NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

RICARDO MIGUEL GONÇALVES LEITÃO

BSc in Computer Science

MANAGING POPULATION AND WORKLOAD IMBALANCE IN STRUCTURED OVERLAYS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2021

DEPARTMENT OF
COMPUTER SCIENCE

MANAGING POPULATION AND WORKLOAD IMBALANCE IN STRUCTURED OVERLAYS

RICARDO MIGUEL GONÇALVES LEITÃO

BSc in Computer Science

Adviser: João A. Silva
Researcher, NOVA Laboratory for Computer Science and Informatics

Co-advisers: João M. Lourenço
Associate Professor, NOVA University Lisbon

Hervé Paulino
Associate Professor, NOVA University Lisbon

Examination Committee

Chair: Jorg Matthias Knorr
Assistant Professor, NOVA University Lisbon

Rapporteur: António Gelásio Frazão Isidro Teófilo
Adjunct Professor, Instituto Superior de Engenharia de Lisboa

Member: João A. Silva
Researcher, NOVA Laboratory for Computer Science and Informatics

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2021

Managing Population and Workload Imbalance in Structured Overlays

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.9.1) [25].

To my friends and family.

Acknowledgements

My most profound gratitude goes to my advisers, Doctor João Silva and Profs. João Lourenço and Hervé Paulino, who guided me throughout this journey.

I'm very grateful to FCT-UNL and the Department of Informatics for being a second home to me and helping me grow not only as a student but as a person.

The completion of my dissertation would not have been possible without the support and nurturing of my family, who have been supporting me since the first day, and never doubt of my capabilities even when i did.

I would like to thank my friends, namely, André Augusto, my longest friend, we have been together through so many adventures, and I am sure many more will follow. Pedro Fernandes, a caring friend that I had the privileged to work with. "Mengas" Rodrigues and Duarte Mesquita, although our paths are more distant now, I leave you with my wishes of success in your adventures and I am sure that our paths will cross again. Tiago Gonçalves, my caring mate, that although separate at first, we have reached our goals together. Henrique Fernandes, Henrique "OwO", Tomás Alagoa and Pedro Mota, never since I have started this adventure, there was a week where you did not support me. Eduardo Súbtil, we have started this adventure together and finally we will end it together. We all have been together for what seems a life time, and we always have each other backs.

Finally, this text would be complete without thanking my cat Kitsu, which spent every night sleeping in my lap, ensuring that I would not leave my chair.

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT-MCTES) in the scope of the DeDuCe (PTDC/CCI-COM/32166/2017) research project.

“Nature does not hurry, yet everything is accomplished.”
(Lao Tzu)

Abstract

Every day the number of data produced by networked devices increases. The current paradigm is to offload the data produced to data centers to be processed. However as more and more devices are offloading their data do cloud centers, accessing data becomes increasingly more challenging. To combat this problem, systems are bringing data closer to the consumer and distributing network responsibilities among the end devices. We are witnessing a change in networking paradigm, where data storage and computation that was once only handled in the cloud, is being processed by Internet of Things (IoT) and mobile devices, thanks to the ever increasing technological capabilities of these devices. One approach, leverages devices into a structured overlay network.

Structured Overlays are a common approach to address the organization and distribution of data in peer-to-peer distributed systems. Due to their nature, indexing and searching for elements of the system becomes trivial, thus structured overlays become ideal building blocks of resource location based applications.

Such overlays assume that the data is distributed evenly over the peers, and that the popularity of those data items is also evenly balanced. However in many systems, due to many factors outside of the system domain, popularity may behave rather randomly, allowing for some nodes to spare more resources looking for the popular items than others.

In this work we intend to exploit the properties of cluster-based structured overlays propose to address this problem by improving a structure overlay with the mechanisms to manage the population and workload imbalance and achieve more uniform use of resources.

Our approach focus on implementing a Group-Based Distributed Hash Table (DHT) capable of dynamically changing its groups to accommodate the changes in churn in the network.

With the conclusion of our work we believe that we have indeed created a network capable of withstanding high levels of churn, while ensuring fairness to all members of the network.

Keywords: Structured Overlays; Distributed hash Tables.

Resumo

Todos os dias aumenta o número de dados produzidos por dispositivos em rede. O paradigma atual é descarregar os dados produzidos para centros de dados para serem processados. No entanto com o aumento do número de dispositivos a descarregar dados para estes centros, o acesso aos dados torna-se cada vez mais desafiante. Para combater este problema, os sistemas estão a aproximar os dados dos consumidores e a distribuir responsabilidades de rede entre os dispositivos. Estamos a assistir a uma mudança no paradigma de redes, onde o armazenamento de dados e a computação que antes eram da responsabilidade dos centros de dados, está a ser processado por dispositivos móveis IoT, graças às crescentes capacidades tecnológicas destes dispositivos. Uma abordagem, junta os dispositivos em redes estruturadas.

As redes estruturadas são o meio mais comum de organizar e distribuir dados em redes peer-to-peer. Gradas às suas propriedades, indexar e procurar por elementos torna-se trivial, assim, as redes estruturadas tornam-se o bloco de construção ideal para sistemas de procura de ficheiros.

Estas redes assumem que os dados estão distribuídos equitativamente por todos os participantes e que todos esses dados são igualmente procurados. no entanto em muitos sistemas, por factores externos a popularidade tem um comportamento volátil e imprevisível sobrecarregando os participantes que guardam os dados mais populares.

Este trabalho tenta explorar as propriedades das redes estruturadas em grupo para confrontar o problema, vamos equipar uma destas redes com os mecanismos necessários para coordenar os participantes e a sua carga.

A nossa abordagem focasse na implementação de uma DHT baseado em grupos capaz de alterar dinamicamente os grupos para acomodar as mudanças de membros da rede.

Com a conclusão de nosso trabalho, acreditamos que criamos uma rede capaz de suportar altos níveis de instabilidade, enquanto garante justiça a todos os membros da rede.

Palavras-chave: Redes Estruturadas; Tabelas de dispersão Distribuidas.

Contents

List of Figures

List of Tables

Acronyms

DHT	Distributed Hash Table vi, vii, 3–5, 9, 11, 12, 15, 16, 22, 23, 39, 40, 50
GHT	Geographic Hash Table 17, 18
IoT	Internet of Things vi, vii, 1, 3
KBR	Key Based Routing 9
P/S	Publisher Subscriber 4, 5, 16, 17, 19, 21, 38
P2P	Peer-to-peer 7
PNS	Proximity neighbor selection 11
UUID	Universally Unique Identifier 27–32, 34, 36

1

Introduction

1.1 Context and Motivation

We are witnessing a revolution in the field of communication devices. Well established in our day-to-day life is the paradigm of *cloud-computing* [18]. *Cloud computing* was built with a well-defined hierarchical structure. At the center of the *cloud*, we can find large data centers with unparalleled storage and processing power. Each data center is fed by millions of different networks. Finally, connected to these networks are billions of devices, with the most numerous being Internet of Things (IoT) and sensing devices, followed by smart mobile phones. This layered architecture connects these networks and devices seemingly throughout the globe.

As seen in Figure 1.1, the larger and most powerful components are located in the highest layers, closer to the cloud. However, due to their cost of production and maintenance, these components are also the least numerous. As we get closer to the edge, the computational power of individual devices decreases, while the number of such devices increases many-folds.

Devices on the edge have the least amount of computational power, and sometimes need to offload their storage and computational needs to the data centers in the cloud. Thus, the *cloud computing* plays multiple simultaneous roles, from hosting simple storage and computation services to hosting full applications delivered as a service. The constant enhancements in the cloud-related technologies has increased the speed and reliability of the services it hosts, while the prices have dropped considerably.

Simultaneously with the ever-increasing technological capabilities of the *cloud*, mobile devices are experiencing an even greater evolution, obtaining an unprecedented place in today's society. This technological growth enabled for applications and services targeting these devices on the edge to become more data-intensive, challenging the scalability and usability of cloud servers, translating in network latency. Furthermore, as these devices have additional battery restrains, these type of applications will reduce the recharging cycle.

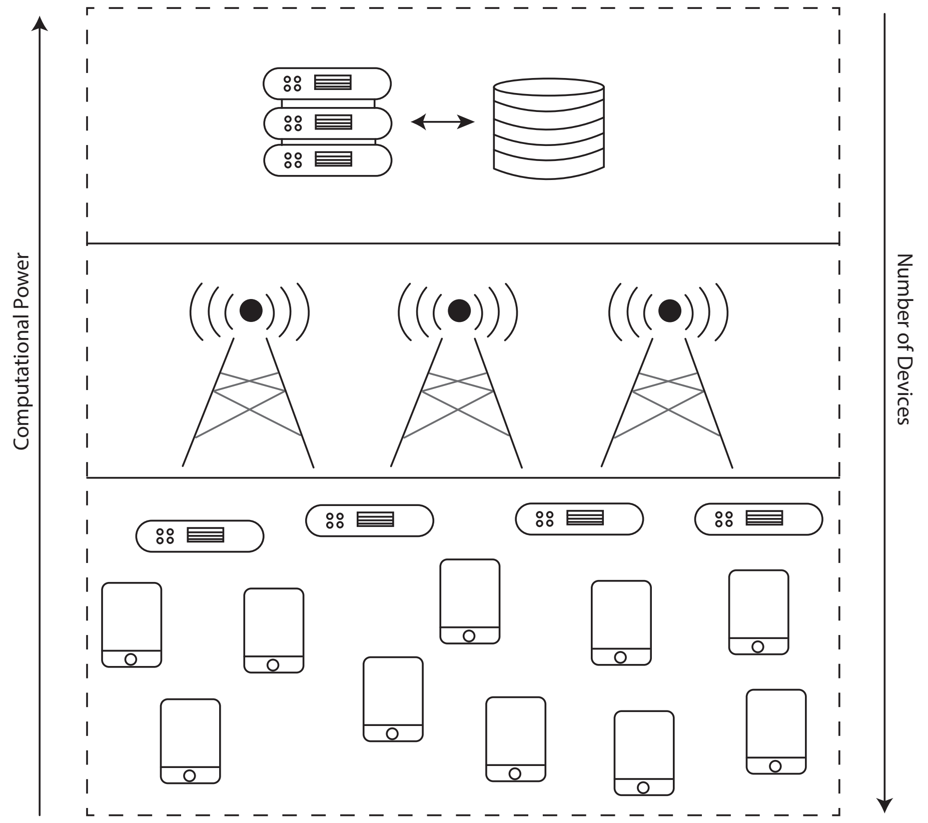

Figure 1.1: Exemplification of the structure of a network. I) On the top layer we have large data centers interfacing with databases, capable of processing huge amounts of data. II) At a medium level we have smaller server and networks connecting to the cloud. III) At the edge level, billions of mobile devices and edge servers produce large amounts of data offloading them to the cloud.

A recent study from CISCO [9] predicts that by 2023 there will be 29.3 billion networked devices, and the overall traffic generated will surpass the value of 77.5 exabytes per month. And that this trend is poised to keep rising every year. The study also confirms that mobile clients offload the generated traffic to the closest infrastructure station, such has Wi-Fi hot-spots. Consequently, the same study from CISCO predict that the number of public Wi-Fi hot-spots will grow four-fold, reaching nearly 628 millions by 2023.

On the time of *cloud computing* introduction, devices at the edge offered little computational resources, assigning the majority of the computational load to the *cloud* was an efficient way to offer scalability to networks. However, although smartphones and Wi-Fi hot-spots hardware capabilities have greatly increased, with such edge devices generating more data, mobile communications still remains a bottleneck due to the lack of location awareness when connecting to a data center [12]. Additionally, due to the very nature of mobile devices, better support for mobility has been required.

To overcome the limited computing and storage resources in mobile devices, mobile applications often make use of a computing infrastructure closer to the user's Wi-Fi access point. These infrastructures, often called cloudlets, are far more numerous than cloud data centers but, unlike data centers, they can only be accessed by Wi-Fi rather than a wired connection, creating a smaller coverage area. Additionally, by design, cloudlets are less resourceful than cloud centers, making them harder to scale [46].

Thanks to the growth of the technological capability of mobile devices, we start to notice a shift in the responsibility placed in these devices. In an attempt to bring an answer for the problems described above, new networks try to leverage the storage and computational capability of numerous mobile or IoT devices in order to create a new layer below the *cloud*. This new computational paradigm is called *edge computing*. With the employment of *edge computing*, the cloud hierarchy is reformed by pushing computational resources in the proximity of end users [44].

