighbors is higher (or equal) than the current size of the node's view, we don't have to take action since the algorithm will naturally add neighbors (up to the threshold $maxNumberOfNeighbors$) throughout its normal execution.

The procedure TrimNeighbors($toTrim$) receives one argument which represents the number of neighbors to be removed (Alg. 3, line 5). It removes, one by one, $toTrim$ neighbors from the node's view, according to the $removalMode$ criteria.

4.6 Summary

In this chapter, we presented a decentralized algorithm—Proteus—which computes the appropriate number of neighbors that a node should have, according its own resources and an estimation of the distribution of capacities of the other nodes in the network. The algorithm does this by performing a calculation of the percentile of the node (in the distribution of capacities) and then uses that to compute the desired number of neighbors according to a parameterizable function.

We tested Proteus in emulation with 1000 nodes in various configurations. The details of its evaluation, its results, and analysis can be found in Chapter 5. The experiments

performed while evaluating the algorithm proved its validity, demonstrated its adaptability to sudden node capacity changes, and its capability of reducing latency while maintaining the reliability of the messages being disseminated through the network.

We also presented an enhancement to the algorithm, in order to reduce the variability of the results sent to the membership protocol over time. After evaluating Proteus with this optimization and comparing it with the base algorithm (in Chapter 5), we concluded that this enhancement proved to have a positive impact, since it reduced the variability of the average number of neighbors of the nodes over time, improving overall overlay network stability.

In addition to this, and even though out of the scope of Proteus itself, we presented an abstract way for the membership management protocols to correct the view of the node upon receiving a notification from Proteus. In other words, we presented the simple adaptations needed to be done in a membership protocol in order for it to be possible to manage by Proteus. This shows the versatility of Proteus, i.e., how our solution can be easily integrated in most peer-to-peer distributed systems.

With the solutions devised in these last two chapters—ResEst and Proteus—a node in a decentralized unstructured network can now adjust its number of neighbors according to its capacity relative to the capacities of the other nodes. That way, less resourceful nodes will process fewer messages and more resourceful nodes will process more messages, distributing the load fairly towards the overlay participants. Potentially, this can also be employed to improve the operation of structured overlay networks that exhibit more relaxed management of partial views, such as Kademlia [35] and Kelips [13].

This fair distribution of capacities presents advantages not only for each node but also for the network as a whole. Firstly, because since each node now contributes according to its capacity, nodes will be less overloaded with messages, being able to perform other actions while participating in the network. Additionally, and more importantly, since there is less overloading of nodes' capacities, the existence of bottlenecks in the network is reduced, hence reducing the latency of the messages being disseminated through the network.

Therefore, to prove those advantages, in the next chapter we are going to perform an experimental evaluation of the devised solutions. Firstly, of ResEst—using metrics on the quality of the estimation obtained, and also of the full solution (ResEst + Proteus), gathering different metrics to assess its impact in message dissemination.

5

Experimental Evaluation

In this chapter, we present our experimental work. While we have already reported on experimental work conducted through simulation for the ResEst algorithm previously (Chapter 3), here we focus on validating and understanding the operation of both proposed solutions—ResEst (Chapter 3) and Proteus (Chapter 4)—using prototypes built by us and emulation.

Therefore, we built a peer-to-peer distributed system prototype with our solution (ResEst + Proteus) integrated on it, aiming to compare its performance metrics against a similar distributed system without our integrated solution. The objective is to experimentally demonstrate that our solution proves to be valid and effectively allows us to obtain overall better performance.

This chapter is structured as following: in Section 5.1, we start by describing the prototype that we built to evaluate our solution, as well as the prototype we built to serve as benchmark; in Section 5.2, we describe the framework used to implement our prototype and cover the network constraints added to each node; in Section 5.3, we detail the experimental settings used—both on the hardware, and the software parameterization; then, in Section 5.4, we evaluate ResEst; after evaluating ResEst in isolation using emulation, we evaluate Proteus in Section 5.5, using ResEst with its best parameters (found when testing ResEst isolated); then, in Section 5.6, we take conclusions over the parameter values that result in the best performance of our solution, exposing such results; finally, in Section 5.7, we summarize the evaluation process, taking conclusions over the validity and performance of ResEst and Proteus in realistic scenarios.

5.1 Prototype Architecture

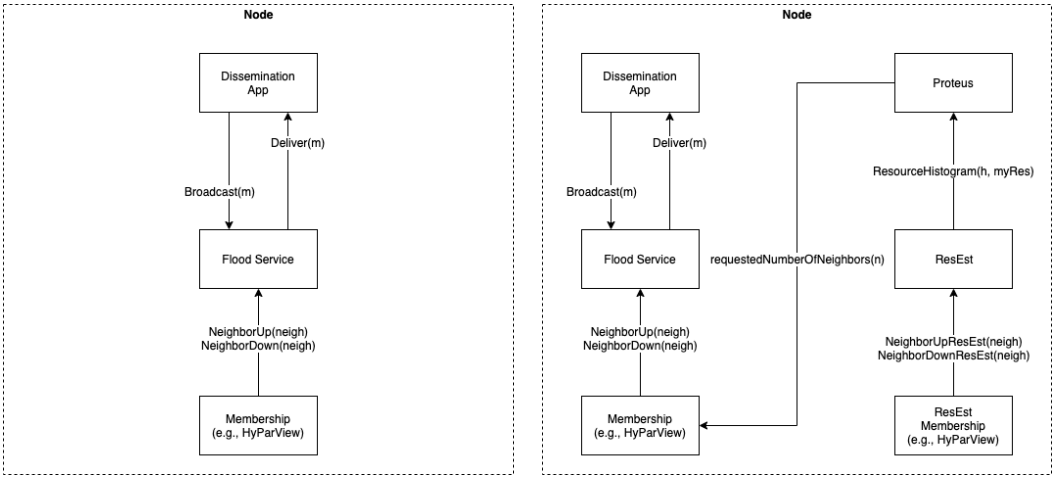
In order to evaluate the contributions produced in this thesis—ResEst (Chapter 3) and Proteus (Chapter 4), we devised two prototypes[1]. Both prototypes contain a membership management protocol (e.g., HyParView), in order for the nodes to build an unstructured

[1] https://github.com/hisetip/a-novel-approach-to-load-balancing-in-p2p-overlay-networks-for-edge-systems

overlay between them. Both prototypes also include a flood protocol and the consumer for that protocol (i.e., an application that broadcasts messages of a certain size according to a certain frequency using the flood protocol). This allows us to measure the impact of our approach on a particular peer-to-peer application.

The difference between both prototypes is that one of them contains only the three protocols described above (i.e., the membership management, the flood and the application/consumer protocol), as presented in Figure 5.1(a). We call this prototype the "base solution". We will use it as a baseline in our experiments.

Despite also having those three protocols, the second prototype (which we label "our solution") has ResEst and Proteus integrated, as presented in Figure 5.1(b). This is the prototype that contains the contributions devised in this thesis. Therefore, this is the prototype that we want to evaluate (using the base solution as the comparing baseline).



(a) Base solution. (b) Our solution.

Figure 5.1: Prototype diagrams.

Examining both prototype architectures in Figure 5.1, we can observe that the Dissemination App protocol sends requests to the Flood Service protocol for it to flood messages through the network. When the Flood Service receives a message, it delivers it to the Dissemination App protocol. The Membership protocol communicates with other nodes in order for each node to build a (partial) view of the network. This protocol is responsible to send a notification to Flood Service when it adds or removes a new neighbor to/from the view. Flood then uses these neighbors fed by the Membership protocol to flood messages to (according to flood dissemination approach, explained in Chapter 2).

Besides the protocols described above, our solution (presented in Figure 5.1(b)) contains three other protocols. One of them is Proteus, presented in Chapter 4. This protocol receives a notification from ResEst with the capacity of that node and a histogram representing an estimation of the node capacity distribution of the network. Upon receiving this information, it uses it to compute the appropriate number of neighbors for that node,

sending then a notification to the Membership protocol, so that that protocol can then adapt the size of its local partial view according to the number of neighbors that it should have (that depends on its own computational resources).

ResEst (presented in Chapter 3) receives a notification from the ResEst membership protocol every time this membership protocol adds or removes a new neighbor (in the same way as Flood Service receives notifications from the Membership protocol). ResEst uses these neighbors to regularly make estimations of the distribution of resources of the network in order to send them to Proteus, as explained in the previous paragraph.

The Membership protocol and the ResEst membership protocol can be similar—they can even be both implementations of HyParView [26] for example, only changing the type of notifications that they trigger to inform protocols above of changes in the local partial view (*NeighborUp(neigh)* vs *NeighborUpResEst(neigh)*). However, they have to be two independent instances, i.e., they can not communicate with each other. For this reason, in our solution, we can not, *a priori*, use just one instance of membership management protocol. This is the case because if we were to use just one instance of a membership protocol, then Proteus would be biasing the overlay that ResEst was using to gather information. This would result in a smaller chance of nodes with a smaller view to be sampled in the ResEst random walks (and on a bigger chance of nodes with bigger views to be sampled), damaging the ResEst estimation, which would then damage the Proteus operation. Hence, all the nodes of the overlay that ResEst gathers information from have to possess the same degree (i.e., the same number of neighbors).

5.2 Software Implementation

5.2.1 Babel Framework

We implemented the prototypes described above in Java, over the Babel[2] framework. This is a framework built for the development of distributed protocols and systems, which simplifies part of the network management code and provides us with useful tools to build our protocols, e.g., the ability to send and receive notifications in each protocol, to set timers per protocol, and to easily send and receive messages across nodes. In simpler terms, Babel allows us to implement a series of protocols to run in a node without having to worry about implementing aspects related with communication between those protocols (like notifications and requests), timers, etc.

In Babel, protocols communicate through messages with the same protocol of other nodes (different protocols do not communicate across nodes), and with notifications and requests with the different protocols within the same node. We used TCP for sending and receiving messages. The TCP connections are independent from protocol to protocol, e.g., node a can have a TCP connection open to node b in protocol p but not in protocol $p2$. We used a TCP multiplexer to make this possible.

[2] https://github.com/pfouto/babel-core

Sharing TCP connections between protocols would reduce the bandwidth used (since there would be fewer connections being opened and closed). The reason why protocols can't do this is because, as mentioned above, the Membership protocol and the ResEst membership protocol have to be independent, which means that closing a connection in the first protocol can not close the connection in the second one and vice-versa. That said, to minimize the bandwidth used to open and close TCP connections, we tried to keep protocol TCP connections open until that protocol would not need that connection anymore. For example, in Flood Service, we open a connection every time a *NeighborUp* notification is received, and close a connection in the case of a *NeighborDown* notification.

5.2.2 Node Network Constraints

With the aim of reproducing a distributed system operating in a realistic scenario, we added two types of network constraints to each node—fixed latency and bandwidth.

5.2.2.1 Latency

For each pair of nodes in the network, we established a fixed value of latency between them. The upload latency of one node of that pair has the same value as the download latency of the other node. We did this in order to simulate the variable distance that can exist between them. That way, a message can take $20ms$ from node a to node b, but one sent from node a to node c might take $200ms$. The first case can represent two nodes that are close to each other (e.g., same country), while in the second case nodes would in different continents, for example.