Although far less resourceful, the *edge computing* paradigm leverages the number of resources in the edge, allowing for devices to share their storage and computing capabilities. Thus, *edge computing* is a decentralized computing paradigm where participants share their computing resources and other computational services. *Edge computing* answers the computational needs of devices at the edge when low-computational and context-aware services are required. Furthermore, it allows wireless subscribers to access the closest computing servers within the range of wireless Radio Access Network [26].

The CISCO study [9] also refers that the majority of the data produced will be consumed by the end users (in the edge), and thus storing the data closer to the possible consuming devices will increase its availability.

To leverage the complexity of mobile applications and the limited resources available in mobile devices is a challenge that many *edge computing* solutions face. To address this problem, works like [13, 37] introduce an additional component to the edge, called *edge server*. The *edge server* takes its place between the mobile devices and the cloudlets or data centers, allowing for data to be stored as close as possible to the consumers. Allowing for the least amount of delay in reaching data.

In the works like Krowd [13], *edge servers* takes upon the role of coordinator and manager of the network. The *edge servers* can have a complete membership of the network, and act as a one-hop interface with the mobiles networks. However, building a network of mobile devices that will offload their computational needs to the nearest *edge server* may introduce another wave of problems, more precise scalability problems [1].

To combat this situation, authors on works like [49, 48, 51] choose to organize the networks at the edge into a Distributed Hash Table (DHT), organizing the network into a distributed key-value store. Organizing the network into a DHT is a scalable alternative to the central *edge server* solution. Additionally, by splitting the storage over a number of participants, these solutions eliminate the problem of single point of failure that networks based only on *edge servers* face.

Finally, as a way to increase resilience within the DHT, works like KAD [41] propose to cluster devices together. These clusters become a mean of replication within the network, increasing data availability and increasing the chances of survival in situations when members leave the network.

1.2 Problem and Challenges

To increase the resilience of a structured overlay and at the same time increasing data availability, DHT opt to join the mobile devices in virtual groups. The same principle was applied by the authors of *Thyme* [8, 39, 38].

Thyme is a Publisher Subscriber (P/S) system, that distinguish itself from others on the same category by supporting subscriptions for any period of time, being past, present or future. The system offers a novel approach to P/S systems targeting mobile devices, where participants in the network take the roles of clients (data consumers), publishers (data producers) and data brokers (data managers).

Thyme offers a group-based DHT to deliver messages inside the network and although the authors, thoroughly present a scalable and resilient solution, the current iteration of Thyme offers only a static number of groups. This number is configured before the system boot. This limitation will be the focus of our solution.

With a static number of groups, solutions built atop Thyme are faced with scalability problems, as they are limited to the number of groups configured. Additionally, no selection is made when assigning members to a group, originating groups with uneven participants.

Thus, to attack these issues we propose a further augmentation to the Thyme DHT. In order to reduce latency, the previous iteration of Thyme, sorted the participants by their geographic position. However, in the context of this work, we will be working under the assumption that all devices will be connected to an edge server as seen in [36]. This will allow us to be confident that latency between devices will roughly be the same, as all devices will have a one-hop reach to the edge server. Furthermore, all devices will be contained in a geographic area fairly small as they are limited by the edger server range. Thus, we are able to introduce fewer restrictions when forming groups. With this work our objective is to eliminate the need of configuring the limits of groups in the network, thus we propose a DHT that can dynamically adjust its group size according to the participants. We will introduce mechanisms for groups to be created when the device count gets high, and ways to join groups together when devices leave.

Finally, the main challenges lie on the following topics:

1. How to update the information inside the group in order for all members to be up-to-date as much as possible?

2. How to create and delete groups during the execution of the system?

3. How to ensure that the implemented operations will execute with the least overhead as possible?

1.3 Proposed Solution

In this work, we propose to implement a group-based DHT, where data replication will be done at group level. Devices will be added to a group at the moment they join the network. To address the unbalance problem, our DHT is able to create new groups as the devices join. Additionally, as the devices leave the network, our DHT is able to merge groups with low member count to preserve data integrity. Finally, devices will be able to switch groups to better suit the network needs.

We will implement this DHT into the mentioned Thyme system, a P/S system that already makes use of a group-based DHT to route messages between the devices.

Finally, our DHT will be inspired by the proposition made in [40], a dynamic group-based DHT capable of answering load unbalance during the system execution, designed to introduce low overhead during the balancing operations.

1.4 Contributions

With the development of this work we created and tested an implementation of a dynamically adaptable DHT, proposed by João Silva in is work [40].

Thus, our contributions ultimately are:

- An implementation of a Group-Based DHT designed to reduce the negative impacts of churn and network unbalance.
- The integration of this DHT within Thyme.
- Test results and analysis obtained from the current Thyme with our implementation.

1.5 Document Outline

This document is structured as follows. In Chapter 2 is presented in a bottom up perspective, work related to the described problem and their proposed solutions. Following in Chapter 3 we present *Thyme* and *GardenBed* [36] the systems we used to implement our solution. We will present a brief overview of their characteristics, highlighting the problems that our solution covers. Then in Chapter 4 we describe in detail our proposed solution, highlighting any interesting particularly. Consequently, we present the evaluation of performance and behavior of said implementation in Chapter 5. Finally, in Chapter 6, we conclude our work with our final remarks and discussion of open challenges for future work.

2

State of the Art

The purpose of this chapter is to present related work, concepts and implementations of multiple technologies and systems of network organization. We will present alternative paths taken by many systems and the challenges they bring.

2.1 Distributed Architectures

The development of computer networking opened the doors to information sharing.

Machines are connected through a physical layer, and use software to be organized and behave cooperatively forming a network, allowing machines to communicate even trough large distances, and thus distributed systems were created.

In a distributed system independent machines are connected by a network and do not physically share hardware. These machines cooperate on some common task, in such way that from the outside of the scope of the distributed system it appears to be a single machine working. We will be referring to all machines participating in a system as nodes and to the set of nodes that a node communicate with as its neighborhood. Naturally distributed systems behave in an asynchronous way, since there is no global clock to synchronize all the nodes nor a global notion of the correct time, nodes achieve synchronization by exchanging messages. Nodes behave in an independent way, meaning that concurrency is usually present trough the system, because of this independent behavior, a node fault is usually an independent occurrence, but as nodes are connected to each others a node failure may render some nodes inaccessible and isolated, as it may have been the connection point to the other nodes, this phenomenon is know as network partition. From these properties derives an impossibility to detect when a node truly crashes, since messages can be delayed indefinitely while trying to access a shared resource or because the node is temporary isolated. During these situations the other nodes can no longer distinguish between a delayed message or an nonexistent response from a node that is no longer working like intended [15].

To circumvent this situation, assumptions have to be made, and restrictions have to be imposed. For example, when designing the system we can impose an upper time restriction (or timeout) in the delivery of messages, when it's reached we can consider that the node has crashed. Since their introductions, distributed systems have evolved, and now they are the center of huge global applications. As of this thesis we will show case two main types of distributed systems: one where the participants assume distinct functions and importance, where all minor participants rely on a master server and the second one where usually all participants share the same responsibility in the system. This thesis will give a greater importance to the former architecture, as it is a building block to our algorithm.

2.1.1 Client-Server Architectures

In a client-server network, the participants assume two distinct functions: the client, responsible for smaller and lighter operations usually backed by the server and the server, designed to operate as a centralized point of access for all clients.

The client-server architecture offers better performance in smaller networks where the server has more resources than its clients counter part, but the issues start to be evident when trying to scale to larger networks, where the increasing number of clients will consequentially increasing the demand for the server resources. Since usually the only way for the servers to support the growing number of clients is to grow vertically (by increasing the quality of the hardware), maintenance and upgrades will become more costly. This architecture presents a centralized point of failure, as the network and its services are dependent on the server.

2.1.2 Peer-to-Peer Architectures

Peer-to-peer (P2P) networking is a distributed system architecture, where the workload is partitioned between peers by sharing their own hardware resources [35]. Commonly in this type of architecture peers are equally privileged, in principle no peer shares more hardware with the network than any other peer, and organize themselves in a structure akin to a graph, where nodes are the vertices and the link between nodes the edges. In P2P architectures, peers do not have a predefined behavior and can leave, join or fail at any time a phenomenon known as churn [43]. Whenever a peer joins or leaves the system, the other peers must calculate its impact on the graph, therefore the bigger the network is, the bigger is the probability for nodes to leave of join, amplifying the effects of churn and hindering the capability of the system to scale. This phenomenon brings the challenge for P2P architectures to be constructed in way to survive higher rates of churn. P2P allow for a decentralization of huge monolithic systems [28], granting a better distribution of the workload necessary to store, distribute and compute data, therefor allowing a better performance under failure since there is no longer a single point of failure. The first generation of P2P system were created to solve the problems of file sharing and storage applications among a large group of independent users (many separated by large geographic distances), an example of such algorithms is Napster [17] and Gnutella [2].

Napster used central directory servers, peers that had increased resources, these peers were used to keep track of the files stored in the system. The behavior of Napster super-peers was, very close of one of a centralized server, thus they inherited the same scalability problems. As Napster became more popular, super-peers had increasing difficulty in keeping up with he demand to serve a process requests, from the users. Furthermore the negatives impact, in the occurrence of a super-peer failure may impact the network greatly, as regular peers must find a new super-peer [24].

Unlike Napster, Gnutella does not enforce any special responsibility to any particular peer, atoning for some scalability issues. Furthermore Gnutella does not organize peers in a particular way in the network graph, allowing for peers to make random links with other peers when joining the network, creating a network organized in what is known as an *unstructured overlay* [30].

As links between peers are randomly generated in *unstructured overlays*, it is not always possible for a peer to know an exact location of another. Furthermore it would not be feasible, for all peers to know the location of every other peer, as this method would be exponentially demanding in larger networks. However it is common for peers in *unstructured overlays* like [2], to store a fraction of the network state, usually the peer stores the peers that maintain links with, this set of peers is commonly known as a peer neighborhood, furthermore as peers only have knowledge of their immediate neighbours the effects of churn are minimized.

To route messages peers resort to flood the network, duplicating and delivering the same message to all peers in its neighbourhood, this process is then repeated by all peers, until the target peer is reached [10]. From the previous property we can deduce, *unstructured overlays* behave optimally when the objective is to deliver messages to more than one target peer. However in more particular cases like resource location, *unstructured overlays* behave sub-optimally, as the resource can be anywhere in the network, the only way to locate it is to flood all peers, demanding resources from all peers in the network even when not needed.

To improve the behaviour of networks in cases where knowing the location of a peer is useful, algorithms like Chord [42] impose logical constraints of how neighbourhoods can be formed and how messages should be forwarded. Neighbourhood constraints allow for a deterministic evolution of the network and organizing the network graph, thus this overlays are known as *structured overlays*.

2.2 Structured Overlays

In *structured overlays*, peers organize themselves and the network in a predefined graph, due to the diversity of devices that participate in this type of networks (computers, mobile devices, IoT devices...), in this work we will treat all peers participating in a network as *nodes*. Algorithms that operate under a *structured overlay*, impose constraints on how nodes should connect to each other, furthermore these constraints will also influence how routing is made between nodes [29].

Structured overlays are excellent building blocks, of application where it is required for nodes to find resources in the network. As the same algorithms applied to build the overlay, can be applied to the routing and mapping of messages and data objects, allowing for deterministic look-ups in the network, this family of algorithms, behave like a traditional *Hash Tables* distributed between the network, thus earning the name DHT.

2.3 Distributed Hash Tables

DHT are a type of *structured overlays*, where the concept of network is divided among all participating nodes. As mentioned previously, in a DHT, nodes present a behaviour akin to traditional data structures, distributed trough the network, in this case *Hash Tables*.

In essence DHT offer a single operation, look-up, given a characteristic, DHTs find the closest node with said characteristic, the first DHTs assigned to nodes and messages an unique identifier and routing was made by nodes that shared some similarities with the *ID*. Some notable DHT algorithms that introduced Key Based Routing (KBR) include Chord, Pastry, CAN and Tapestry [42, 33, 31, 50]. Furthermore it is worth noting that all of the previous algorithms differ between them, as different network organization will impact how nodes behave, we will try to highlight the difference in algorithms and how they influence the behaviour of the algorithms.