We distributed latencies symmetrically through all the possible existing links in the system using the *TC* tool from Linux, according to a uniform distribution, with a minimum of $10ms$ and a maximum of $250ms$. We chose these values with the aim of realistically representing nodes that are geographically close to each other (e.g., in the same state) and nodes that are further away from each other (e.g., in different continents).

Even though we aim at creating a network of heterogeneous nodes, it is not our concern that the latencies between them have any connection to their intrinsic capacities (in whatever metric we are measuring capacities on). As in a realistic scenario, two powerful nodes can be geographically far away from each other, resulting in a high latency between them, just like two weak nodes can be very close to each other, resulting in a low latency between them.

5.2.2.2 Bandwidth Limitation

As mentioned in the previous paragraph, we aim at creating a network of heterogeneous nodes. Our solution is agnostic to what metric of capacity it is used. For example, we could use the number of CPUs of each node as a capacity metric, or the upload or download bandwidth of each node, or the total RAM available to each one, etc.

For our experiments, we decided to use upload bandwidth as the capacity metric. Therefore, in our network, there are nodes with high upload bandwidth (up to $20MB/s$) and nodes with very low upload bandwidth (as low as $200KB/s$). We chose these maximum and minimum bandwidth values based on the size and the frequency of the messages flooded per node, detailed in Section 5.3.2.5. We set the maximum upload bandwidth ($20MB/s$) to be double the value of the payload used in the most stressful experiment we did in the system. That way, the most powerful node will never be saturated, even in the most stressful condition. For the minimum upload bandwidth, we set it to 100 times less than the maximum, so that we could have a realistic range of bandwidths. This way, the weakest node in the network would have an upload bandwidth of 100 times less than the most powerful node, making it easily saturated (in the most stressful experiment).

We chose upload bandwidth as the metric for the capacity of a node for two reasons. Firstly, because the bandwidth of a node directly limits the number of messages that that node can send per second. Due to the fact that we use flood dissemination in our experiment, that results in nodes with low upload bandwidth to be *easily* saturated, unlike nodes with high bandwidth.

The second reason why we used upload bandwidth as a metric (instead of CPU, for example), is that, unlike CPU, upload bandwidth can be precisely limited without affecting other ongoing processes required for the node to operate (e.g., OS processes). In addition, since we are using flood to measure metrics like latency and reliability, limiting upload bandwidth would have the same effect as limiting CPU (if we could do it safely), because using either of these strategies, the nodes would theoretically send a number of messages proportional to how limited they are.

Hence, for the rest of the chapter, when we mention the *capacity* (or *resource* level) of a node, we are referring to its upload bandwidth. That said, a node being weak means that it presents a low upload bandwidth (i.e., close to $200KB/s$), while a powerful node would have a bandwidth close to $20MB$. The distribution of capacities used for the nodes is detailed in Section 5.3.2.1.

5.3 Experimental Settings

5.3.1 Hardware

In order to emulate 1000 nodes, we used 50 different machines in the Grid5000-Nancy[3] cluster. Hence, every machine contained 20 docker containers, each container representing one node. Each of the 1000 docker containers had its own IP and ran independently from each other.

Since all the machines used were from the same cluster (*gros*), we can assume that there are no inherent significant latency and bandwidth constraints. Each one of the 50

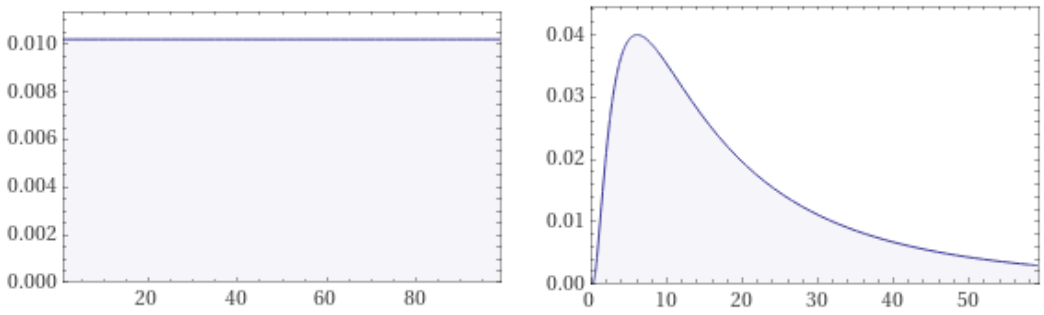
[3] https://www.grid5000.fr/w/Nancy:Hardware

machines used have the same specifications, with Intel Xeon Gold 5220 (Cascade Lake-SP, 2.20GHz, 18 cores) as CPU, 96 GB of memory, and 480 GB SSD SATA of disk. All the machines were connected through Ethernet at 25 Gb/s.

5.3.2 Parameterization

5.3.2.1 Resource Distribution

We used two different resource distributions for our experiments—uniform and lognormal. The parameterization of these distributions is equal to the parameterization used in the resource distributions of ResEst simulations (presented in Section 3.3). Therefore, the uniform distribution is controlled by two values—min and max—which limit the range of possible values. The values used to generate the uniform distribution were $min = 1$ and $max = 99$. The lognormal distribution is controlled by two values—μ and σ—which respectively encode the mean and the standard deviation of the variable's natural logarithm. The values used to generate the configuration of lognormal capacities were $\mu = 2.8$ and $\sigma = 1$. In Figure 5.2(a), we present the probability distribution function of the uniform distribution, where the average of the node capacities is 50, with the number of nodes weaker than the average approximately equal to the number of nodes more powerful than this. In Figure 5.2(b) we present the probability distribution function of the lognormal distribution, where a more realistic network is represented. The selection of these specific parameters ($\mu = 2.8$ and $\sigma = 1$) was carried out experimentally, in order to simulate a network where there are many moderately weak nodes and few powerful nodes—similar to a realistic scenario [7].



(a) Uniform distribution.

(b) Lognormal distribution.

Figure 5.2: Distribution of probability functions.

As mentioned in Section 5.2.2.2, we assigned the upload bandwidth of each node as its capacity, with a maximum of 20 MB/s and a minimum of 200 KB/s. However, as one can observe in Figures 5.2(a) and 5.2(b), the values sampled from those distributions would not be in that range (e.g., the uniform distribution ranges between $[0, 100]$, instead of $[200, 200000]$). Therefore, we had to transpose each of the 1000 samples generated from each distribution to the appropriate bandwidth range. Hence, after generating all

1000 random samples from each distribution, we set the lower obtained sample of each distribution as 200 and the higher obtained sample as 200000. All the other samples were fit proportionally in that range, maintaining the correct distribution.

5.3.2.2 ResEst Protocol

In our experiments, we used the following parameters for the ResEst protocol.

- **Maximum Margin of Error** This maximum margin of error is a percentage referring to the maximum acceptable error rate that the computed estimated mean of collected samples can have for the random walk to finish (i.e., to activate the stopping criterion).

  The values used for this parameter were 25% and 15%.

- **Confidence Level** This confidence level consists of a percentage that indicates the confidence in the estimated value of the mean on the collected samples. It is used to calculate the margin of error of the estimated mean of collected samples. The higher the confidence level, the higher the margin of error.

  The values used for this parameter were 90% and 95%.

- **Query Frequency (s)** The query frequency represents the frequency upon which a node triggers a random-walk in order to perform an estimation.

  We used 60 seconds as query frequency, which means that every 60 seconds, the node would trigger a random-walk.

To control the execution of the experiments, we also use the following additional parameters (that are not directly related with the operation of ResEst).

- **First Node Launch Timestamp (s)** This is the time (since epoch) that the first node in the system was launched.

- **Start Time (s)** The first random-walk is triggered at the timestamp $l + s$, l being the *first node launch timestamp* and s being the *start time*. This timer is necessary since the 1000 nodes are not launched exactly at the same time. Hence, the first nodes can not start triggering random-walks right away—they have to wait for all nodes to be launched. This way, all 1000 nodes start triggering random-walks at the same time.

  We used 2000 seconds as start time. This parameter was estimated after the time that it takes all 1000 nodes to start-up.

- **Running Time (s)** The running time is the amount of time that the node will be running ResEst.

  We used 1200 seconds as running time, so that each node would trigger 20 random-walks.

5.3.2.3 Proteus Protocol

In our experiments, we used the following parameters for the Proteus protocol.

- **Default Number Of Neighbors** This parameter represents the default number of neighbors of a node. It should have the same value as the membership protocol starting maximum view size.

  We used 7 as the default number of neighbors. This number was chosen in order to probabilistically ensure that a flooded message would be delivered to all nodes, since $ln(1000) = \pm 7$ [9].

- **Minimum Number Of Neighbors** This represents the minimum number of neighbors that a node can have. Proteus will never send a notification to the membership protocol with a value lower than this parameter.

  We used 4 as the minimum number of neighbor since every node needs to have at least 4 neighbors to ensure the connectivity of the overlay[47].

- **Mode** This is the mode that Proteus is operating on—either the simple algorithm or the algorithm with the stability improvement (that considers the average of the last 5 computed number of neighbors, as presented in Section 4.4 of Chapter 4).

  Therefore, for this parameter we used *simple* (for the simple algorithm) and *average* (for the algorithm with stability improvement).

- **Rate of Optimization** The rate of optimization is the parameter that controls the slope of the optimization function. Hence, it controls what the maximum number of neighbors of the most powerful node can be.

  We used 1.0, 0.5 and 1.5 as values for this parameter.

5.3.2.4 Membership Protocols

We evaluated our solution using HyParView [26] and Cyclon [52] as the membership protocols. Each of them were used as the membership protocol itself and as the ResEst membership protocol (i.e., the membership protocol that feeds ResEst). We used a maximum view size of 7 in both protocols, to probabilistically ensure that a flooded message would be delivered to all nodes [9].

As for the other parameters of each membership protocol, in the case of HyParView, we used the following values:

- **Active View** (maximum active nodes): 7
- **Passive View** (maximum passive nodes): 30
- **ARWL** (active random walk length): 3

- **PWRL** (passive random walk length): 6
- **Shuffle Time** (timeout for shuffle): 8000 ms
- **Hello Backoff** (timeout for hello messages): 7000 ms
- **kActive** (number of active nodes to exchange on shuffle): 2
- **kPassive** (number of passive nodes to exchange on shuffle): 3
- **Node Removal Strategy** (criteria by which nodes are dropped upon Proteus notification): "random", "leastPowerful", "mostPowerful"

For Cyclon, we used the following values for its parameters:

- **Cache View Size** (capacity of cache/view): 7
- **Shuffle Time** (timeout for shuffle): 2000 ms
- **Shuffle Length** (number of nodes exchanged in shuffle): 4
- **Node Removal Strategy** (criteria by which nodes are dropped upon Proteus notification): "random"

5.3.2.5 Dissemination Application

The Dissemination Application is responsible for regularly broadcasting messages, using the Flood Service. In our experiments, we used the following parameters.

- **Messages per Second** This parameter represents the number of broadcasted messages per second by the protocol.

  We used 0.1 msg/s as the value of this parameter. This results in every node broadcasting a message every 10 seconds.

- **Payload Size (B)** This parameter is the number of bytes for the payload of each broadcasted message.

  We used 100 B, 1000 B, and 10000 B as possible values for this parameter.

As discerned for the experiments with ResEst, we further use the following parameters to control the execution of experiments.

- **First Node Launch Timestamp (s)** As in ResEst, this is the time (since epoch) that the first node in the system was launched. The value of this parameter must be equal to the namesake parameter in ResEst.

- **Start Time (s)** The first message is broadcasted at the timestamp $l + s$, l being the *first node launch timestamp* and s being the *start time*. As in ResEst, this wait is necessary since the 1000 nodes are not launched exactly at the same time. Hence, the first nodes can not start broadcasting messages right away since not every node would receive the disseminated message—they have to wait for all nodes to be launched. This way, all 1000 nodes start broadcasting at the same time.

  In the prototype of our solution (i.e., the solution with ResEst and Proteus), we used 3200 seconds as start time. This parameter is computed after the time that it takes all 1000 nodes to start-up (2000 s) + the running time of ResEst (1200 s). This is because we only want to disseminate messages once ResEst (and, therefore, Proteus) are done running, so that each node has already adapted its number of neighbors according to its capacity.

  In the case of the base solution, we used 2000 seconds as start time. As there is no ResEst and Proteus running, we don't have to wait for the neighbors to adapt their number of neighbors according to their capacity, reducing the total experiment time.

- **Running Time (s)** The running time is the amount of time during which the node will be broadcasting messages.

  We used 400 seconds as running time, so that each node would broadcast around 40 messages, to which we concluded is a sufficient number of messages to gather performance metrics.

Therefore, an experiment with nodes running the prototype of our solution will take a total of 3600 s to complete (1 hour), while experiments with nodes running the base solution will take a total of 2400 s to complete (40 minutes).

5.4 ResEst Evaluation

Doing experiments through simulation, like the ones reported previously in Chapter 3, does not take into account the proprieties of the network nor the impact of membership protocol being used. Therefore, in order to take these into account and prove the validity of ResEst in a realistic scenario, we tested it in emulation, with 1000 nodes.

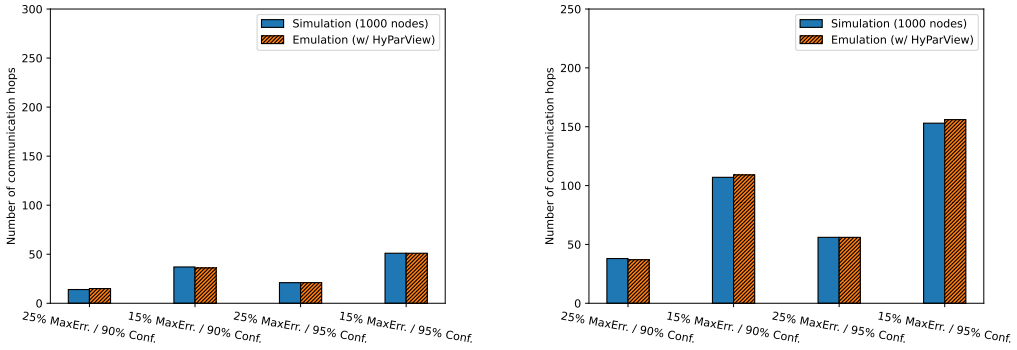
The parameters used were the same as in the simulation experiments (presented in Chapter 3), as described in Section 5.3.2.2. During the 1200 seconds of experiment time, each node performed 20 estimations. In Tables 5.1 and 5.2, we summarize our experimental results for the two distributions used. These tables follow the same structure as Tables 3.1 and 3.2 (from ResEst simulation results, in Chapter 3) with the difference that the column with the number of nodes (now always 1000) was replaced with a column indicating the membership protocol used in that experiment—either HyParView or Cyclon.

Parameters			Average Results of 10 Experiments	
Max. Margin of Error	Confidence Level	Membership Protocol	Number of Hops	Histogram Error
25%	90%	HyParView	15	16.46%
25%	90%	Cyclon	15	16.80%
15%	90%	HyParView	36	10.98%
15%	90%	Cyclon	36	11.07%
25%	95%	HyParView	21	14.83%
25%	95%	Cyclon	20	14.61%
15%	95%	HyParView	51	9.02%
15%	95%	Cyclon	53	8.94%

Table 5.1: ResEst emulation results for uniform distribution.

Parameters			Average Results of 10 Experiments	
Max. Margin of Error	Confidence Level	Membership Protocol	Number of Hops	Histogram Error
25%	90%	HyParView	37	9.08%
25%	90%	Cyclon	38	9.96%
15%	90%	HyParView	109	5.32%
15%	90%	Cyclon	110	5.54%
25%	95%	HyParView	56	7.82%
25%	95%	Cyclon	55	8.13%
15%	95%	HyParView	156	4.50%
15%	95%	Cyclon	159	4.34%

Table 5.2: ResEst emulation results for lognormal distribution.



(a) Uniform distribution.

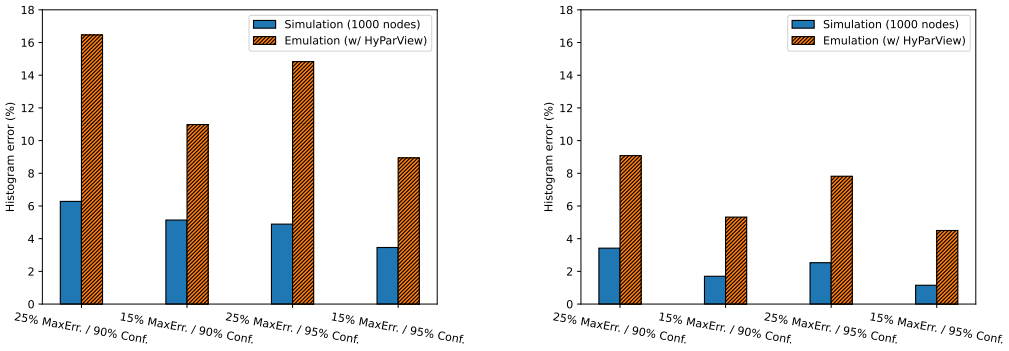
(b) Lognormal distribution.

Figure 5.3: Number of communication hops for uniform and lognormal distributions.

Tables 5.1 and 5.2 confirm that regardless of the distribution or membership protocol used, the algorithm proved to be valid in emulation, being able to compute estimated histograms whose error is smaller than the maximum error margin used, as in the simulation experiments (presented in Chapter 3). As one can see in Figure 5.3, the number of hops needed to obtain a correct estimation was virtually the same for the simulation and for the emulation experiments.

On the other hand, even though always below the maximum margin of error parameter, the average histogram errors are higher in emulation than in simulation, as one can see in Figure 5.4. We think this happens because the membership protocols used do not build a perfectly random unstructured overlay. However, we are not exactly sure of why this phenomenon occurs. Doing more experiments in order to find the exact reason for this is left for future work.

On other note, in theory, for ResEst to be able to make a valid estimation of the



(a) Uniform distribution.

(b) Lognormal distribution.

Figure 5.4: Histogram error for uniform and lognormal distributions.

capacities of the nodes of the network, every node needs to have the same probability of being found during the random-walk. Hence, as mentioned in Chapter 3, ResEst can only be used with overlays that guarantee symmetric views (e.g., HyParView). However, we wanted to test how ResEst would perform with Cyclon—a widely used membership protocol that does not guarantee symmetric views. In other words, we wanted to verify if most nodes in the Cyclon overlay have an in-degree close enough to its out-degree so that ResEst would return a valid estimation with a similar histogram error to the experiments with HyParView.

As we can observe in Tables 5.1 and 5.2, ResEst presented very similar results with Cyclon and HyParView across all 16 parameter combinations—both in the average number of hops and in the average histogram error—suggesting that ResEst can be safely used with Cyclon. Nevertheless, since they show very similar results, to be safe, we opted to mostly use HyParView when evaluating our full solution prototype (ResEst + Proteus), in the following section.

5.5 Proteus Evaluation

In this section we evaluate and analyze the performance of our prototype, i.e., Proteus with ResEst, as presented in Figure 5.1(b). As mentioned in Section 5.3.1, the experiments were performed with 1000 nodes, and the parameterization used is the one described in Section 5.3.2.

It is worth noting that, due to the high number of different parameters, we could not test every different combination of them, as we did when evaluating ResEst standalone for example, since that would be unfeasible due to the amount of time needed to execute all experiments. Consequently, we evaluated what we thought were the best or more relevant combination of parameters, making sure that all the values of each parameter were included. We are going to present and analyze the results from those experiments,

by metric, in the rest of this section.

5.5.1 Load per Node Capacity

The first metric that we are going to analyze is the load per node capacity/resource, which is the amount of flood messages that each node sent (using the Flood Service), given its capacity. To do such analysis, we present a series of scatter plots (in Figures 5.5, 5.6, and 5.7) in which, for each one of them, the horizontal axis represents the capacity of the node and the vertical axis represents the number of messages it sent. Hence, each node is represented by a data point (a dot in the figure). In order to fit all 1000 data points in each figure, we set the transparency of each one to a low value. Therefore, in the figures, the more opaque a dot is, the more data points it represents.

We used the following parameterization in the experiments whose results are discussed in this section. For the base solution prototype, we used a dissemination message payload of 100 bytes and HyParView as the membership protocol. For our solution prototype, we used a dissemination message payload of 100 bytes, HyParView as the membership protocol with the *random* removal strategy, HyParView as the ResEst membership protocol, a ResEst maximum margin of error of 25% and confidence level of 90%, and a Proteus rate of optimization of 1.0. Table 5.3 summarizes this parameterization.

Table 5.3: Parameterization of the experiments whose results are discussed in Section 5.5.1.

Prototype	**Protocol**	**Parameter**	**Value**
Base solution prototype	Dissemination (Flood Service)	Message payload	100 bytes
	Membership protocol	Name	HyParView
Our solution prototype	Dissemination (Flood Service)	Message payload	100 bytes
	Membership protocol	Name Removal strategy	HyParView random
	ResEst membership protocol	Name	HyParView
	ResEst	Max. margin of error Confidence level	25% 90%
	Proteus	Rate of optimization	1.0

In Figure 5.5, we can see the load per node capacity of the base solution. The first insight that can be taken is that most of the nodes have few resources. This is the case because we are using a lognormal distribution, as shown in Figure 5.2(b). More importantly for this analysis, however, is the observation that most of the nodes sent the same number of flood messages. This makes sense, since most of the nodes have the same number of neighbors, hence, each one of them will flood roughly the same number of messages.