2.3.1 Name Space

DHT are based on the concept of dividing the virtual space of the network, by using deterministic algorithms in *unique identifier* assignment to nodes and messages, name spaces allow for data objects in the system to be mapped into the participant nodes. Thus an ID can be seen as a coordinate for a subspace inside the name space. By principal nodes are responsible for the subspace that matches their ID, data objects like files are mapped into a subspace to be handled by the corresponding node.

One of the objective of using *structured overlays* and DHT, is to divide the name space equally by all nodes, as all nodes will be responsible for the same amount of virtual space, all nodes in theory should have the same workload. It is however worth noting some works like [29], where by purposely allowing uneven load distribution to some nodes, the overall network achieves greater reliability and less bandwidth consumption.

2.3.1.1 Name Space Topology

The topology of a network dictates how nodes will connect to each other and how message routing will be done, usually algorithms like [42, 33, 31, 50, 27, 3], try to balance the load of each region withing the space by assigning sub-spaces of the same size to every node. By assigning nodes sub-spaces of the same size, the probability of a new item to be assigned to any sub-space will be the same, thus, uniformly spreading them.

The topology shape has influence in a node neighbourhood, and in the number of messages needed to form the topology, different algorithms try to optimize different metrics whiting the network.

Tree shaped Topology In tree based topology, the prefix of the IDs are used to maintain order between nodes, nodes that share similar prefixes are closer to each other. When trying to forward messages, the node on the neighbourhood that shares most similarities with the ID prefix of the message is selected.

DHTs like Tapestry [50] and Kad [41], are implemented using a tree-like mesh of connections, granting it more resilience to node failures at the cost of more routing information being forwarded and stored in each node, to shorten the number of hops a message has to do until it reaches its target, Kad nodes store many neighbours in cache, allowing for more options when choosing the next node to send the message. While some algorithms like Kademlia [27] organize the network into a binary tree, relying less on path redundancy to maintain availability, but commenting less resources to store and calculate neighbourhood paths, improving the network resilience to *churn*.

Ring shaped Topology Algorithms like Chord [42] and Pastry [33] opt to organize the nodes into a ring like structure, where ID are organized by ascending order, and the node with the biggest id is connected to the node with the smallest, completing the ring. A problem may be immediate following this structure, in the extreme case forward a message in a ring with n nodes, may take $O(n)$ hops to reach the target. As seen before, many algorithms chose, to have non neighbouring nodes in their cache to ease the process of routing. Chord nodes store other nodes exponentially away, thus achieving an average routing of $O(log_n)$ number of hops. MobiStore [21], values the number of hops in the network, using a combination of a larger cache, and a probabilistic routing algorithm, messages are delivered in $O(1)$ hops, only in rare cases falling back to $O(log_n)$

N-Dimensional Topology So far we have presented algorithms, that treat the virtual space has a two dimensional plane, however due to the virtual connections between nodes, a network topology can scale to any dimension. Works like CAN [31], choose to represent the network space as a torus with n dimensions. As the geometric shape can be further subdivided into single points, CAN appoints this coordinates has nodes IDs, then each node is responsible for the messages which ID matches an area around the node coordinate, messages are then routed to the neighbour that coordinates have the smallest distance to the target coordinate. Following a similar principle [3], organizes the nodes as vertices of an hypercube, recurring to the properties of the hypercube [34] and by allowing for nodes to utilize more memory to cache all the neighbours paths, PeerCube [3] manages to create a resilient network built on redundant and independent paths.

2.3.2 ID Selection impact On the Network

As we described so far, the majority of the algorithms presented, route message to the neighbour, that ID presents the most similarities with the message ID. All of these algorithms, calculate and assign the IDs in a similar way, since assigning an ID to an object in the system, is to map it to a position within the name space. To maintain a behaviour similar to traditional *Hash Tables*, it is required for IDs to be unique, thus the majority of the algorithms assign IDs trough the form of *consistent hashing* [19].

Consistent Hashing, generates IDs trough the use of Hash Functions applied to a property. Hash functions properties [11] align with the desired properties of an unique ID generation method, as hash functions present a deterministic output and an output range large enough to minimize collisions.

The goal of using *consistent hashing* is to minimize the number of neighbours a node has to store, allowing it to store only those who are closer to its ID, messages IDs are compared to those of the neighbours and the forwarded to the closest mach, minimizing the number of hops a message needs to do to reach a target and eliminating the need of broadcasting look up messages. Additionally minimizing the number of neighbours in cache, allows to also minimize the number of updates nodes have to make to their cache, when a neighbours leaves or joins the network.

The work presented in [19], described consistent hashing as a way to distribute requests among a changing population of web servers. DHT benefit greatly from this property, since by design they strive to be resilient to changes on the topology. However due to the nature of the hashing algorithms consistent hashing doesn't preserve order as it randomly assigns IDs to nodes, making it incompatible with partial keys queries, wildcards or ranges of values.

Due to nature of the most common used algorithms to generate IDs, nodes with close IDs may not necessarily be geographically closer to ach other, as consistent hashing tries to minimize the number of hops a message has to travel in the network . It is important to consider that the latency between nodes can have the same impact in the time to deliver a message as the number of hops between the nodes communicating. Proximity neighbor selection (PNS) tries to form neighborhoods between nodes using the latency as key of sorting the proximity between them, resulting in exchange of minor latency at the cost of allocating more resources, as latency between nodes have to monitored. The work [6], presents a modified version of Pastry, taking advantage of Pastry existing neighbour cache, and configuring them to pick the closest node in the underlying network from among those whose IDs have the required prefix.

Table 2.1: Comparison of how the network topology impacts messages routing.

Algorithm	**ID Assignment Method**	**Network Topology**	**Routing Criteria**	**Routing Complexity**
PeerCube	Consistent Hash	HyperCube	Prefix Matching	$O(log_n)$
RollerChain	Consistent Hash	Double Link Ring	Prefix Matching	$O(log_n)$
Chord	Consistent Hash	Single Link Ring	Prefix Matching	$O(log_n)$
Kad	Consistent Hash	Unbalanced Tree	Prefix Matching	$O(\sqrt[2]{[n]})$
Scatter	On join to balance groups	Ring	Prefix Matching	$O(n/2)$
MobiStore	Consistent Hash	Ring-like Mesh	Prefix Matching	$O(1) to O(log_n)$
CAN	Cartesian Coordinates	d-Dimensional Torus	Cartesian Distance	$O(\sqrt[2]{[n]})$

2.4 Challenges in DHTs

DHTs operate under the assumption that data objects will be distributed uniformly among all nodes [29], however in a remote file storage system scenario, due to various factors data items will have different popularity, meaning that some nodes responsible for storing and managing the data will receive a heavier load of look up requests for items than others, causing load imbalance in the system. Furthermore in the same context of a file storage system, some nodes may be more actively posting than others, this may overload the neighbouring nodes, as now they have to forward all the messages being inserted into the network.

Load imbalance is caused when the workload across the network is not distributed efficiently, many nodes in the path of the target node will receive a heavier overhead than others. In each request, the node needs to compute the path to forward the message and the target node has to process that request. This leads to an imbalance that can compromise the availability of certain nodes or even overload them entirely and removing them from the system due to a crash, nodes that receive higher workload than others are known to be in an HotSpot.

In the network, nodes store information about the routing of messages and data from applications running on them. When a node leaves the network, either freely or by crashing, the data stored in the node will be lost, to avoid this loss of information and to avoid a single node overload by being flooded with requests, data must be replicated between any number of nodes, thus splitting the load between the network, it is important to understand what to replicate and to where, as replicating data across the network will result in an increase demand for storage and additional request may be needed to locate the new data position.

In the literature, it has been study two approaches to reduce the overloading of nodes. By replicating data trough the network, its availability is increased, furthermore the nodes with the replicated data, are now able to answer for queries of the data, releasing some work load form the original target node. The activation mechanism for this method is usually defined *à priori*, in works like DataCube [32], where replication is the chosen method for maintain availability, data objects are replicated to all the nodes neighbour upon the data insertion in the network, after which all become available to answer queries for the data item.

The second approaches involves monitoring the network, and allow for nodes to make decisions based on the state they perceive, Scatter [16] follows a ring like topology, however nodes are capable of detecting an existence of HotSpots and may potentially share or shift the responsibility to more nodes, allowing for fairer utilization of the network, similar techniques where a decision is made accordingly to the state of the network will be referred as *adaptive load balancing*.

2.4.1 Replication to Minimize Unbalances

Previously we presented two situations that originated load unbalance within the network: one a node or a small set of nodes, being targeted with unexpected amounts of requests and when a node is responsible for a large number of data objects and leaves the system.

A solution often employed, is to replicate the data trough the system, replicated data is will not leave the system when the original owner does and nodes with a copy of the data can answer queries for it.

To define what should be stored and replicated by the peers is important in order to maximize the lookup-hit ratio to a point where it is not detrimental to the overall performance of the network. Thus a trade-off between storage and availability is usually considered when choosing what type of data will be replicated.

Fully replicating the object will maximize its availability as more nodes can leave the network before the object is lost, however fully replicating data objects will demand an increase in storage, furthermore an increase overhead will be placed on the bandwidth as replicating larger objects will require for more messages to be delivered, The work of [5] analysed the cost of replication and concluded that a significant part of the bandwidth consumption is due to replication mechanism.

To minimize the additional storage required by the replication mechanism, some approaches focus on replicating only some key elements related to the data, *metadata*. Metadata can be used locate the original object or a group of objects holding the same properties has the ones present in the metadata. Metadata tends to have a much smaller size than the associated object, allowing for smaller additional storage, as metadata is smaller than data replicating metadata will transmit in smaller messages to circulate in the network, reducing the overhead caused by delivering messages. However each node has to implement another control mechanism to keep in cache the original location of the metadata that is stored. Furthermore has metadata are properties of the original object, it can be used to make partial queries, even when using *consistent hashing*, a problem that was analysed in 2.3.2. Not all data may be worth to replicate, and a middle ground between replicating data or metadata can be found using a *popularity-based* approach. The work of [14] analyses mobile users access patterns to create a file popularity prediction, defining access frequency, access spread and access consistency as good metric to measure the object popularity. Additionally [4] noted that all popular objects will have approximated access rates and the less popular objects will follow a *Zipf*-like distribution.

2.4.2 Replication techniques in the Literature

Replication can be implemented in many strategies as depending on the network, as replicating data will demand an increase in resources by the system, different forms of replication will offer different trade-offs and compromises. As data objects can be fully replicated to guarantee availability at the cost of more memory, replicating metadata guarantees availability on if the original data does not leave the network.

In Scatter [16], a system where nodes are grouped together in groups, the authors opted to employ *group replication*. When a node inserts an object into the network, a copy is delivered to all of neighbours, the neighbours keep a full copy of said object, as all members of a group can answer queries for any object in the group, the original owner can leave the network without presenting a data loss in the network. A similar principle is applied in PeerCube and RollerChain.

A less memory demanding method is *neighbour replication*, where upon insertion data is replicated only to the nodes closest neighbours, a similar but more proactive approach is *path replication*, where when a query is made, the result is then cached in the nodes in the path until the target is met.

As seen before in 2.3.2 *consistent hashing*, due to the nature of hash functions used to generate the IDs, it is possible to add some random piece of information about the data to the key that will generate the node ID, this method is known as *salted key replication*, thus generating a completely different key, that according to the size of the network it has a great probability to map to a different node in the system, this way, multiple deterministic keys can be generated, splitting the lookups for the data around the network. Although this solution may use more bandwidth than neighbourhood replication since the target of the replications may be more spread throughout the network, and requests have to be sent to more than one node, it requires less caching investment since the key holds information's about the node and not their neighbours.

2.4.3 Adaptive Load Balancing

Many algorithms use consistent hashing as it is expected from the algorithm to produce consistent and almost random outputs, and it is expected that in a network with N nodes and M data items each node will have around M/N items, however as shown in [47] this is not true in a practical example as the variance of the output of the hash functions was high and the load was not balanced between nodes. To deal with this situation many DHTs implement a second method of ID selecting, from those we present the following.