However, our goal is that each node sends a number of messages proportional to how powerful it is in relation to the other nodes in the overlay—and that is exactly what happened with the experiments of our prototype. In Figure 5.6, one can see the load per node capacity of our solution (using Proteus *simple* mode) with a uniform resource distribution. The results show that there is a direct correlation between the resources of a node and the

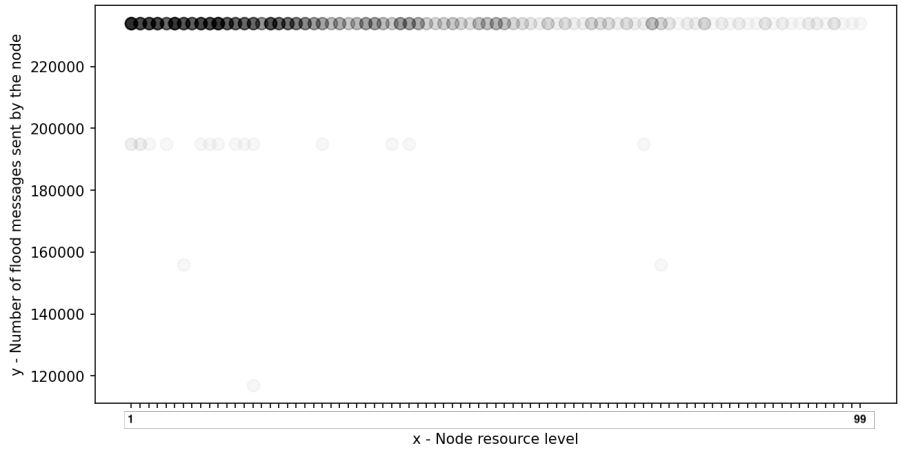


Figure 5.5: Load per node capacity of baseline solution, with lognormal distribution of resources.

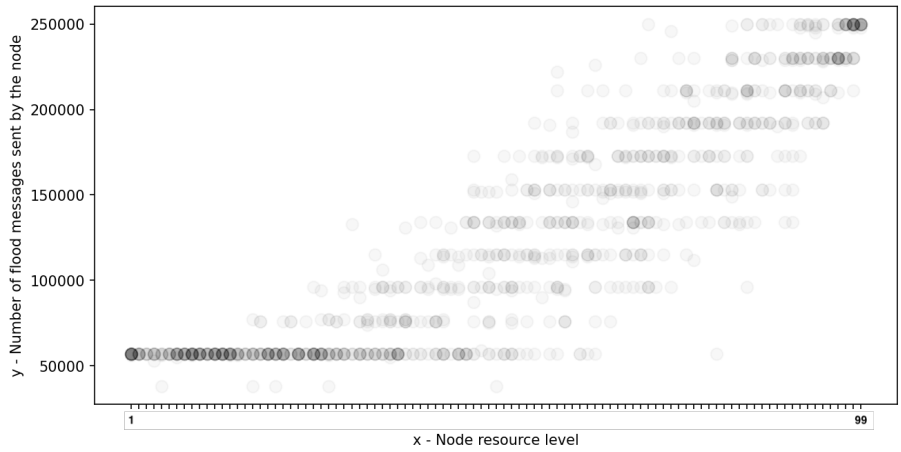


Figure 5.6: Load per node capacity of our solution, using Proteus *simple* mode, with **uniform** distribution of resources.

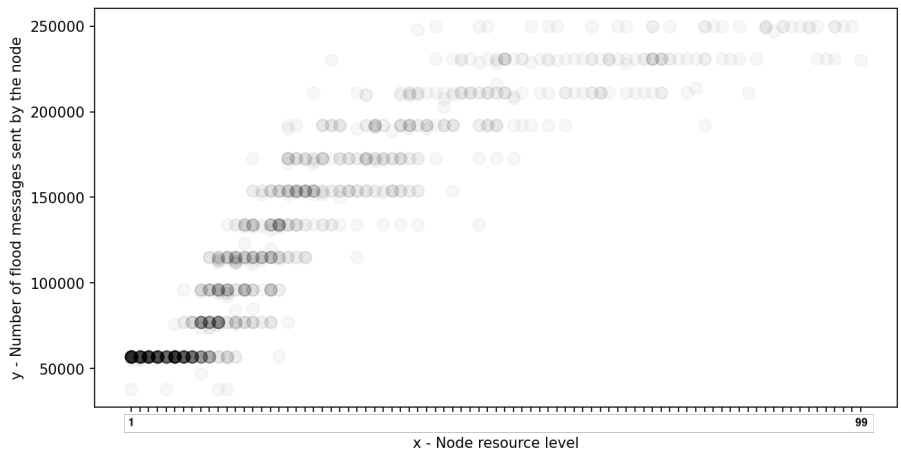


Figure 5.7: Load per node capacity of our solution, using Proteus *simple* mode, with **lognormal** distribution of resources.

number of messages that it sent—the nodes with more resources send more messages and ones with less resources send fewer messages, exactly as supposed. Moreover, it can be observed that the nodes below a certain resource level send the same amount of messages, which can be explained by the use of a minimum number of neighbors, that sets a lower bound to the number of messages sent by each node, to ensure to correct behavior of the epidemic protocol and the connectivity of the managed overlay network.

In Figure 5.7, we can see the results of an experiment performed with the same parameters as the one in Figure 5.6, but with a different distribution—lognormal instead of uniform. Here, we can notice that the number of messages per resource level grows much faster in the beginning, taking the shape of a logarithmic function, as expected. This is the case because since most of the nodes have few resources (due to the lognormal distribution), a node with a medium amount of resources (e.g., 50) is already more powerful than almost every other node (i.e., it is already in a very high percentile)—hence, it has to send a much larger amount of messages than weaker nodes, in order to compensate for their low number of neighbors. This does not happen in the uniform distribution since the number of weak nodes is roughly the same as the number of powerful ones. For example, a node with a capacity of 50 in our uniform distribution should be considered as average (since the number of nodes with less resources than that node should be approximately the same as the number of nodes with more resources), sending an average number of messages; however, a node with a capacity of 50 in our a lognormal distribution is a very powerful node, since most of the nodes are less powerful than that node, hence it should send a very high number of messages.

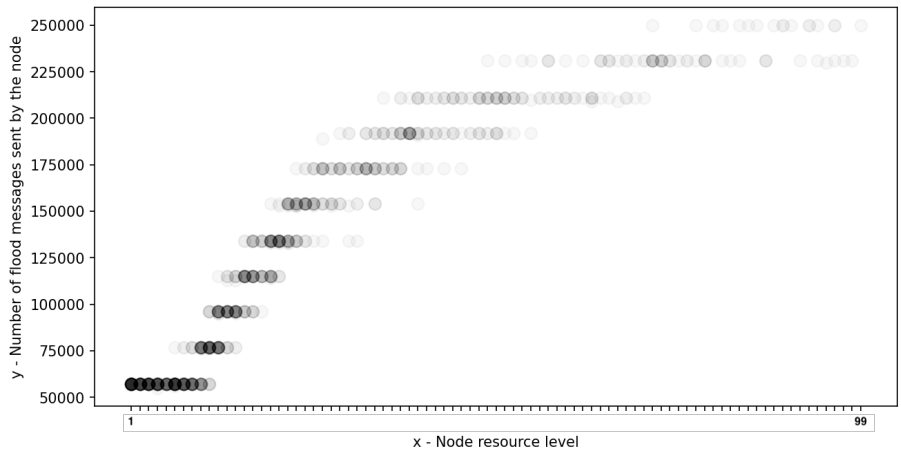


Figure 5.8: Load per node capacity of our solution prototype using Proteus *average* mode, with the lognormal distribution.

Therefore, we can conclude by Figures 5.6 and 5.7 that the first objective of this thesis was successfully achieved—each node is contributing to the network according to its own resources, taking into account how powerful it is in relation to the other nodes in the network. Nevertheless, the distribution of load by resource level can be even better. One

can note that the data points are very scattered in both Figures 5.6 and 5.7. This is the case because we are using the *simple* mode in Proteus, which results in instability in the number of neighbors of the nodes, making many nodes with the same capacities send slightly different numbers of messages, creating the rather dispersed scatter plot we can see in those figures. This aspect can be mitigated by using the *average* mode in Proteus (as explained in Section 4.4 of Chapter 4). Thus, in Figure 5.8, we can observe the results with this mode. As one can notice, the scatter plot is less dispersed, since there is less instability in the number of neighbors of each node.

5.5.2 Number of neighbors variability over time

In this section we are going to analyze the average number of neighbors variability over time, which indicates how many neighbors were added and removed from the nodes over time (on average). We used the following parameterization for our solution prototype in the experiments whose results are discussed in this section: we used the lognormal resource distribution, a dissemination message payload of 100 bytes, HyParView as the membership protocol with the *random* removal strategy, HyParView as the ResEst membership protocol, a ResEst maximum margin of error of 25% and confidence level of 90%, and a Proteus rate of optimization of 1.0. Table 5.4 summarizes this parameterization.

Table 5.4: Parameterization of the experiments whose results are discussed in Section 5.5.2.

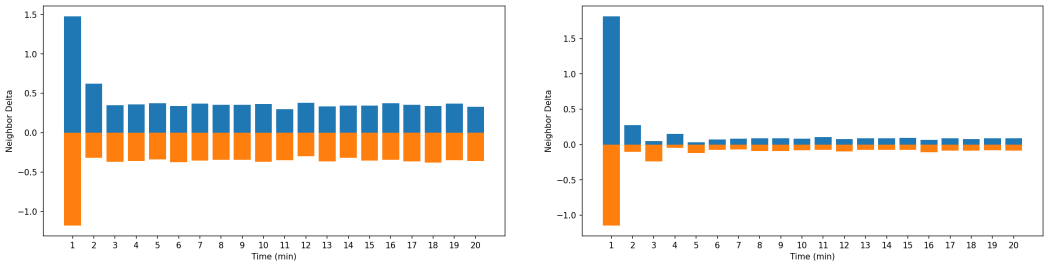
Prototype	**Protocol**	**Parameter**	**Value**
Our solution prototype	-	Resource distribution	lognormal
	Dissemination (Flood Service)	Message payload	100 bytes
	Membership protocol	Name Removal strategy	HyParView random
	ResEst membership protocol	Name	HyParView
	ResEst	Max. margin of error Confidence level	25% 90%
	Proteus	Rate of optimization	1.0

In Figure 5.9 we can see this variability over the course the 20 minutes that ResEst and Proteus were running during the experiments. On the vertical axis of each histogram, we report the average number of neighbors that were added to the nodes in that minute, in blue, and the average number of neighbors that were removed from the nodes, in orange.

As one can observe in Figure 5.9, the first bar ($Time = 1$ min) in both histograms is larger than the remaining ones (both for the adding and the removal of neighbors). This can be explained by the fact that when Proteus runs for the first time, every neighbor has the default number of neighbors, which is likely not appropriate, making it suddenly change its number of neighbors to the appropriate number. Another observation is that the remaining bars (from 2 up to 20 minutes) are notably constant. This is the case because those bars represent small adjustments to the nodes' number of neighbors, due to the resource estimations made by ResEst being slightly different every time, making

Proteus send to the membership protocol notifications to change the number of neighbors with a slightly different number of neighbors each of these times. For example, Proteus might send a notification requesting that the number of neighbors is 8, then 7, then 8 again, etc.