Virtual Servers Clustering Until now we characterized nodes as physical machines in the network, however many algorithms rely on virtual nodes, virtual nodes are logical instance of the physical node, and can share physical resources between them, although in the network they will be treated as a normal physical node. This brings many advantages as the data stored can be accessed in parallel without traveling in the network to be replicated, virtual nodes can maintain different relations than their physical counterparts, allowing for a dynamic relationship in clustering. In Chord [42], the problem of load imbalance is solved by using $log(N)$ virtual nodes per physical node. The idea is to offset the unbalance caused by consistent hashing method, [42] states that without using virtual nodes the load in each node tens to be $(1+e)|K|/|N|$ where N is the number of nodes and K the number of keys each node is responsible for, it also states that when using virtual nodes, e can be reduced to a negligible value.

Dynamic Intervals This method consists in dividing the DHTs address space into intervals. All intervals hold a configurable amount of nodes f, as nodes join the network, they are mapped into these intervals, when the number of nodes in a single interval reaches a number of $2f$, the interval is split into two halves. Any data item stored in a single interval is stored on all nodes in this specific interval. As data items are stored on all nodes in a single interval. The loss of a single node would not lead to the loss of all data items stored on that node. The split of any interval is done by splitting the interval in its center and thus results in no movement of any node. If an interval holds between f + 1 and 2f nodes, a node can dynamically move into a different interval when its own capacity reaches a defined threshold. The works of [20, 22] actively probe the system and assign IDs to new nodes so they can be placed in cluster where load unbalance is detected.

3 Thyme GardenBed

In this chapter we present the foundation of our work. In Section 3.1 we present Thyme, a topic-based publish/subscribe system, that already takes advantage of a DHT to deliver messages to its users. In Section 3.2 we introduce GardenBed, a distributed system composed by stationary nodes located at the edge of the network. Finally in Section 3.3 we describe Thyme GardenBed, an approach resulting from the combination of the previous two systems.

3.1 Thyme

Thyme [8, 39, 38] merges two paradigms together, being a P/S system as well as a distributed and persistent data storage system specially designed to suit the needs of mobile edge networks. Thyme distinguishes itself from similar systems by being the first to support time-awareness during its subscriptions.

Thyme leverages the most relevant features from the P/S paradigm and storage. In Thyme, in order for the storage layer to be able to notify the user when new data is available, it synergies with the P/S layer. The P/S layer feeds the storage with the data received, and on the other way, the storage layer is able to provide persistent publications by using the storage layer. Together the system is able to manage subscriptions and published data providing time-awareness for their duration.

To manage publications and subscriptions within the system while ensuring a low computational environment, Thyme indexes data using a topic-based strategy. When a data object is published a set of tags also needs to be provided, these tags double as descriptors of the object and as topics for the P/S layer. Unlike typical Topic-Based P/S system, the subscribe operation was modified in order to allow the user to specify more than a single topic.

In Thyme, devices (hereafter named nodes) are considered to be functionally symmetric, sharing the same responsibilities and having no particular roles. This means that there are no centralized or specialized components, and each node can be a publisher, a subscriber, or both.

To deliver messages within the network, Thyme makes use of a Geographic Hash Table (GHT), leveraging the nodes geographic position in order to reduce latency between nodes close together. Additional the geographic space is divided into multiples groups called cells, all with equal sizes. Nodes inside the same cell collaborate together in order to ensure the convergence of all nodes to the same state.

Nodes joining the system wait a configurable amount of time for a beacon sent from a neighbouring cell. Upon receiving such beacon, the nodes uses the sender as a entry point into the network. After a suitable entry point is found, the node sends a join request, if successful the node will be updated with the cell state. However if no beacon is received, the node assumes to be alone in the network and starts operating normally.

3.1.1 Publishing Data

Upon the publish of a data object a new metadata item related to the object is also created. Every metadata object created is a tuple of data $\langle id_{\text{obj}}, T, s, ts^{\text{pub}}, id_{\text{owner}} \rangle$ where:

- id_{obj} is the object identifier;
- T the set of tags related to the object;
- s a summarized representation of the shared object, e.g., a thumbnail of an image in the case of a photo sharing network;
- ts^{pub} the object's publication timestamp to guarantee temporal perception, used to bootstrap the time-aware functionality of the developed solution;
- id_{owner} the publisher's node identifier.

In Thyme the publish operation in the P/S layer and the insert operation in the storage layer are merged. Thus every publication in the network is at the same time an insertion onto storage. Upon a publication Thyme leverages the indexation mechanism of cells in the GHT to produce a similar address for each tag of the object inserted. Then the metadata object is sent to each cell that matches the tag identifier.

To save bandwidth only metadata is sent to the cell, as it tends to be much smaller than the corresponding data object. The data object itself is only sent to the nodes present in same cell as the publisher.

3.1.2 Replication

Thyme was designed for dynamic and ever changing mobile networks, where the high levels of churn can compromise data accessibility or permanence in the system. To face this situation, Thyme offers two types of data replication:

Active Replication. Takes advantages of the cells present in the GHT. Upon a publish operation, the publisher node disseminates a copy of the published data object to all nodes within the same cell. Ensuring that every node in the cell is able to reply for requests to the object. Overall increasing the network tolerance to churn. Furthermore it allows for data to stay in the network even when the original publisher leaves;

Passive Replication. When ever nodes retrieve data objects published in another cell, they store it within their storage. Passive replication spreads copies of data across multiple cells, increasing its availability. Additionally by allowing for nodes to search the same data across multiple cells may lead to lower levels of latency, as a future requester may be closer to one of the replicas relatively to the original publisher.

To support both strategies of replication, the location of the replicas need to be widespread trough the network. Thus whenever a replica is created, the identifier of the replica owner and its cell are added to the object metadata, in a field called *replicationlist*. To avoid creating a large metadata object, a limited number of only the most recent replicas are stored in this list.

3.1.3 Subscription

When nodes wants to subscribe to a topic, the requesting node sends a message to the network. This message is composed by $\langle id_{\text{sub}}, q, ts^{\text{s}}, ts^{\text{e}}, id_{\text{owner}}, cell_{\text{owner}} \rangle$, where:

- id_{sub} is the subscription identifier;
- q represents the query's logic formula which is allowed to contain keyword conjunctions and disjunctions;
- ts^{s} is the initial timestamp of published data to be retrieved, 0 if the users want to get all data from the beginning of the system related to the specified query;
- ts^{e} contains the timestamp's upper of published data to be retrieved, ∞ if the user wishes to receive all future data matching the query;
- id_{owner} represents the requesting node identifier;
- $cell_{\text{owner}}$ holds the identifier of the logical cell where the requesting node is located.

When a subscription request is issued, the Thyme client of the issuer parses the query section of the message. Each of the clause is then subjected to the same id generation process of the GHT, in order to discover which cells are holding the matching object metadata. After this process is complete a subscription request message is then sent to the selected cells.

Upon receiving a subscription request, the receiver node stores the identifier of the sender in their list of active subscription requests, thus after this moment the receiving node will be responsible to notifying the requesting nodes whenever a new data item is published that matches the specified parameters.

From now on, whenever a notification messages containing the object's metadata is triggered, the node will transmit a data retrieval message to the closest replica in the list of replicas. Finally upon receiving the data requested, the receiving node will store it in its storage, being able to contribute the to the passive replication for that particular object.

3.2 GardenBed

GardenBed [36, 45], is a framework that leverages the capabilities of edge server infrastructure, built upon the presented Thyme system. GardenBed allows for mobile nodes to create, retrieve and delete any object inside the network without the need of an active cloud service. GardenBed offers stationary nodes located near the mobile clients.

Multiple edge server using GardenBed are able to be fully or partially connected trough a mesh-topology, allowing to support the mobile network with additional storage, processing and routing. Consequently, mobile nodes are able to retrieve data that was generated in others regions.

To be able to support P/S systems, GardenBed servers offer an interface for mobile devices to communicate with. Among the operations supported we can find typical P/S operations such as publish, unpublish, subscribe, unsubscribe and download.

GardenBed was also designed to support virtual groups or clusters. As clusters are virtual groups of devices, to avoid introducing unnecessary communication overhead GardenBed selects one member of each group to communicate, naming it the cluster-head node.

Client will keep in memory operations performed, and the cluster-head node will regularly collect and batch-disseminate this list to the GardenBed servers. On the other hand, the server will receive and notify the clients whenever new data is available. Ultimately the server is able to:

- Notify the clients when a new publication from another geographic location is made;
- Update object's metadata, allowing for clients and servers to share the same metadata of a given object;
- Cache data published within the region in order to make it available to other regions;
- Delete data that has been unpublished, ensuring that the object is removed entirely from the system.

Periodically the edge servers compute a list of subscriptions within the region called *subscription catalog*. This catalog is disseminated to other servers. As servers exchange information about subscription, they are also able to notify each other when data produced within the region matches one of the subscription received, thus GardenBed ensures that every subscriber is able to be notified about new data even when published in another data.

When ever a download operation is issued, the clients sends a message to the server, specifying as well if the request is local or remote. For a local request, the server forwards the request to the correct group if the data object was not cached within the server. In case of a remote download, the server will first look within its cache, if contains the object, the object is sent to the client. Otherwise the server will forward the request to the correct region. The receiving server will look for the that within its cache, if it is not found, it forwards the request to the request to the mobile replicas within the region. When a server receives the reply to a remote download with data from another region, before routing the requested item to the end-user, it will proactively cache the data.

As seen by the above mechanism, published data may cross and be replication outside of the published region. Thus whenever a unpublish operation is issued, it has be executed within every replica. Whenever a server receives an unpublish operation it deletes the object associated in all of its caches. The servers outside of the unpublisher region also forward the message to the cluster-head of the group responsible for handling the item's metadata, for it to delete any replicas of the data.

3.3 Thyme GardenBed

Thyme GardenBed [45] combines the device to device communication capabilities of Thyme with the edge server suport of GardenBed.

As both Thyme and GardenBed were designed with virtual groups in mind, a relation of one-to-one can be made of Thyme's cells and GardenBed's clusters. Leveraging this relation, a cluster-head node is elected per cell.

The cluster-head node, becomes the communication point of the cell. All messages originated from the GardenBed server targeting the cell are delivered to the cluster-head, only then forwarded to the remaining members. As this strategy focus on the over utilization of a single node resources, the election of a cluster-head is made within each round of updates.

Cluster-heads are elected trough metrics that predict how willingly is the node to leave the network, called stability index. This index is calculated using information from device such has the battery capacity or the GPS location. The higher the stability index of a node, the lower is the probability of it leaving the system, either by moving too far from the server or just shutting down.

Thyme GardenBed extends the Thyme's metadata structure, previous described in Section 3.1.1. The new metadata structure has an additional field called *onTheEdge*, that indicates if the object is replicated on the edge server.

By combining the inter-region capabilities of GardenBed, the Thyme system P/S layer is augmented, allowing it to fetch and retrieve data published outside of the region.

When a subscribe request is issued to a data object outside of the subscriber region the object is cached on the server, ensuring its availability for request within the server's cell. Furthermore the object is also indexed within the Thyme clients.

By design the system will prioritize the usage of the edge servers. Whenever a subscription is made to an object whose metadata is flagged onTheEdge, the responsible cell will refrain from sending the data, as it is cached on the server, the server will reply in its place.

Finally, the replica selection policy follows the same logic as the notification priority policy, which is to favor the edge servers in order to preserve the battery capacity of mobile nodes.

4

Implementation

In this chapter we will present, in detail, our implementation of a self-balancing DHT. We will begin in Section 4.1 by presenting an overview of the proposed solution, describing our assumptions of the problem and clarifying the notation used through this chapter. Then, in Section 4.4.1 we will review the most important components previously implemented, highlighting the ones that were modified during the time of our work. The implemented protocols will be described in Sections 4.2 and 4.3, being these sections presented in a bottom-up perspective. Finally, in Section 4.5 we will give our closing considerations.

4.1 General Overview

During the time of our work we have developed a resilient, self-balancing DHT. Taking particularly use of the advantages provided by a *Group-Based Distributed Hash Table*, our DHT, clusters devices into groups, known as *cells*. Each *cell* is a group of devices that during this dissertation we will be referring as *nodes*. Furthermore when referring to nodes in the same cell as the one in question we will use the term *peers*.

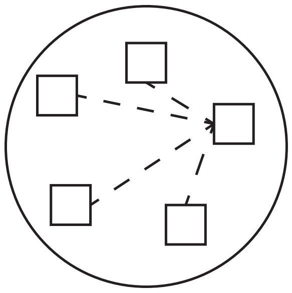

Figure 4.1: Representation a cell, nodes within have a connection to every other node inside the cell.

Nodes in the network are represented trough the cell they are inserted in, making the cell the basic unit of our DHT. Within the the cell, nodes form a $1 - to - N$ connection with all remaining peers as seen in Figure 4.1. Cells are organized into a ring-shaped topology represented in Figure 4.2 similar to the one presented in Chord [42], and are connected to neighbouring cells, where each member forms a $1 - to - N$ connection with the neighbouring cell's nodes. This ordering style allows cells to keep track of cells that are before them (predecessors) and cells that follow them in the ring (successors). As with our implementation, cells have an incomplete knowledge of the network, thus routing messages to cells that are not present in the cell's predecessor our successor are made by routing the message to one of the nodes in the successor cell. This process is repeated by the successor until the target cell is found.

Akin to the cluster-based works cited in Chapter 2 our DHT is capable to organize and manage its own clusters, differing however as it is capable to be autonomous in its own balancing.

Our DHT was designed to be completely self-sufficient during its execution, being equipped with the mechanisms necessary to respond to churn.

Beyond the basic DHT operations like *insert* and *look-up*, we implemented three other operations, that are triggered automatically during the system execution. These operations are *split*, *merge* and *relocation* where:

1. Split. Is triggered when nodes join a cell and it reaches its maximum capacity. This operation splits the members and data of the cell into two groups. Upon this division a new cell is created using one of the groups generated, allowing for the network to grow and accommodate more members.

2. Merge. The merge operation allows for two cell to be combined, resulting in a cell with members and data from both cells. A merge operation is triggered when two neighboring cells have a less than ideal member count, allowing for data to be easily preserved.

3. Relocation. Devices leaving or joining the network, do not follow an uniform distribution, meaning that during the system execution some cells may have to handle a greater amount of churn than others. The relocation operation allows for a transference of nodes between cells, allowing for cells to balance their member count without the need for splitting or merging.

During the development of our work we considered a classical *asynchronous model* comprised of $\Pi = n_1, ..., n_k$ nodes. Additionally we assumed a *silent fail* failure model, in which nodes who fail stop receiving or sending members without previously advertising the crash to the other nodes, but do not behave maliciously.

4.2 Dynamic Logical Cells

As previously mentioned in Section 4.1 physical devices assume a virtual identity within the network called nodes. When a node joins the network, it is assigned to a cell. Inside a

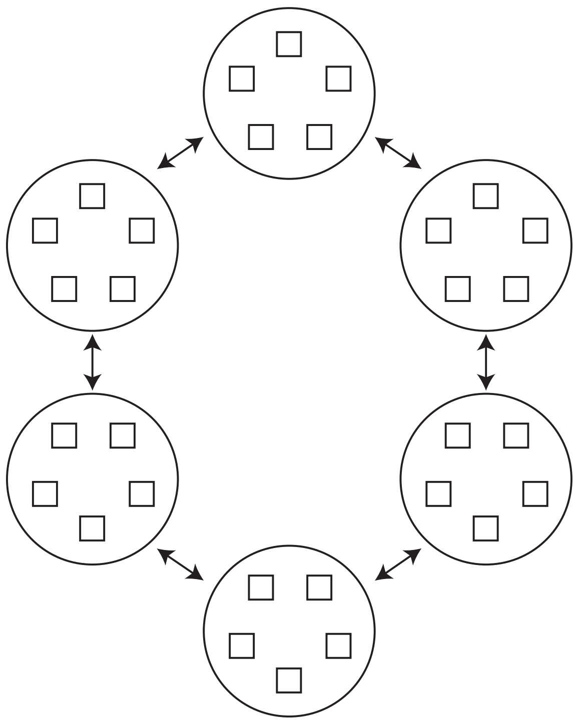

Figure 4.2: Exemplification of a network using our DHT. Nodes are grouped into cells, and cells are organize into a ring. The main communication outside of the cell is directed to the neighboring cells.

cell all nodes form connections with every other peer, allowing all nodes to communicate and share the same state. Thus in the network, every device is represented by its cell.

The system was designed to dynamically modify cells during its execution. As nodes are connected to every other node in the cell, all nodes know the exact size of their cell. The size of the cell is the main deciding factor for cells to know which of the aforementioned protocols to execute as exemplified by the Figure 4.3.

As mention, nodes only exist within the cell, thus cells have a minimum size of 1 member as cells without members do not exist since no node can represent the cell. The maximum size of the cell can be configured prior the system boot and during our implementation and validation of the proposed solution we defined the maximum cell size as 12 members. Between the maximum and minimum sizes, cells have an additional 5 configurable size thresholds. During this dissertation we have referred to cells with a member count of 6 to 8 as *good enough*, this is the ideal state for cells to be, as it is not at risk of loosing members nor is overspending their resources while synchronizing their state with the rest of the cell, means the cells is in a stable state.

Bellow this threshold, there is a risk for the network for data to be permanently lost as cells with *low* member count are in risk of disappearing. Additionally when cells have 4 or less members are marked as *dangerously low* and should take immediate action in the form of a *merge*. On the contrary cells with member count above the *good enough* threshold are marked as *high*. Although a *high* member count ensures the livelihood of the data in the cell, synchronizing a *high* member count cell becomes slower as nodes need to exchange more messages. When cells reach a member count of 10 are marked as *full*, the cell will avoid accepting new members if the neighboring cells can receive them and will forcefully start the *split* process.

To allow cells to dynamically change its data and membership during the system execution, we have implemented 5 protocols:

Join Protocol. Nodes joining the network will contact a pre-configured contact node. If no node is provided the node will create the first cell and start functioning as normal. However upon selecting a cell to join, all communication will be directed towards the selected cell, ensuring that joining nodes will cause the less disruption as possible to the existing cells.

Upkeep Routine. After joining a cell, all nodes periodically share information with their peers. This information sharing ensures that all nodes within the cell can agree and converge to the same state. Additionally when all nodes have knowledge of the remaining peers, the cell can take actions to dynamically enhance the performance of the cell.

Split Protocol. As nodes join a cell, the cell will grow until it reaches a *high* member count. To avoid saturating the network with too many messages from the *Upkeep Routine*, the cell splits. When a cell splits, it is divided into two new cells, each containing half of the original cell data and members. Thus the resulting cells can accept new peers.

Merge Protocol. Similar to joining the network, cells can lose members via nodes leaving the network. As the member count becomes *low* the risk of data loss get higher. Thus the *Merge Protocol* allows for cells to join with a neighboring cell. Resulting in a new cell containing all data and members from the previous two cells.

Relocate Protocol. The *Merge Protocol* may have a negative impact on the network thanks to the amount of synchronization necessary to join two nodes. The impact of this consequences are even bigger during periods of high churn and network instability. As a way to preserve cell integrity, the *Relocate Protocol* allows for cells to share a small amount of peers, thus reducing the synchronization needed to re-balance cells.

The introduced protocols and their relations to the cell size are summarized in Figure 4.3 and 4.4. The ultimate goal of the implemented protocols is to allow for networks built with Thyme system to be dynamically self-managed during their execution. Additionally we strive to create a self-managing network that will work to achieve an optimal configuration, capable of bringing resilience and fairness to all devices participating.

As described the execution of the *Split*, *Merge* and *Relocate* protocol will alter the network topology. All of the mentioned protocols are initiated by a single node in the cell, this node is called the *cell leader*. The *cell leader* is elected during the *Upkeep Routine* using the disseminated *stability index*, this election is further elaborated in Section 4.2.2. The election of the *cell leader* follows a set of deterministic rules applied by all members of a cell, and is responsible to initiate and coordinate all processes above mentioned according to the cell state.

Finally to support the implemented operations, cells need to save their state. Through the rest of this work we defined state as the data present with the cell caches $\langle id_{\text{cell}}, id_{\text{lead}}, M, P, S, L, J\rangle$ where:

- id_{cell} the unique identifier of the cell;
- id_{lead} the address of the cell leader;
- M a collection of all the addresses of the members of the cell;
- P a list of the N cell's states that precedes the cell in the ring;
- S a list of the N cell's states that succeeds the cell in the ring;
- L a list of addresses whose nodes have left the cell recently;
- J a collection of addresses whose nodes have joined the cell while the cell was executing any of the implemented operations.

During the system execution nodes inside a cell will strive to agree and share the same state. However the implemented protocols were designed with a perspective of *best effort*, thus we expect situation were the nodes will diverge their state, these situations will be further explored in Section 4.3.

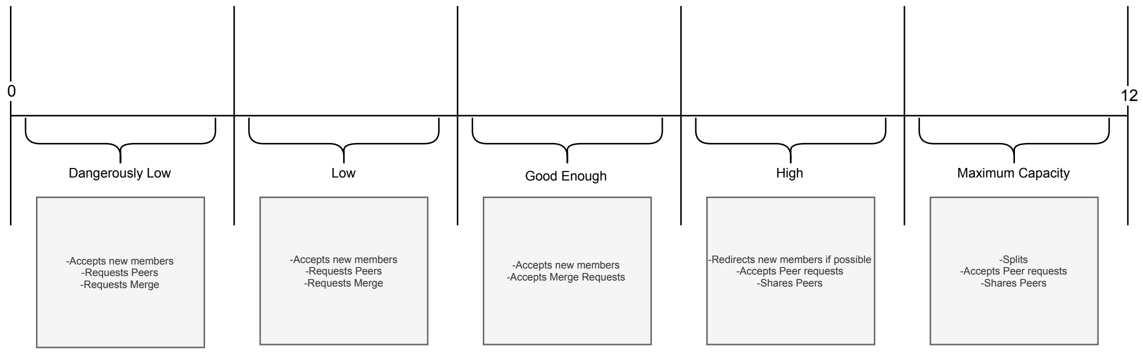
0
12
Dangerously Low
Low
Good Enough
High
Maximum Capacity
-Accepts new members -Requests Peers -Requests Merge
-Accepts new members -Requests Peers -Requests Merge
-Accepts new members -Accepts Merge Requests
-Redirects new members if possible -Accepts Peer requests -Shares Peers
-Splits -Accepts Peer requests -Shares Peers

Figure 4.3: Scheme of how the cell size enables different dynamic loading mechanism.

4.2.1 Joining the Network

Devices joining the network will select and contact nodes from a list of possible entry points in the network. After selecting a contact node, the *Join* procedure is then started.

The joining node sends a *JoinCellRequest*, this message contains the joining node's address and stability index.

As contact nodes are selected from a list of nodes already in the system there is a possibility for the *JoinCellRequest* to be delivered to a node whose cell is reaching its maximum capacity. When the above situation is realised, the receiving node will chose a random node in its successor cell and forward the *JoinCellRequest* to the selected node. The forwarding of *JoinCellRequest* messages is always done trough the successor to ensure the lowest risk of the message being stuck in a cycle. This process can be repeated a configurable number of times, or until a cell with optimal member count is reached. If the maximum forward count is reached the cell will accept the new member regardless of its size.

When the *JoinCellRequest* message is successfully delivered to a cell, the contact node adds the joining node to its membership view and replies with a *JoinCellReply* message. This message contains the identifier of the cell, a collection of all members address of the cell, and a collection of all their stability indexes. Additionally, the message contains all the data items the cell is responsible for. This message ensures that after the join process is concluded the newly joined node is up-to-date with the cell, allowing it to be a fully functional member right after joining.

After the join protocol is finished, both the contact and joining node, will update their peers within the next round of the *Upkeep Routine* as we will elaborate further in Section 4.2.2, to ensure that all remaining peers take notice that a new node has joined the cell.

However the first device that joins the network will not have a contact node to join, as there are no nodes in the system yet. Furthermore the node cannot join any cell. Thus the first node after a configurable time without receiving *Hello* messages, creates the first cell. The first cell is the only cell that has a fixed identifier. Through this work we used Universally Unique Identifier (UUID) as unique identifiers of nodes and the first node as an *UUID* of *00000000-0000-0000-0000-000000000000*. By enforcing this rule, we safeguard our implementation from cases where multiple devices assume to be the first in the network.

4.2.2 The upkeep routine

To ensure that all nodes within a cell agree and converge to the same state, we have implemented the *Upkeep Protocol*. The *upkeep protocol* ensures that nodes within the cell can converge to the same state, by constantly updating the nodes when a new node joins the cell. Furthermore the upkeep routine is responsible to detect when nodes silently leave the cell. First to be able to detect nodes failure, nodes are expected to periodically send an *Heartbeat* messages. For the context of this work, we refer each execution of the upkeep routine as a *round*.