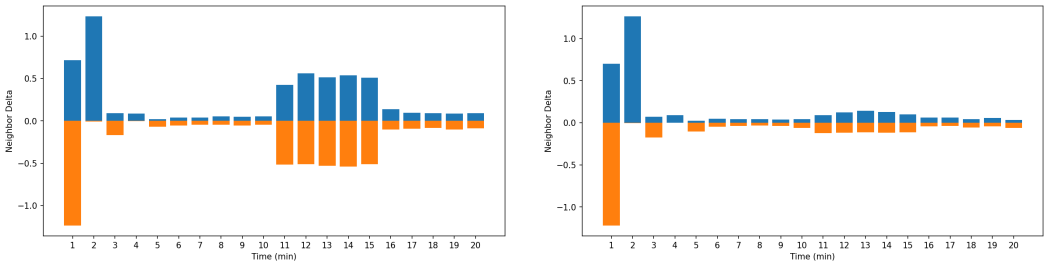
To reduce that instability, as explained in Section 4.4 of Chapter 4, we developed the *average* mode in Proteus, which makes the notifications sent by Proteus to the membership protocol more stable, since they rely on the average of the last 5 computed values instead of sending the recently computed value each time. The results from this optimization to our solution can be observed in Figure 5.9(b), in which we used Proteus *average* mode, comparing it with Figure 5.9(a), in which we used Proteus *simple*. We can notice that, when using the *average* mode, after the first five minutes, the average number of neighbors variability stabilized in very low values (at around 0.1), meaning that, every one minute, on average each node removed and added 0.1 neighbors, 4 times less than when using *simple* mode, where on average each node removed and added 0.4 neighbors. Therefore, since it provided better results, we decided to use Proteus *average* mode in all the remaining experiments whose results are discussed throughout the rest of this chapter.



(a) Using Proteus *simple* mode.

(b) Using Proteus *average* mode.

Figure 5.9: Average number of neighbors variability over time.



(a) The less powerful nodes were turned into more powerful nodes at minute 10.

(b) The most powerful nodes were turned into less powerful nodes at minute 10.

Figure 5.10: Average number of neighbors variability over time while modifying node resources mid-execution, while using Proteus *average* mode.

We now show how our proposed solution can react to changes in the resource distribution of the network. To that end, in the experiments whose results are presented in Figure 5.10(a) and Figure 5.10(b) we changed the capacities of some nodes in order to see how that would affect the average node variability over time. More precisely, in the experiment of Figure 5.10(a), we turned every node with a capacity below 20 into a node with a capacity above 80. This has the same practical effect of removing the weaker nodes and replacing them for very powerful nodes. Per contra, in the experiment of Figure 5.10(b), we did the opposite, turning every node with a capacity above 80 into a node with a capacity below 20, which has the same practical effect of removing the most powerful nodes and replacing them for very weak ones.

As it can be observed in Figure 5.10, in both cases there was a spike in the variability when the change was introduced (at about 10-11 minutes), since many nodes had to change their number of neighbors, adding and removing them. This high variability kept on going for roughly 5 minutes (i.e., five ResEst/Proteus runs), until it finally stabilized in the same values as before the change was introduced, proving the validity of our solution upon changing the capacities of the nodes mid-execution. The reason why it took 5 minutes until the variability stabilized is that, since we are using the *average* mode on Proteus, Proteus sends the average of the last 5 computed values to the membership protocol, and before the 5 minutes have passed, those 5 values still include the computed values from before the change was made. However, we see this as a positive feature. Even though it takes 5 iterations to stabilize, the hit on the network is progressive, since nodes will not change their number of neighbors abruptly. Another detail that can be noticed when comparing the results in Figure 5.10 is that the variability upon resource modification in Figure 5.10(a) had a bigger effect when compared with the results on Figure 5.10(b), due to the fact that in the former the number of nodes who suffered a resource modification is much larger than in the latter, due to the use of our lognormal distribution (as described in Section 5.3.2.1).

5.5.3 Average Neighbor Number Error

The average neighbor number error is the average of the neighbor number errors of every node in the network. The neighbor number error is a metric that indicates the absolute difference between the number of neighbors that a node has in the end of the experiment and the number of neighbors that it should have (i.e., the number of neighbors that that node should have if it had perfect information about the resource distribution across the network). The number of neighbors that the node should have is computed running Proteus with the real histogram of resources of the network. That is to say, if ResEst would make perfect estimations of the distribution of capacities of the network, the neighbor number error of each node would always be 0.

We used the following parameterization for our solution prototype in the experiments whose results are discussed in this section: we used the lognormal resource distribution,

a dissemination message payload of 100 bytes, HyParView as the membership protocol with the *random* removal strategy, HyParView as the ResEst membership protocol, and the Proteus *average* mode. Table 5.5 summarizes this parameterization.

Table 5.5: Parameterization of the experiments whose results are discussed in Section 5.5.3.

Prototype	Protocol	Parameter	Value
Our solution prototype	-	Resource distribution	lognormal
	Dissemination (Flood Service)	Message payload	100 bytes
	Membership protocol	Name Removal strategy	HyParView random
	ResEst membership protocol	Name	HyParView
	Proteus	Mode	average

Table 5.6: Average neighbor number error, for experiments with a Proteus rate of optimization of 1.

ResEst Max. Margin of Err. and Conf. Level	Average Neigh. Nr. Error
25% max. margin of err. w/ 90% conf. level	0.273
15% max. margin of err. w/ 95% conf. level	0.135

In Table 5.6 we report the average neighbor number error for two different experiments. These experiments (both using a Proteus rate of optimization of 1) share the same values in every parameter except the maximum margin of error and confidence level used in ResEst. Ergo, the one with 15% max. margin w/ 95% confidence level has a more restrictive stopping criterion than the other, resulting in a lower average neighbor number error. However, this lower error comes at a price—as one can see in Table 5.2, the number of hops for a maximum margin of error of 15% with a confidence level of 95% is 156, when compared to the 37 hops for a maximum margin of error of 25% with a confidence level of 90%. Nevertheless, due to the lightweight characteristic of ResEst (exposed in Section 3.3.4 of Chapter 3), and the high average neighbor number error difference detected in our experiments, we found it is worth it to use the most restrictive ResEst setting (a maximum margin of error of 15% with a confidence level of 95%) in all the remaining experiments whose results are discussed throughout the rest of this chapter.

Table 5.7: Average neighbor number error, for experiments with a ResEst maximum margin of error of 15% and confidence level of 95%.

Proteus Rate of Optimization	Average Neigh. Nr. Error
0.5	0.079
1.0	0.122
1.5	0.163

An interesting insight that we found when analyzing this metric in our tests was that there is a correlation between the Proteus rate of optimization used and average neighbor number error. As explained in Section 4.3 of Chapter 4, the Proteus rate of optimization

is the parameter that controls the slope of the Proteus optimization function (hence controling what the maximum number of neighbors of the most powerful node can be). In Table 5.7 it is displayed the average neighbor number error for three experiments. These experiments have exactly the same values for every parameter except the Proteus rate of optimization. As one can infer by the presented results, the lower the Proteus rate of optimization, the lower the average neighbor number error. This is the case because when the rate of optimization is lower, the number of number of neighbors that a node could have is lower than when the rate of optimization is higher—therefore lowering the chances of a node having the wrong number of neighbors. For example, when the Proteus rate of optimization is 0.5, the possible number of neighbors that the nodes can have varies between 4 and 10, but when the rate of optimization is 1.5, this number can vary between 4 and 17, making the chances of a node having a sub-optimal number of neighbors higher.

5.5.4 Reliability

As explained in Section 2.3.2.2 of Chapter 2, the reliability is a key performance metric used to evaluate message dissemination solutions. It can be defined as the percentage of nodes of the system that received a broadcasted message [29]. The average reliability is, therefore, the average of the reliabilities across all broadcasted messages. Since we used flood in our experiments, an average reliability of 100% is expected to validate our solution.

From all the performed experiments with different parameterizations, the only ones that did not achieve a 100% average reliability were the ones in which we used Cyclon [52] as the membership protocol. We think that this happens because Cyclon temporarily creates isolated groups of nodes, which makes it impossible for some of them to receive all broadcasted messages. Still, our solution managed to achieve average reliabilities of 98% in the experiments where Cyclon was used as the managed membership protocol.

Nevertheless, by achieving average reliabilities of 100% using HyParView, our solution proved to be valid using that membership protocol. Therefore, we opted to use HyParView as the membership protocol in all the remaining experiments whose results are discussed throughout the rest of this chapter.

On another note, as mentioned in Section 5.3.2.5, the message dissemination in our experiments was only started after ResEst and Proteus had already concluded their execution. However, this was not the case for the experiments where we modified the capacities of the nodes mid-execution. In the case of these experiments, we purposefully started the message dissemination just after Proteus and ResEst started running, in order to prove that our solution (Proteus + ResEst) can operate in a realistic scenario while messages are being flooded, while ensuring an average reliability of 100%. That was the case—Proteus and ResEst operating at the same time as messages were being flooded had no impact in reliability, even with some nodes changing their capacities mid-execution.

5.5.5 Latency

If reliability is the most relevant metric to prove the validity of our solution, latency is likely the most relevant metric when it comes to performance. As explained in Section 2.3.2.2 of Chapter 2, the latency of a message is given by the difference between the time upon which the last node receives the message and the time upon which the message was first broadcasted (by the original broadcaster). Therefore, the average latency of an experiment is given by the average of the latencies of all the broadcasted messages.

Besides average latency, another important metric that we also analyzed is the average *nth* percentile latency. The *nth* percentile latency (*nth* being 90th or 95th, for example) is given by the difference between the highest timestamp in which a message was received in the *nth* percentile of the nodes, and the time upon which the message was first broadcasted (by the original sender). For example, if a message was broadcasted at time tb and the message arrived to 95% of the nodes in time ta, then the 95th percentile latency for that message is $ta - tb$. The average *nth* percentile latency of an experiment is the average of the *nth* percentile latencies of all broadcasted messages in that experiment. Finally, another important metric is the average experienced latency per node. This metric represents, from the point of view of a node, the average of the latencies that it experienced in the messages that it received, i.e., the average of the differences between the time that it received a message and the time that that message was broadcasted.

We used the following parameterization in the experiments whose results are discussed in this section. For the base solution prototype, we used the lognormal resource distribution and HyParView as the membership protocol. For our solution prototype, we used the lognormal resource distribution, HyParView as the membership protocol, HyParView as the ResEst membership protocol, a ResEst maximum margin of error of 15% and confidence level of 95%, and the Proteus *average* mode. Table 5.8 summarizes this parameterization.

Table 5.8: Parameterization of the experiments whose results are discussed in Section 5.5.5.

Prototype	Protocol	Parameter	Value
Base solution prototype	-	Resource distribution	lognormal
	Membership protocol	Name	HyParView
Our solution prototype	-	Resource distribution	lognormal
	Membership protocol	Name	HyParView
	ResEst membership protocol	Name	HyParView
	ResEst	Max. margin of error Confidence level	15% 95%
	Proteus	Mode	average

In Table 5.9 we present the results of 6 different experiments on 4 different metrics. The metrics are the average latency, the average 90th percentile latency, the average 95th percentile latency, and the average 99th percentile latency. We present the results from 6 different experiments, in which 3 of them were performed with the baseline solution

Table 5.9: Comparing average latencies, in seconds, from our solution prototype (using a membership *random* removal strategy and a Proteus rate of optimization of 1.0) and the base solution (as benchmark).

	100 Bytes Payload			1 KB Payload			10 KB Payload		
	base solution	**our solution**	**delta**	**base solution**	**our solution**	**delta**	**base solution**	**our solution**	**delta**
avg. latency	0.630	0.520	-17%	0.718	0.675	-6%	1.804	1.432	-21%
avg. 90th per. latency	0.468	0.374	-20%	0.487	0.388	-20%	1.362	0.720	-47%
avg. 95th per. latency	0.498	0.401	-19%	0.516	0.429	-16.8	1.411	0.774	-45%
avg. 95th per. latency	0.565	0.445	-21%	0.553	0.524	-5%	1.547	0.914	-41%

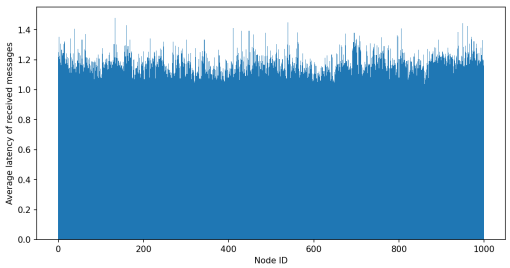
prototype, and the remaining 3 were performed with our solution prototype (i.e., the solution combining ResEst and Proteus), using a membership *random* removal strategy and a Proteus rate of optimization of 1.0. Therefore, we divided the experiments in three pairs, so that the only difference within each pair is that one of the experiments was performed with the base solution prototype and the other one was performed with our solution prototype—every other parameter in both experiments had the same value. Then, for each pair, we presented the difference between the latency of the benchmark and the latency of our prototype, in percentage, to which we called *delta*.

The only parameter that is different between the three pairs of experiments is the message payload. The first important insight to take is that, in all 6 experiments, our solution performed better than the benchmark on all metrics. When the payload being used is 100 B and 1 KB, our solution managed to achieve average latencies 17% and 6% lower than the baseline solution, respectively. The results are even more impressive when looking at the 90th percentile latency, where we achieved a latency 20% lower when using 100 B and 1 KB.

However, it is when the payload is 10 KB that our solution had the best results. With this payload, it presented a reduction of 21% in the average latency against the benchmark, and a reduction of more than 40% on the average 99th percentile latency. Our solution had such good results with this high payload size because the network was close to become saturated, as one can infer by the high latencies acquired when comparing with the experiences with lower payloads. In the baseline solution, since the less powerful nodes (with less bandwidth) could not timely send more messages, they ended up rising the latency for the other nodes, acting as bottlenecks. On the other hand, in our solution, since the less powerful nodes had a lower number of neighbors, the load imposed on them was lower, making them less saturated, while the powerful nodes (with more bandwidth) compensated with having many more neighbors, taking more of the load. This is what was responsible for such high reduction in latency.

Nevertheless, one can notice that in the 10 KB payload experiments there is a big difference in the delta of average latency and the delta of average 99th percentile latency, for example, with a delta of -20% in the former and double of that (-40%) in the latter. This can be explained by the fact that our solution has a very significant positive impact in the vast majority of the nodes (highly reducing their average experienced latencies), but only a moderate positive impact in a small portion of the total number of nodes. This

(a) Baseline solution.



(b) Our solution, using a membership *random* removal strategy and a Proteus rate of optimization of 1.0.

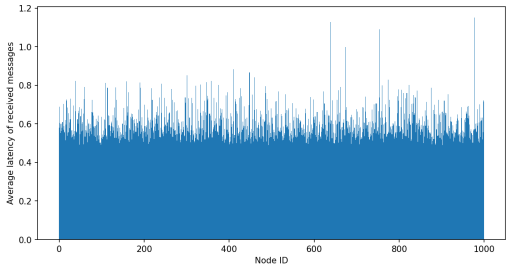


Figure 5.11: Histogram with the average experienced latency per node (in seconds), using 10 KB of payload.

can be easily seen in Figure 5.11(b), where we can observe a very low average experienced latencies for most of the nodes, but with some outliers having higher average experienced latencies. In comparison, in Figure 5.11(a), showing the results for the same metric but for the baseline solution, it can be noticed that the average experienced latencies are higher than in our solution, but also more uniform.

The reason for those outliers in Figure 5.11(b) is due to some weak nodes (i.e., with low bandwidth) having the misfortune of only being connected to other weak nodes, which produces localized delays for those particular nodes. However, this does not happen often, as can be observed in the figure. Nevertheless, even for these outliers, their average experienced latency was lower than the average experienced latencies in the baseline solution (Figure 5.11(a)), proving that our solution reduces, in fact, the average experienced latency of every node.

On another front, all the experiments reported up to now that use our prototype use a *random* node removal strategy. As explained in Section 5.3.2.4, this means that when the membership protocol has to remove some of the neighbors from its view upon receiving a notification from Proteus, it removes them according to the criterion *random*. However, we also evaluated two other node removal strategies—*by most powerful* and *by least powerful*.

In Table 5.10, we present the results for three new experiments, in order to compare the nodes' experienced average message latencies (in seconds) of the three experiments described in the previous paragraph. The first thing to notice is that using the strategy *by most powerful* resulted in much higher latencies than the other experiments. This is due to the fact that nodes remove neighbors by the criterion *most powerful*, which results in many weak nodes (i.e., with low bandwidth) ending up being connected solely to other weak nodes, creating many bottlenecks, resulting in high message latencies. One might think that, since that is the case, removing nodes by *least powerful* would provide us with the best results. However, the results show that the latencies experienced by nodes when that criterion is used are very similar to when the *random* criterion is used. This is likely

because the only requirement for a bottleneck not existing is the existence of at least one connection between a low bandwidth node and a high bandwidth node—and the *random* removal strategy already guarantees this most of the times. Therefore, we opted to use the *random* removal strategy in all the remaining experiments whose results are discussed throughout the rest of this chapter.

Table 5.10: Comparing nodes' experienced average message latencies (in seconds), using our solution (with a Proteus rate of optimization of 1.0 and a message payload of $10KB$), with different membership protocol removal strategies.

Node removal strategy:	**random**	**by most powerful**	**by least powerful**
90% of the nodes experience an avg. msg. latency below:	0.704	0.931	0.696
95% of the nodes experience an avg. msg. latency below:	0.742	1.117	0.739
99% of the nodes experience an avg. msg. latency below:	0.822	1.792	0.884

On a different note, in Table 5.11 we present the results from three other experiments, where the only different parameter value between them is the Proteus rate of optimization—1.0, 0.5, 1.5. The metrics reported in the table are the same as the ones reported in Table 5.9. As one can notice, the experiment with a rate of optimization of 1.0 presents the best results, with the clearly lower average latency, even though both other solutions presented similar results in the three remaining metrics. Therefore, we opted to use a Proteus rate of optimization of 1.0 in all the remaining experiments whose results are discussed throughout the rest of this chapter.

Table 5.11: Comparing average latencies (in seconds), using our solution (with a membership *random* removal strategy and a message payload of 10 KB), from experiments with different Proteus rates of optimization.

Proteus rate of optimization:	**1.0**	**0.5**	**1.5**
avg. latency	1.431	1.663	1.743
avg. 90th per. latency	0.720	0.710	0.720
avg. 95th per. latency	0.774	0.758	0.773
avg. 99th per. latency	0.914	0.883	0.918

5.5.6 Overhead from our solution

One important aspect to analyze is the overhead of our solution, comparing to the baseline solution. Since our solution includes three more protocols (Proteus, ResEst, and ResEst membership protocol) than the base solution, a node running our prototype will inevitably process a higher number of control messages. In this section we show the results of the measured overhead. To do so, we compare the average number of control bytes sent/received per node in our solution and in the benchmark. By number of control bytes sent/received per node, we mean the sum of the size of the non-flood messages that each node sent/received, i.e., every message related with the management of overlays and ResEst.

Table 5.12: Average number of control bytes sent and received per node with the base solution and our prototype, with a payload of 10 KB.

	baseline solution	**our solution**
avg. number of control bytes sent per node	74248	423672
avg. number of control bytes received per node	117516	10786079

Therefore, in Table 5.12, one can find the average number of control bytes sent per node in our solution and in a base solution. In order to get accurate results, note that both of these experiments share the same values in every parameter, with the only difference that one was performed using our solution and the other using the baseline solution. We used the following parameterization in each experiment. For the base solution prototype, we used the lognormal resource distribution, a dissemination message payload of 100 bytes, and HyParView as the membership protocol. For our solution prototype, we used the lognormal resource distribution, a dissemination message payload of 100 bytes, HyParView as the membership protocol with the *random* removal strategy, HyParView as the ResEst membership protocol, a ResEst maximum margin of error of 15% and confidence level of 95%, and a Proteus rate of optimization of 1.0 with the Proteus *average* mode. Table 5.13 summarizes this parameterization.

Table 5.13: Parameterization of the experiments whose results are discussed in Section 5.5.6.

Prototype	**Protocol**	**Parameter**	**Value**
Base solution prototype	-	Resource distribution	lognormal
	Dissemination (Flood Service)	Message payload	100 bytes
	Membership protocol	Name	HyParView
Our solution prototype	-	Resource distribution	lognormal
	Dissemination (Flood Service)	Message payload	100 bytes
	Membership protocol	Name Removal strategy	HyParView random
	ResEst membership protocol	Name	HyParView
	ResEst	Max. margin of error Confidence level	15% 95%
	Proteus	Rate of optimization Mode	1.0 average

As expected, our solution presented a much higher number of processed bytes per node, when compared to the baseline solution, especially in the number of received bytes. This is expected due to the use of the three extra protocols. Even though, when compared to the baseline solution, while the number of control bytes is significantly higher, in absolute, it is not that a high value, since, on average a node sends less than 1 MB of control data and receives less than 11 MB, during the whole experiment, which is about an hour. In other words, each node sends around 118 control bytes and receives around 2996 control bytes every second. Note that this is assuming that ResEst runs every minute, which is a high frequency, since in most realistic scenarios ResEst would only have to run once per hour or even once per day.

On the other hand, when doing the same experiment but with a ResEst maximum margin of error of 25% with confidence level of 90% (instead of 15% and 95%, respectively), the overhead was heavily reduced, with each node sending on average 77 control bytes per second and receiving 285 control bytes per second. This is due to the fact that, with this parameterization, ResEst sampling visits a lower number of nodes (as previously reported in Section 3.3.3 of Chapter 3), resulting in less communications.