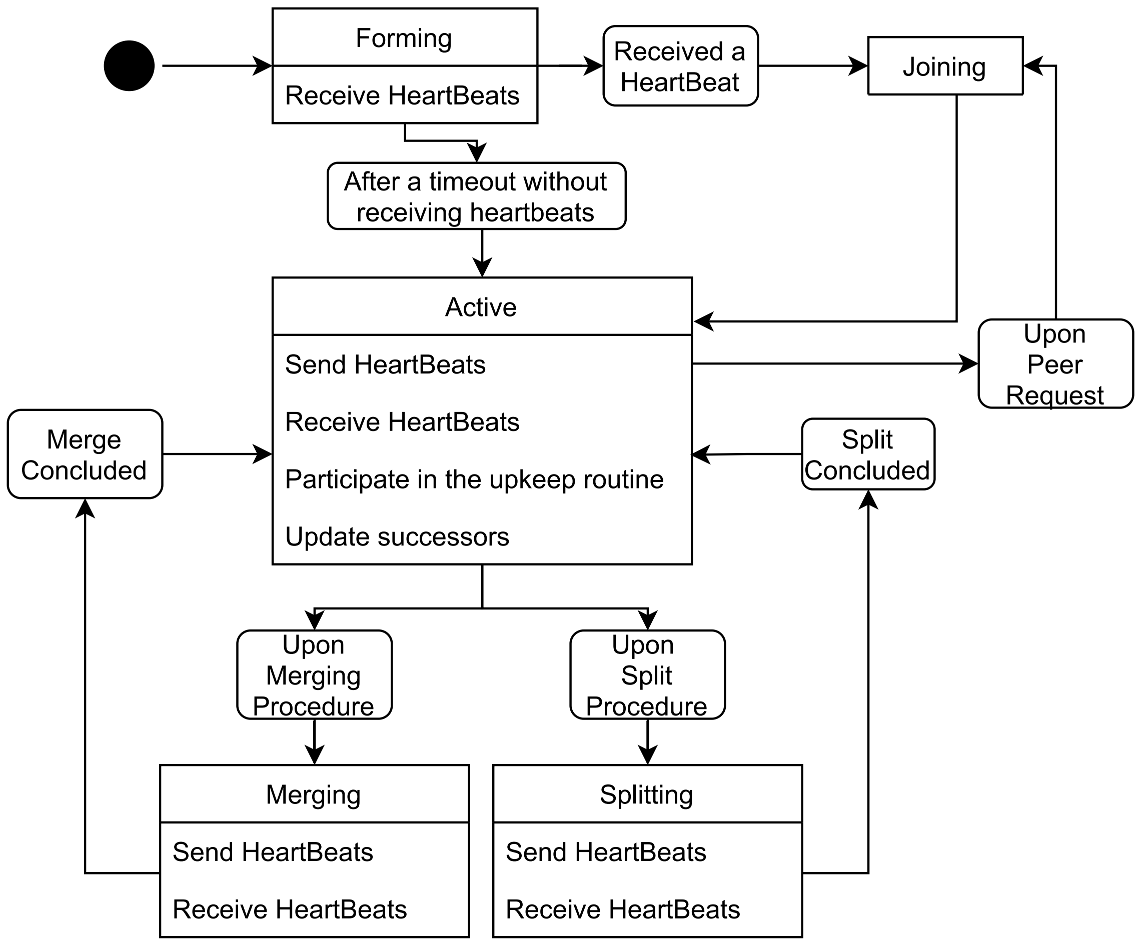

Figure 4.4: The possible cell states and allowed operations in each state.

The *HeartBeat* message contains:

- The cell's UUID. Used to distinguish new messages from delayed messages sent during any node relocation process.
- A collection of successor and predecessor nodes. Used by all nodes within the cell to agree their position inside the ring shaped network and to contact nodes during external heartbeats, further explained in Section 4.2.3.
- A list of known members and their stability index. Used to propagate new members in the cell and to assist all nodes to agree on the most stable peer during cell leader election.
- A pair with an identifier of the data in storage and a timestamp of its latest modification. Nodes calculate an hash of all keys of the data present in their storage. When the hash received in the message differs from the hash of the storage, the timestamp allows for nodes agree on the most recent modification. By exchanging an hash of they keys of the data, we achieve a smaller message.

This structure is ideal to ensure that peers can update and converge to the same state with the minimum data necessary. Furthermore we aimed to achieve synchronization with the lowest number of messages as possible. Thus in our implementation we aimed to achieve eventual synchronization, as during each round of the *Upkeep Routine* each node sends only *Heartbeat* messages to a configurable fraction chosen at random.

Storage Hash A	Time Stamp A
8	"2021-09-01 09:01:15"

Storage Hash B	Time Stamp B
8	"2021-09-01 09:32:18

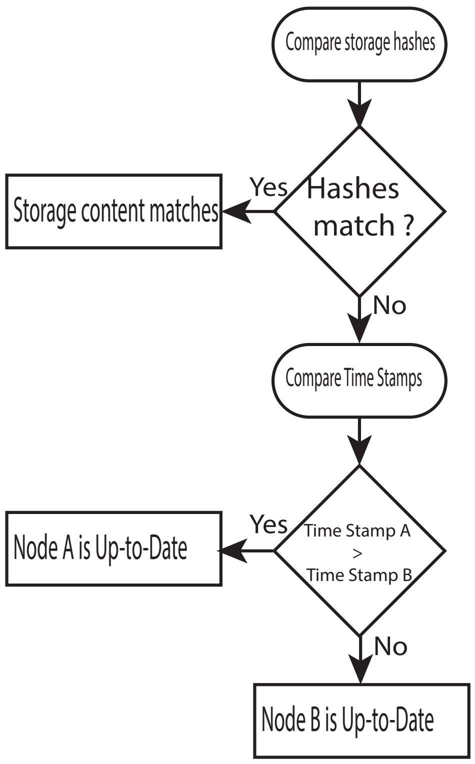

Figure 4.5: Nodes up-to-date decision process. In this example two nodes names A and B are deciding which one is the most up-to-date.

Additionally to eliminate the permanent propagation of nodes that no longer belong to the cell, each nodes caches the addresses of nodes that leave the network in their *Left Recently Cache*. Upon receiving an *Heartbeat* message, the receiving node, first checks if the sender's address is currently stored in their *Left Recently Cache*, ignoring the message if yes. Additionally messages that are delayed followed any dynamic loading operation as explain in Section 4.3, are detected and suppressed using the *Left Recently Cache*.

Otherwise the receiving node verifies that the UUID present in the message matches the UUID of its cell, as the *Heartbeat* messages are delivered only within the cell they are created. Messages with different *UUID* were sent before a dynamic loading operation that altered the cell membership and were delayed until its completion, thus the information they hold it is no longer relevant and the receiving node does not process the message during the *Upkeep Routine*. After discarding the message, the receiving node informs the sender that they no longer belong to the same cell, by replying with an *HeartbeatNack* message.

Continuing by processing the message, the receiving node checks if it has not been updated with the cells successors and predecessors, updating them accordingly to the ones received in the message, the update of the cell neighbors will be further explored in Section 4.2.3.

Furthermore as exemplified in Figure 4.5 the receiving peer verifies the storage hash received. If the hash received it is not the same as the hash of the storage of the node, the node uses the timestamp provided in the message to figure which of the nodes is outdated. If the receiving node is outdated in sends a *RequestUpdate* message to the sending node. The *RequestUpdate* message, indicates that the sending node is out-of-date and is requesting to receive the storage belonging to the node contacted. Upon receiving a *RequestUpdate* message, the node replies with an *UpdatePeer* message, containing all the data items present in the node. However it worth noting that this mechanism is only triggered by the receiving node, and if the sending node is outdated, no update mechanism will trigger as a result of this upkeep routine. As eventually the sending node will receive an up-to-date *Hearbeat* message trigger itself the above process. Finally after consuming all data present in the *HeartBeat* message, the receiving node replies with an *HeartbeatAck* message, confirming to the sender that this round of the *Upkeep Routine* was a success. After sending an *Heartbeat* message, each nodes caches the target address and waits a configurable number of *Upkeep Routine's* rounds. If an *HeartbeatAck* is received before this number is reached, the address is removed from the cache. However if no reply or an *HeartbeatNack* is received, the node removes the target from its membership and adds it to the *Left Recently Cache*. However as the *HeartbeatNack* message carries the UUID of the cell, as a small optimization implemented, the receiving node can look in its neighbors list to check if any cell matches said UUID. If this search is successful the node update that cell view to accommodate the new node.

The upkeep routine doubles as a way for nodes to share their stability indexes, and elect the most stable of them as the *cell leader*, using as a tie breaker the alphabetical order of the most stable peers UUID. Finally the cell leader during its *Upkeep Routine* takes the role of the coordinator of the cell, allowing it to make decisions based on the cell state and reorganize the cell members with neighboring cells, initiating the *Dynamic Loading Operations* as described in detail in Section 4.3.

4.2.3 Upkeep outside of the cell

As of now, we have presented the nodes responsibilities to maintain the stability of a cell. However as we previously described, a network is formed by multiple cells. So far, the only communication made outside the cell, was used while joining the network. Thus

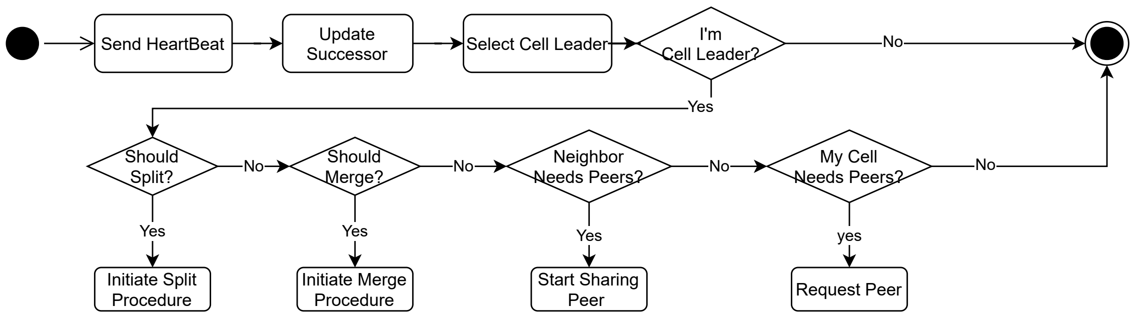

Figure 4.6: The cell upkeep routine.

during our implementation to ensure communication across all cells of the network, the cell employs a similar mechanism to the one describe in the above section.

Adding to the already established in-cell *Upkeep Routine* in Section 4.2.2, all nodes synchronize their cell according to neighboring ones. This routine is executed in each round of the *Upkeep Routine*, as long as the cell is not the only in the network.

During this process, each node selects a configurable number of nodes from the most direct successor and sends them an *UpdateSuccessor* message, containing the UUID of the target cell, the sending cell's UUID and list of its members, and a list of the cell predecessors. Inspired by Chord [42] ring maintenance logic and similar to the in-cell mechanism, to detect if the selected targets are still in the network, the sending node caches the addresses of the receiving nodes. By receiving no reply within a configurable period of time, the sending nodes removes their addresses from the view of its successor and from the cache.

Upon receiving an *UpdateSuccessor* message, the receiving node, first, checks the sending node target cell UUID with the receiving node cell UUID. If the UUIDs do not match, the receiving node replies with an *UpdateSuccessorNack*, containing the cell UUID, and discards the rest of the message. However if the UUIDs do match, the receiving cell proceeds to analyse the rest of the message. It takes the view provided by the sending node, and updates the view of its most direct predecessor according. Finally updates the rest of the predecessors with the information provided, and replies with an *UpdateSuccessorAck* message, containing a list of peers of the receiving node, and list of the node successors.

Processing an *UpdateSuccessorAck* message is similar to the above. The receiving node updates its view of the successor, with the one received from the message, following by updating the remaining successor. However in case of receiving a negative reply in the form of a *UpdateSuccessorNack* message, the receiving cell removes the sender from its view of the successor, in case of the UUID of the cell provided matches any UUID of the receiving cell successors, it adds the sender address to that cell.

With the above procedure terminated, it is possible for either the sender or the receiver to be left with a view of a cell with no members, in that case the view of the cell is discarded and it is selected a new neighbor from the list of predecessors or successors accordingly.

4.3 Dynamic Loading Operations

In the previous section, we discussed the approach taken to sort devices into our virtual groups, called cells. In this section we will expose the implemented methods that allow the network to dynamically change its topology during its execution.

4.3.1 Split Operation

As described in the Section 4.2.1, when a device comes online and tries to join the network, it looks for available cells or creates one if no cell is present. However, as exemplified by the first panel of Figure 4.7, as more and more devices join the network and are placed in the existing cell, the cell reaches its limit capacity (seen the in the second panel), meaning, the guarantees of data availability and fail proof, do not offset the work done by each of the nodes to send and reach a state agreed upon. As every stored item is replicated by every member and, each round of the upkeep routine, updates only a small amount of devices each time, it becomes increasingly difficult for every member to agree upon a single cell state.

Thus within each *upkeep routine*(4.2.2), the cell leader takes upon the responsibility to monitor the cell, using the information exchanged within the *Heartbeat* messages. When the cell reaches its configured maximum capacity, as seen in the third panel of Figure 4.7 the cell leader starts the split operation. First it generates a new UUID, to be the identifier of the cell that will be created. Selects half of its peers to be the starting members of the new cell. Finally, selects half of the key items in the cell to be moved with the leaving members. After this process is concluded, the *cell leader* informs the rest of the cell of its decision, by sending a *SplitRequest* message, containing all the items generated previously.

Upon receiving a *SplitRequest*, the receiving node marks its cell as *splitting* and checks if the collection of addresses contain its own address. If the node's address is contained on the received list, the node will leave its cell to be part of the newly created cell. Join the new cell follows the following steps: First the moving node sets its cell address as the new UUID received in the message, then removes from its membership the nodes whose addresses are not present in the relocating list, proceeding to add them to the *Left Recently Cache*. Then deletes all content from its storage whose keys were not present in the *splitRequest* message. Finally the node calculates the resulting state of the previous cell without the members whose address was present in the message, and adds that view to the successor list and its most direct successor, if the cell has no predecessors, to close the ring shape, that our network is based on, the node adds the calculated view onto the predecessor list as well as exemplified by the forth panel of Figure 4.7.

Similarly to the the process described above, is the case where the receiving node does not leave the cell. Instead, the node creates a view of a cell with all the information received in the *SplitRequest*, and adds it as the node most direct predecessor, also adding it to its successor list if necessary.

While the cell is marked as *splitting*, joining nodes are flagged as *joined while dynamic loading*. These nodes are treated as partial members of the cell. While the nodes are still added to the cell membership, they do not participate in the dynamic loading operations, as they were not present when the *SplitRequest* was sent.

To complete the *split operation*, both cells need to return to the *active* state. This process happens in a *best effort* environment, where nodes exchange *Heartbeat* messages and keep track the number of rounds which no update was necessary to any of the involved nodes, when a configurable number of rounds with no changes to states is reached, the nodes update their views of the cell marking it as *active*. It is possible however for nodes to fail at any point during this operation. If the majority of the nodes in one cell fail, the rest wont be able to achieve the conditions necessary to terminate the *split operation*, thus, nodes wait a predefined period, after which, the cell is marked as *active*, regardless of its state.

Finally after the *split operation* is concluded, the nodes that were flagged as *joined while dynamic loading* are updated with a *FixSplit* message, this message contains the data items and nodes that left the cell during the *split operation*.

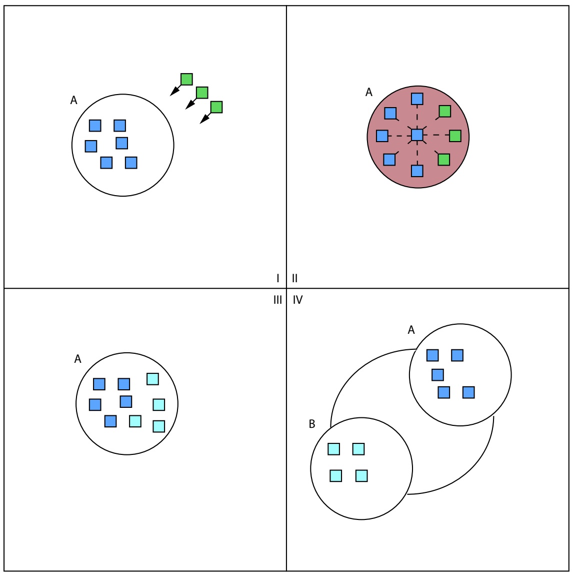

Figure 4.7: Simplification of a split operation. i) Nodes join a cell. ii) The cell reaches its full capacity, every node within a cell needs to keep active connections to every other node. iii) The cell leader selects half of the members to move to the new cell. iv) The cell splits the new cell becomes the predecessor of the split cell.

4.3.2 Merge Operation

With the joining of nodes into the network, the network develops into a ring shape topology, however akin to the joining of nodes, for diverse reasons the devices can leave the network. As data is only replicated within the cell it is stored, a cell losing all of its members implies that all data in the cell will be permanently lost. Furthermore neighboring cells spend resources in their upkeep routine, to synchronize with a cell that is in risk of leaving the network.

To bring an answer to this situation, during the development of our work, we implemented the *Merge Protocol*. Built on the same foundations as the *Split Operation*. As explained in Section 4.2.2 the cell leader during its *Upkeep Routine* monitors the size of the cell. Whenever the cell loses members, and its membership falls bellow the *Good Enough* threshold, the cell leader tries to find a suitable neighboring cell to merge with. As the cell only has information about the neighbor's size, an ideal candidate should be able to receive all members without reaching the its *high* threshold. If no cell matches this criteria, the *cell leader* awaits until the next *Upkeep Routine* round. However neighboring cells that have a an *high* member count can spare a few nodes to join the cell, avoiding the need for the *Merge Protocol* to be executed. This process is further elaborated in Section 4.3.3.

However if the cell leader finds a neighbor that successfully meets the presented criteria, the *Merge Protcol* starts. First the cell leader selects the most stable member accordingly to the information that was sent during the *Upkeep Routine* as explained in Section 4.2.3. After the most stable peer has been selected, the cell leader delivers it a *MergeRequest* message. This message contains all addresses of the nodes in the requesting cell, as well as the cell *UUID*.

Upon the reception of a *MergeRequest* message, the receiving node checks if the cell is already participating in another dynamic loading operation. If yes, it replies with a *MergeRequestNack* message, containing the cell UUID. However if the cell is not performing any dynamic loading operation, the receiving node sends a *MergeRequestAck* to every member of both cells, this messages contains all addresses of the participating members.

After receiving a *MergeRequestAck* message the nodes mark both of their cells as *merging*, and create a membership containing all nodes from the participating cells, nodes whose cell is merging, set their cell UUID as the UUID of the cell is merging into. Thanks to the UUID of the participating cells being present in the *MergeRequestAck* message, the nodes make use of the neighbors list to know exactly if the cell is being merged to a successor or predecessor and are able to fix the the list accordingly.

Note that the goal of the *Merge Operation* is to safeguard the data present in both cells, however no information related to the data is being transmitted in the messages so far. We strive to implement the *Merge Operation* as light as possible, and delay the synchronization of data to a period where the membership is already established. Thus after the membership has stabilize, nodes mark their cell as *converging merge*, where the nodes will shorten the time between the *Upkeep Routine* rounds. During this time, any discordance in the storage storage of two nodes, is resolved by both nodes exchanging storage data and adding each other data to their own storage, allowing for a quick synchronization of both cells.

Similarly to the *split operation*, nodes that joined during the merge process are flagged as *joined while dynamic loading* and updated at the end of the operation using a *FixMerge* message, with the difference that unlike the *FixSplit* message, this one doesn't updates any storage, as that will be made during the next upkeep round.

Finally in the case of the merging cell selects a busy neighbor and receives a *MergeRequestNack* message in response, it selects the neighboring that was not selected in the previous attempt. If all neighboring cells are busy, the cell will stop requesting to merge until the next round of the upkeep routine.

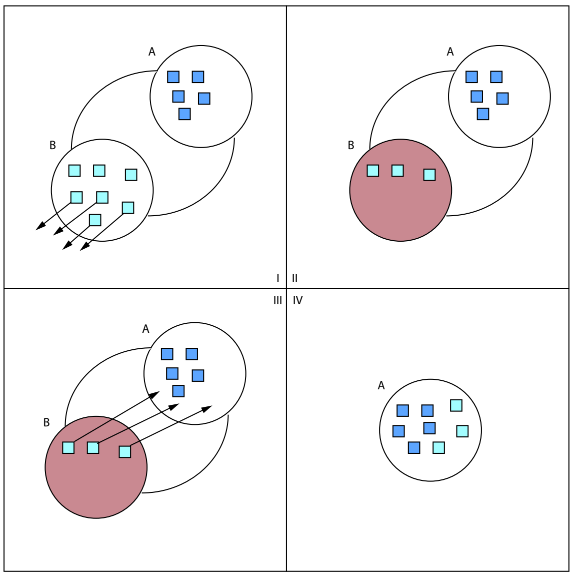

Figure 4.8: Simplification of a merge operation. i) Nodes leave the cell. ii) The cell becomes low in members. iii) The cell leader selects one cell to merge with. iv) The cells merge, resulting in a cell with both memberships combined and all data contained in the initial cells.

4.3.3 Peer Relocation

As of this section, we have presented our mechanisms that allow the network to develop into a dynamic ring-shaped topology, by creating cells with the *Split Protocol* or by removing cells using the described *Merge Protocol*. However as we have described in Section 4.3.2, there are situations where the network cannot evolve. As detailed, when a cell leader tries to initiate the *Merge Protocol*, it looks for suitable cells in its neighbors. A suitable neighbor is a cell that can receive the nodes without reaching its *high* member count state. This safeguards the network from having two cells merging only to split right after. Thus, to avoid the situation mentioned, we have implemented two operations that allow for cells with low member count to stabilize without the need of the *Merge Operation*.

As cells share their states with neighboring cells during their *Upkeep Routine*, a cell has an up-to-date information about its neighbor's state. We built upon this system, the *Peer Relocation Protocol*, where cells were equipped with the ability of sharing a single peer.

The *Peer Relocation Protocol* helps avoiding large periods of synchronization caused by the *Merge* operation, as intuitively synchronizing a single node will result in less synchronization needed compared to synchronizing two cells in the *Merge* operation. The *Peer Relocation Protocol* can be triggered in two ways:

Proactively initiated by a cell with low member count. When a cell is below the *good enough* size threshold, the cell leader will try to stabilize the cell. The cell leader will select a random node from one of its direct neighbor as seen in the second panel of the Figure 4.9 and deliver it a *RequestPeer* message, containing the UUID and membership of the requesting node. To avoid destabilizing the leader of the cell, this message can only be processed by non-leader nodes, being forwarded to a regular node if the selected node happens to be the cell leader. Upon receiving the *RequestPeer* message, the node clears all caches and data from the current cell and sends a *JoinCellRequest* message, to one of the addresses present in the *RequestPeer* message. This node is then treated as a newly join node, and the process of joining the cell described in Section 4.2.1 is executed.

Initiated by a neighboring cell. When a cell is above the *good enough* size threshold, the cell leader will begin to look for cell with low member count that can receive members. When a cell is found, the cell leader delivers to one of its peers a *RelocateRequest* message. This message contains the UUID and membership of the target node. The receiving node will then leave the cell and execute the same process described above.

It is however worth noting, *Peer Relocation Protocol* aim, is not to replace the *Merge* operation. During periods of high instability where multiple nodes leave from multiple cells, it is not feasible to share peers. As multiple cells reach a *low* or even *dangerously low* member count, the *Merge* operation is still preferred.