5.6 Solution with the best parameterization

Having evaluated our solution and analyzed a series of metrics, in this section we are going to deduce the best combination of parameters that provide us with the best performance. Note that, however, there is no one-size-fits-all solution, since different parameters offer us different trade-offs. For example, as we discerned in Section 5.5.3, using ResEst maximum margin of error of 15% with confidence level of 95% is what guarantees the lower average neighbor number error, however, as seen in Section 5.5.6 that is also the parameterization that causes the highest overhead (when compared with using a maximum margin of error of 25% with confidence level of 90%).

Nevertheless, when it comes to the ResEst membership protocol to use, both Cyclon and HyParView presented similar results (as can be in Section 5.4). Therefore, we conclude that the best option would be to use HyParView, due to the use of symmetric views. As per the maximum margin of error and confidence level used by ResEst, the best results were achieved with 15% maximum margin of error and 95% confidence level, due to the lower average neighbor number error that it presented in Section 5.5.3. The membership protocol that provided us with the best results was HyParView, since we could not achieve 100% reliability with Cyclon, for the reasons pointed out and discussed in Section 5.5.4. The membership protocol removal strategy with the best performance was *random* (with *by least powerful* sharing similar results), since it was the parameter that delivered the lower average latency, as explained in Section 5.5.5. When it comes to Proteus, a rate of optimization of 1.0 provided the best results, with the lowest average latency of the 3 different rates of optimization evaluated (in Section 5.5.5). Finally, as seen in Section 5.5.2, the Proteus mode with the best results proved to be the *average* mode, demonstrating a lower neighbor variability over time than the *simple* mode.

5.7 Summary

In this chapter, we started by describing the architecture of the prototypes used to evaluate our solution; then, we explained their software implementation, describing the framework used and the software network constraints added; afterwards, before delving into the actual evaluation, we enumerated and described the experimental settings used, both in hardware and in the software parameterization.

We evaluated ResEst in emulation with one thousand nodes, in order to access the applicability of it in realistic scenarios. ResEst proved to be valid, making good estimations of the resource distribution of the network while being lightweight. For example, with HyParView, ResEst could generate a good estimation of the network distribution (with a histogram error lower than 10%) in less than 40 communication hops (with a maximum margin of error of 25% and a confidence level of 90%).

Moreover, we then evaluated our full solution prototype (i.e., with ResEst and Proteus) and empirically proved that our solution is valid, maintaining a reliability of 100%. Besides, we also demonstrated that our solution achieved a much better performance than the benchmark—reducing the average latency up to 40% with certain parameterizations.

Having evaluated our solution, in the next chapter we are going to conclude the thesis. More specifically, we are going to revisit the problem that we tackled, the contributions that resulted from our work and their results, and introduce the future work.

6

Conclusion

In the previous chapters we have presented the challenge that this thesis wanted to address, examined the related work, proposed our solution, and finally, evaluated it. In this chapter, we are going to conclude this thesis. Therefore, this chapter is divided into two sections: in Section 6.1, we summarize the work performed on this thesis and its applicability; and in Section 6.2, we close-off with enumerating possible optimizations that can be done to our solution in future work.

6.1 Conclusion

Edge computing is being widely used nowadays, addressing some of the problems of cloud computing by bringing the computation towards the edge of the system, closer to the client. There is a panoply of devices that can be integrated into future edge computing platforms, from local data-centers to IoT devices. For these devices to communicate fruitfully, we need to build systems that enable the seamless cooperation between them. However, this is not trivial, due to the different capacities of the existing devices—some very powerful (like local data centers), and some with lower capacities (like IoT devices). In addition, most of these devices should be managed in a decentralized way, with robust overlays that can manage all these different machines. Unstructured overlays provide exactly this—a decentralized and robust way to manage a very high number of devices. However, most of the work that has been done in this area assumes that the network is homogeneous—which is not the case in general. Therefore, in this thesis, we tackled this problem by devising a solution that adapts the number of overlay neighbors of each device according to how powerful that device is in relation to the rest of the network.

Nonetheless, before devising our solution, we examined different other algorithms and protocols that aimed to tackle this heterogeneity problem in unstructured overlay networks. However, all of them had critical drawbacks—either because they would tackle the heterogeneity problem on the application layer (instead of the membership layer), or because they would generate less robust structured overlays, for example. None of them adapted the number of neighbors of each device according to its capacity in the network,

which we thought to be the best way to tackle heterogeneity in the edge.

Therefore, we implemented a solution that tackles the heterogeneity problem on the edge by doing this—adapting the number of neighbors of each node according to its capacity in relation to the other nodes in the network. There would be, in practice, two ways of doing this: i) we could build a novel overlay that would have all these tools built-in; or ii) we could create a set of tools that could be integrated in most peer-to-peer distributed systems, working with whatever unstructured overlay those systems already run. We opted for the latter option, since it can be integrated into already existing systems—or even future ones, with very slight adaptations to the membership protocol being used in those systems. Our devised solution can be divided in two: ResEst and Proteus.

ResEst is a distributed algorithm that aims at inferring the distribution of capacities of all the nodes in the network. This is done through the distributed computation of a histogram that approximates the distribution of capacities/resources of all existing nodes. The algorithm is parameterizable, allowing us to adjust the intended confidence in the obtained estimation, in order to allow for some control between the consumed resources by the algorithm (and its running time) and the quality of the obtained estimate. ResEst can be used in stand-alone by any distributed application that wants an estimation of the network resource distribution. We evaluated ResEst in simulation (with up to one million nodes), proving its validity and demonstrating its performance in very large-scale systems.

The second part of our solution, Proteus, receives an estimation of the resource distribution of the network from ResEst and uses that information to compute the number of neighbors that that node should have, taking into consideration its capacity and the estimation of the capacity distribution of the network. Upon computing this number, Proteus sends a notification to the membership protocol being used with the requested number of neighbors. Then, the membership protocol removes or adds neighbors, depending on whether it has more or less neighbors than it should have. The only adaptation needed to be done in a membership protocol of a system where our solution is integrated is just this listener that adds or removes neighbors depending on the notification received, making our solution easily integrable in most peer-to-peer distributed systems.

After presenting our solution, we then evaluated it in emulation with realistic prototype deployments. We started by evaluating ResEst stand-alone, to take into account the membership protocol being used with it and the existing network constraints. ResEst proved to be valid in all performed experiments, returning estimations with a lower error than the parameterized maximum error at all times.

After proving the validity of ResEst in realistic scenarios, we evaluated our complete solution—ResEst with Proteus. We performed a number of experiments where we acquired different metrics that we then analyzed. To do so, we performed these experiments on two distributed system prototypes that disseminate messages (through flooding)—one with our solution (i.e., ResEst and Proteus) integrated, and another one without it (to serve

as a baseline). The prototype with our solution integrated proved to be valid, achieving reliabilities of 100% when the benchmark also did. Moreover, it also proved to have a much better performance, reducing latencies up to 40% in some cases, compared to the benchmark.

In summary, we have developed a solution to address the problem of heterogeneity in decentralized systems. Such solution can benefit future edge computing platforms. This solution adapts the number of neighbors of each device (i.e., how many other devices that device knows) according to the capacity of that device and the capacities of the other devices in the network, in order to ensure that the load is fairly distributed between them and, as a consequence, reduce the latencies experienced by them. This solution can be easily integrated into most existing peer-to-peer distributed systems, requiring just a slight adaptation to their membership (i.e., unstructured overlay network) protocol.

6.2 Future Work

In this last section we detail possible future work to integrate in our solution. More specifically, there are two possible optimizations that we believe could have a positive impact in the performance of our solution. Those are:

Automatically finding the best Proteus rate of optimization One of the biggest difficulties of our parameterization was to know what Proteus rates of optimization to evaluate. We decided to perform experiments with 3 different rates of optimization, however, we are not sure about what the optimal one would be—if it was evaluated or not. Therefore, it would be interesting to integrate a tool in Proteus that—using the estimation distribution of resources sent by ResEst—would infer the best value to use as Proteus rate of optimization.

Reducing the overhead of our solution using only one membership protocol Most of the overhead introduced by our solution comes from ResEst random-walks. However, some of that overhead is also caused by the fact that we introduce a new membership protocol to the system—the ResEst membership protocol, increasing the number of control messages. Regardless, as previously mentioned, we can not use only one overlay in which ResEst performs its estimations and the number of neighbors of each node is adapted according to those exact estimations, since we would bias the ResEst estimations themselves. Yet, maybe we could take advantage of the fact that HyParView uses two different overlays and use only one membership protocol (HyParView), in which the active view is used by the application and the passive view is used by ResEst. Nevertheless, we are not sure if such optimization would be as trivial as it sounds, since the quality of the ResEst estimations would depend on how independent the HyParView active view is from its passive view. In either case, it is an optimization that is worth exploring in the future.

Bibliography

[1] D. Angluin et al. "Computation in networks of passively mobile finite-state sensors". In: *Distributed computing* 18.4 (2006) (cit. on p. 30).

[2] C. Baquero et al. "Extrema Propagation: Fast Distributed Estimation of Sums and Network Sizes". In: *IEEE Transactions on Parallel and Distributed Systems* 23.4 (2012) (cit. on p. 30).

[3] M. Bawa et al. *Estimating Aggregates on a Peer-to-Peer Network*. Technical Report 2003-24. Stanford InfoLab, Apr. 2003. URL: http://ilpubs.stanford.edu:8090/586/ (cit. on p. 31).

[4] J. Benet. *IPFS - Content Addressed, Versioned, P2P File System*. https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4k7zrJa3LX/ipfs.draft3.pdf. 2014. arXiv: 1407.3561 [cs.NI] (cit. on p. 2).

[5] B. Beverly Yang and H. Garcia-Molina. "Designing a super-peer network". In: *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*. 2003, pp. 49–60. DOI: 10.1109/ICDE.2003.1260781 (cit. on pp. 28, 29, 41).

[6] N. Carvalho et al. "Emergent Structure in Unstructured Epidemic Multicast". In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. 2007, pp. 481–490. DOI: 10.1109/DSN.2007.40 (cit. on pp. 9, 11, 25–27).

[7] P. A. Costa, P. Fouto, and J. Leitao. "Overlay Networks for Edge Management". In: *2020 IEEE 19th International Symposium on Network Computing and Applications (NCA)*. 2020, pp. 1–10. DOI: 10.1109/NCA51143.2020.9306716 (cit. on pp. 7, 24–26, 29, 34, 38, 57).

[8] C. Diot et al. "Deployment issues for the IP multicast service and architecture". In: *IEEE Network* 14.1 (2000), pp. 78–88. DOI: 10.1109/65.819174 (cit. on p. 11).

[9] P. T. Eugster et al. "Epidemic information dissemination in distributed systems". In: *Computer* 37.5 (2004), pp. 60–67. DOI: 10.1109/MC.2004.1297243 (cit. on pp. 9, 12, 20, 59).

[10] A. Fernandez et al. "Distributed Slicing in Dynamic Systems". In: *27th International Conference on Distributed Computing Systems (ICDCS '07)*. 2007 (cit. on p. 30).

[11] M. J. Freedman et al. "Geographic locality of IP prefixes". In: *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*. 2005, pp. 13–13 (cit. on p. 30).

[12] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié. "Scamp: Peer-to-Peer Lightweight Membership Service for Large-Scale Group Communication". In: *Networked Group Communication*. Ed. by J. Crowcroft and M. Hofmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 44–55. ISBN: 978-3-540-45546-2 (cit. on pp. 15, 21, 22).

[13] I. Gupta et al. "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 (cit. on p. 51).

[14] R. Hasan et al. "A survey of peer-to-peer storage techniques for distributed file systems". In: *International Conference on Information Technology: Coding and Computing (ITCC'05)-Volume II*. Vol. 2. IEEE. 2005, pp. 205–213 (cit. on pp. 8, 9).

[15] R. M. (IBM). *How cloud, fog, and mist computing can work together*. https://developer.ibm.com/technologies/iot/articles/how-cloud-fog-and-mist-computing-can-work-together. Mar. 2018 (cit. on p. 6).

[16] M. Jelasity and O. Babaoglu. "T-Man: Gossip-Based Overlay Topology Management". In: *Engineering Self-Organising Systems*. Ed. by S. A. Brueckner et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–15. ISBN: 978-3-540-33352-4 (cit. on pp. 23, 24, 30).

[17] M. Jelasity and A.-M. Kermarrec. "Ordered slicing of very large-scale overlay networks". In: *Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06)*. IEEE. 2006 (cit. on p. 30).

[18] M. Jelasity et al. "Gossip-based peer sampling". In: *ACM Transactions on Computer Systems (TOCS)* 25.3 (2007) (cit. on p. 37).

[19] M. F. Kaashoek, A. S. Tanenbaum, and S. F. Hummel. "An Efficient Reliable Broadcast Protocol". In: *SIGOPS Oper. Syst. Rev.* 23.4 (Oct. 1989), 5–19. ISSN: 0163-5980. DOI: 10.1145/70730.70732. URL: https://doi.org/10.1145/70730.70732 (cit. on p. 11).

[20] A. Kermarrec, L. Massoulie, and A. J. Ganesh. "Probabilistic reliable dissemination in large-scale systems". In: *IEEE Transactions on Parallel and Distributed Systems* 14.3 (2003), pp. 248–258. DOI: 10.1109/TPDS.2003.1189583 (cit. on pp. 12, 13, 34).

[21] P. Labs. *Filecoin: A Decentralized Storage Network*. https://filecoin.io/filecoin.pdf. 2017 (cit. on pp. 2, 9).

[22] P. Labs. *libp2p*. https://libp2p.io (cit. on p. 2).

[23] P. Labs. *Protocol Labs Website*. https://protocol.ai/about. Feb. 2021 (cit. on p. 2).

[24] J. Leitão. "Gossip-Based Broadcast Protocols". MA thesis. Faculdade de Ciências da Universidade de Lisboa, 2007 (cit. on p. 34).

[25] J. Leitao, J. Pereira, and L. Rodrigues. "Epidemic Broadcast Trees". In: *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. 2007, pp. 301–310. DOI: 10.1109/SRDS.2007.27 (cit. on pp. 10, 13, 14).

[26] J. Leitao, J. Pereira, and L. Rodrigues. "HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast". In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. 2007, pp. 419–429. DOI: 10.1109/DSN.2007.56 (cit. on pp. 1, 12, 15–17, 22, 23, 30, 34, 35, 48, 54, 59).

[27] J. Leitão et al. "X-BOT: A Protocol for Resilient Optimization of Unstructured Overlay Networks". In: *IEEE Transactions on Parallel and Distributed Systems* 99.PrePrints (2012). ISSN: 1045-9219. DOI: http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.29 (cit. on pp. 23, 24, 30).

[28] J. Leitão. *PhD thesis, Topology Management for Unstructured Overlay Networks*. Tech. rep. Technical University of Lisbon, Sept. 2009 (cit. on pp. 5, 7–9, 11, 12, 14, 15, 17, 18, 20, 23).

[29] J. Leitão, J. Pereira, and L. Rodrigues. "Gossip-Based Broadcast". In: Oct. 2010, pp. 831–860. DOI: 10.1007/978-0-387-09751-0_29 (cit. on pp. 8–10, 12–17, 71).

[30] J. Leitão et al. *Towards Enabling Novel Edge-Enabled Applications*. 2019. arXiv: 1805.06989 [cs.DC] (cit. on pp. 1, 6).

[31] J. Liang and K. Nahrstedt. *RandPeer: Membership management for QoS sensitive peer to peer applications*. Tech. rep. 2005 (cit. on p. 15).

[32] Q. Lv et al. "Search and Replication in Unstructured Peer-to-Peer Networks". In: *Proceedings of the 16th International Conference on Supercomputing*. ICS '02. New York, NY, USA: Association for Computing Machinery, 2002, 84–95. ISBN: 1581134835. DOI: 10.1145/514191.514206. URL: https://doi.org/10.1145/514191.514206 (cit. on p. 20).

[33] R. Mahmud, R. Kotagiri, and R. Buyya. "Fog Computing: A Taxonomy, Survey and Future Directions". In: *Internet of Everything: Algorithms, Methodologies, Technologies and Perspectives*. Ed. by B. Di Martino et al. Singapore: Springer Singapore, 2018, pp. 103–130. ISBN: 978-981-10-5861-5. DOI: 10.1007/978-981-10-5861-5_5. URL: https://doi.org/10.1007/978-981-10-5861-5_5 (cit. on p. 6).

[34] L. Massoulié et al. "Peer Counting and Sampling in Overlay Networks: Random Walk Methods". In: *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing*. Denver, Colorado, USA, 2006. ISBN: 1595933840 (cit. on pp. 31, 34).

[35] P. Maymounkov and D. Mazières. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". In: *Peer-to-Peer Systems*. Ed. by P. Druschel, F. Kaashoek, and A. Rowstron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 53–65. ISBN: 978-3-540-45748-0 (cit. on pp. 18, 19, 51).

[36] R. Melamed and I. Keidar. "Araneola: a scalable reliable multicast system for dynamic environments". In: *Third IEEE International Symposium on Network Computing and Applications, 2004. (NCA 2004). Proceedings.* 2004, pp. 5–14. DOI: 10.1109/NCA.2004.1347755 (cit. on p. 25).

[37] V. H. Menino, P. A. Costa, and J. Leitao. "ResEst — Algoritmo Distribuido para a Inferencia de Recursos da Rede". In: Lisbon, Portugal, 2021 (cit. on p. 3).

[38] S. Nakamoto. *Bitcoin whitepaper*. Tech. rep. 2008. URL: https://bitcoin.org/bitcoin.pdf (cit. on p. 6).

[39] J. Pereira et al. "Low latency probabilistic broadcast in wide area networks". In: *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.* 2004, pp. 299–308. DOI: 10.1109/RELDIS.2004.1353030 (cit. on pp. 27, 28).

[40] J. Pereira, R. Oliveira, and L. Rodrigues. "Efficient Epidemic Multicast in Heterogeneous Networks". In: *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*. Ed. by R. Meersman, Z. Tari, and P. Herrero. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1520–1529. ISBN: 978-3-540-48276-5 (cit. on pp. 9, 11, 26).

[41] K. Petersen et al. "Bayou: Replicated Database Services for World-Wide Applications". In: *Proceedings of the 7th Workshop on ACM SIGOPS European Workshop: Systems Support for Worldwide Applications*. EW 7. New York, NY, USA: Association for Computing Machinery, 1996, 275–280. ISBN: 9781450373395. DOI: 10.1145/504450.504497. URL: https://doi.org/10.1145/504450.504497 (cit. on pp. 11, 12).

[42] B Pourebrahimi, K Bertels, and S Vassiliadis. "A survey of peer-to-peer networks". In: *Proceedings of the 16th annual workshop on Circuits, Systems and Signal Processing*. Citeseer. 2005 (cit. on p. 6).

[43] J. Risson and T. Moors. "Survey of research towards robust peer-to-peer networks: Search methods". In: *Computer Networks* 50.17 (2006), pp. 3485–3521. ISSN: 13891286. DOI: 10.1016/j.comnet.2006.02.001 (cit. on pp. 8, 9).

[44] R. Rodrigues and P. Druschel. "Peer-to-peer systems". In: *Communications of the ACM* 53.10 (2010), pp. 72–82 (cit. on p. 6).

[45] W. Shi et al. "Edge Computing: Vision and Challenges". In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646. DOI: 10.1109/JIOT.2016.2579198 (cit. on pp. 1, 6).

[46] N. Shrivastava et al. "Medians and beyond: New Aggregation Techniques for Sensor Networks". In: *Proc. of SenSys '04*. Baltimore, MD, USA, 2004. ISBN: 1581138792 (cit. on p. 31).

[47] M. Sood and O. Yağan. "On the Minimum Node Degree and k-connectivity in Inhomogeneous Random K-out Graphs". In: *IEEE Transactions on Information Theory* (2021) (cit. on pp. 44, 59).

[48] I. Stoica et al. "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications". In: *SIGCOMM Comput. Commun. Rev.* 31.4 (Aug. 2001), 149–160. ISSN: 0146-4833. DOI: 10.1145/964723.383071. URL: https://doi.org/10.1145/964723.383071 (cit. on pp. 17, 18).

[49] Y. Sudo et al. "Self-Stabilizing Population Protocols With Global Knowledge". In: *IEEE Transactions on Parallel and Distributed Systems* 32.12 (2021) (cit. on pp. 30, 34).

[50] C. Tang, R. N. Chang, and C. Ward. "GoCast: gossip-enhanced overlay multicast for fast and dependable group communication". In: *2005 International Conference on Dependable Systems and Networks (DSN'05)*. 2005, pp. 140–149. DOI: 10.1109/DSN.2005.52 (cit. on p. 25).

[51] J. H. M. Valerie J. Easton. *Statistics Glossary v1.1*. http://www.stats.gla.ac.uk/steps/glossary/. Sept. 1997 (cit. on p. 35).

[52] S. Voulgaris, D. Gavidia, and M. van Steen. "CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays". In: *Journal of Network and Systems Management* 13.2 (June 2005), pp. 197–217. ISSN: 1573-7705. DOI: 10.1007/s10922-005-4441-x. URL: https://doi.org/10.1007/s10922-005-4441-x (cit. on pp. 1, 20, 21, 30, 59, 71).

[53] S. Wilkinson et al. "Storj a peer-to-peer cloud storage network". In: (2014) (cit. on p. 9).

[54] D. Xu et al. "On peer-to-peer media streaming". In: *Proceedings 22nd International Conference on Distributed Computing Systems*. IEEE. 2002, pp. 363–371 (cit. on pp. 8, 9).

[55] B Yang and H. Garcia-Molina. "Comparing Hybrid Peer-to-Peer Systems". In: *27th International Conference on Very Large Data Bases (VLDB 2001)*. This is a shortened version; see the extended version for full details. Sept. 2001. URL: http://ilpubs.stanford.edu:8090/727/ (cit. on p. 7).