4.4 Integration with Thyme

Through the course of this work we implemented the proposed solution to Thyme, essentially constructing an additional layer to the Thyme system allowing it to be further augmented with a new component. Where in the previous iteration of Thyme, the cells were statically managed via a configuration file prior the system boot. The the fusion of our implementation with Thyme, allows for a system capable of dynamically regulate

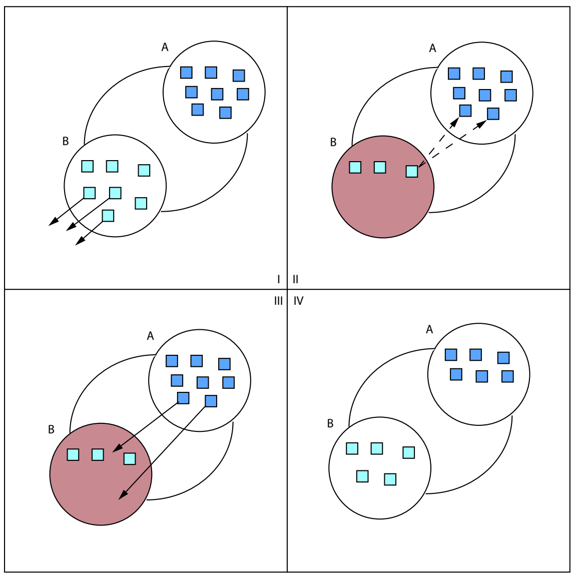

Figure 4.9: Simplification of a proactive relocation operation. i) Nodes leave the cell. ii) The cell becomes low in members, however it cannot join a neighboring cell as it is reaching its maximum capacity. iii) The cell leader selects a random number of nodes from the successor cell and request them to join the cell. iv) The selected nodes, leave their cell to join the requesting cell, balancing the member count of both cells.

cells during its execution. The new version of Thyme is able to adjusted the number of cells and their member's count to better fit the needs of the current membership. The mechanisms proposed in this chapter, allow nodes to form connections and gather information from the network. Implementing our solution with Thyme, allowed for an extension of Thyme's routing layer allowing messages to be delivered within the cell.

As a result, mobile devices, using the Thyme system, are now equipped with the necessary mechanisms to accept the responsibility of managing the network. We took advantage of the communication used to maintain the cells in the network explained in Section 4.2.2) for nodes to exchange state within their cells, allowing for nodes in the same cell to converge to the same state, both in connections with the cell and in terms of the data they store.

By having all nodes in a cell sharing and converging to the same state, any one of those nodes can be the contact point of the cell. Meaning that a message targeting a cell can be delivered to a single one of its nodes. As all nodes can receive and reply to messages, thus enabling operations like data retrieving can be made in a concurrent way.

4.4.1 Thyme Components

In this section, we present our solution to augment Thyme architectural structure (Fig. 4.10), based on the authors' proposal in [7] and further enhanced by [45], highlighting the layers that were created or enhanced during the context of this work.

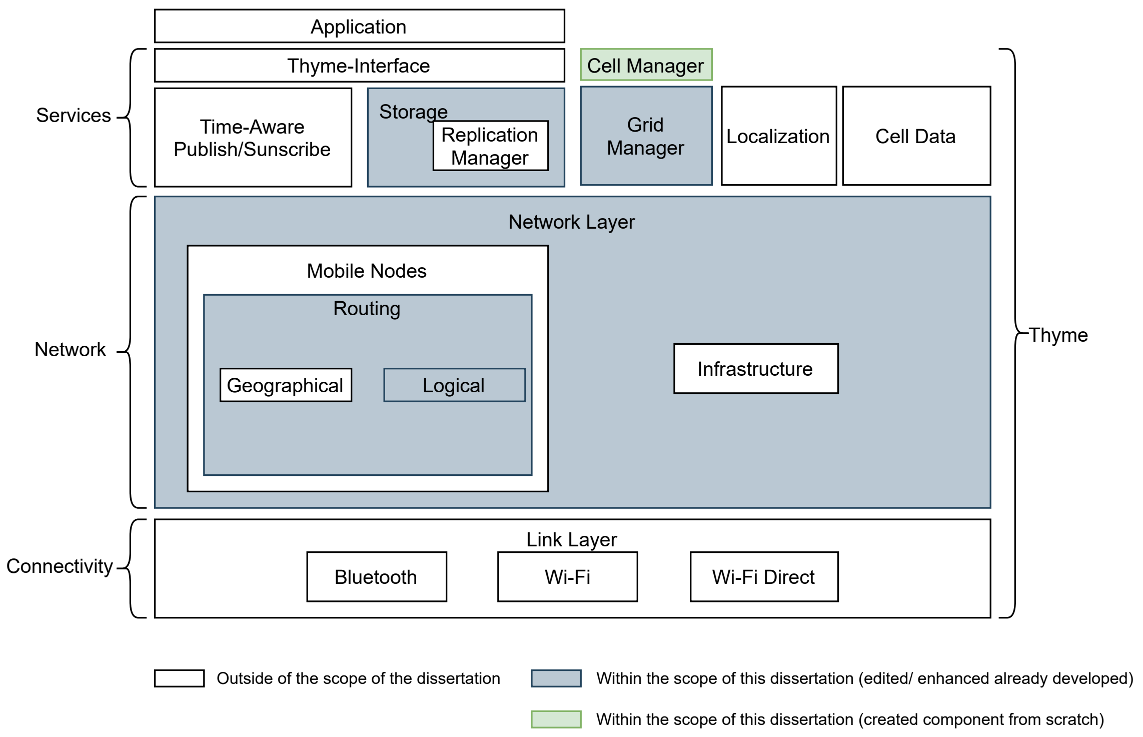

Figure 4.10: Thyme's architecture. Adapted from [7].

DHT Manager. The *DHT Manager* is a completely new component, created during the progress of our work resulting from the integration of our DHT with Thyme. The *DHT Manager* manages the connection of our augmented Thyme's cell, regulating its size and controlling the cell's state. This component holds the information relative to the cell's state described in Section 4.2. In this component we implemented the maintenance routines described in Section 4.2.2. Furthermore this component interacts with the *namespace manager* in order to assign new members to their cell.

Namespace Manager. This module is responsible for managing the network space, allowing for the division of the network in cells and indexation of all devices within. Said devices can group with peers within their proximity forming *Geographic Cells*, each cell representing a fraction of said geographic space. Or form connections on a complete logical way, without taking the geographic distance into account on *Logical Cells*. In order to find a solution for the problem proposed in Section 1.2, this component was modified during our work. The *Grid Manager* is responsible to start the process of joining a network, as well as process requests of incoming devices joining the network, mapping them to a cell.

Storage. The storage layer stores and manages all the data items present in the devices, furthermore this component is responsible to interface with the P/S layer enabling the persistent storage needed to feed the P/S layer as described in Section 3.1. During the development of our work, this layer was augmented to participate in the *Dynamic Loading* operations. This layer was augmented to keep track of its content, calculating an hash of the keys on every object insertion or removal. Additionally the *Storage component* saves the timestamp of the latest storage content change. All the described mechanisms allow for this component to independently synchronize with the rest of the cell, allowing for all the changes in one device storage to be reflected in all devices in the same cell.

Network Layer. This layer supports the communication between Thyme instances in order to connects the storage systems as well as the discovery of new devices in the network. Devices will periodically broadcast small messages using this layer, and devices coming online will listen for them in order to join the network. During our work, we extended this behaviour with the goal of offering a way for devices to join the implemented cells.

4.4.2 Components integration

Devices using Thyme will make use of the *Network layer* to periodically broadcast an *Hello* message, containing the sender identifier. When a device comes online, the device *Namespace manager* listens for *Hello* messages coming from other devices in the network. The process of joining a cell explained in Section 4.2.1 is done trough the device's *Namespace manager*. Upon being assigned to a cell, any communication within the cell is made trough the *cell manager* component.

4.5 Summary

In this chapter we have presented and describe the Thyme implementation and components we have built our DHT underneath. Our solution allows for Thyme to dynamically manages the cells in the network and their composition. All the mechanism and protocols described strive to build a resilient network, capable of withstanding an high level of faults. Additionally to ease the communication within the network, all the protocols allow for synchronization with the minimum amount of messages. Furthermore, our implementation focus fairness as every operation can be made from any node in the cell, even the roll of *cell leader* is a rotating position, ensure that every device contributes with the same amount of resources.

In conclusion our implementation all these mechanisms allow for the creation of a network ever evolving, capable of changing its topology to respond to the problems caused by the workload unbalance in the network.

5 Evaluation

In this chapter we will present the results of the experimental evaluation done to the implementation produced. First, In Section 5.1 we describe the methodologies used to produce this evaluation, as well as the goal of this chapter. Before showcasing the results, in Section 5.2 we introduce the simulator used to run our tests. Finally in Section 5.3, we will present the results of the studies performed and conclusions produced.

5.1 Evaluation Methodologies

During the course of this work, we have developed our DHT to be a resilient distributed system, capable of withstanding large windows of churn by dynamically change its structure accordingly. Each participant of the system shares the same responsibility in the network, and spend the same amount to resources to bring balance and fairness to the network.

The system must be able to create a ring-shaped network starting from a single device. Furthermore the system must be able to employ the protocols discussed in Section 4.3 to ensure that no device spends more resources that it s peers and no data is lost during periods of heavy churn and network instability.

The goal of this chapter is to prove that our solution delivers fairness and resilience to the network under different scenarios and environment conditions. Finally by the conclusion of this chapter we will answer the following questions:

1. What is the behavior of our system under a regular execution?
2. How does the system behaves when data enters the network?
3. How quickly can the system recover when facing large windows of churn?
4. How much time it takes to re-synchronize every node that participates in dynamic loading operations?

To measure the performance of our implementation we chose to implement a battery of controlled scenarios to be run in a simulator. By using carefully crafted scenarios we can retain the maximum amount of useful metrics from each test, giving higher importance to:

- Number of dynamic loading operations (*split*, *merge*, *relocation*) executed as described in Section 4.3;
- Average re-synchronization time after operations;
- Number of messages sent due to the dynamic loading operation;
- Number of messages sent to synchronize a publication;
- Average time until a publication is known by all members of the cell;
- Number of messages sent;
- Size of the messages sent.

Our implementation was tested in two ways. The first tests were done to prove the system's functionality. This batch, helped us prove that our implementation was working like it was supposed to, if the cells were splitting or merging when their sized reached the thresholds and if data was being correctly handled on publish and relocation. Finally the second batch of tests allowed us to make measurements and comparisons by changing the test's variables.

5.2 Simulated Evaluation

We will now be exposing with greater detail the simulated environment used during the course of this work to validate our implementation.

To fully test our implementation we would need a large number of mobile devices to achieve a developed network. However, gathering such amount of devices would not be feasible. Thus to design and create our scenarios we used custom trace based simulation [23], previously developed in java and further augmented by Thyme's many contributors. This simulator connects instances of Thyme by reworking the network layer, allowing for multiples Thyme instances to be run in a single simulation, each running in an independent thread. Furthermore as Thyme utilizes some features or classes dedicated to the Android operating system, the simulator emulates said features to allow for the simulation to be hosted outside of mobile environments.

The traces offered by the simulator help build a detailed scenario where we have full control over the time in which operation is executed.

As the network needs time to converge after starting, in our scenarios, we started our measurements only after the first 60 seconds, and operations would start being issued after that. Similarly in order to guarantee that all pending operations have time finish its execution, no operation is issued 60 seconds prior the end of the test. Finally, to eliminate possible random errors that causes inaccuracy, all tests and theirs variations were repeated 5 times, thus the results we will present are an average of all the tests repetitions.

5.3 Absolute Metrics Evaluation

In this section we will present the results of the tests performed to our system. With these tests, we are able to prove the behavior described in Chapter 4. Furthermore, the results in the following sections are presented in order of complexity of the tests performed. Presenting first, the tests performed with the least variance. Allowing us to organize each test in a way to lay a foundation for comparisons and conclusions for the next ones.

5.3.1 Stable execution test

To understand how the system naturally behaves, we have developed a test with minimum variation as possible. The scenario created for this tests, does not contemplate any data publication nor changes in the system membership. As the only membership changes happen at the start of the test and no device joins or leaves the network during the rest of the test. Additionally as no data is being published, nodes do not incur in periods where their storage diverge. Since during the test, nodes do not join the network, there will be no *JoinCellRequest* or *JoinCellReply* circulating the system, as an additional consequence, no *Split* operation other than the ones generated by the boot of the network will be triggered. Akin to the previous situation, as no nodes leave the network the *HeartBeat* messages will always receive a response and no *Merge* operation will be issued. Finally, as we have established a static membership in the network, no cell will meet the criteria to perform *Peer Relocation*.

As we have previously explained in Section 4.2.2, the majority of the messages and operations are enabled by the *Upkeep Routine*. The *Upkeep Routine* was implemented to be run periodically. To study the implications of this routine, we run tests executing the previously explained scenario, varying only the frequency at with the *Upkeep Routine* is triggered.

All variations of the tests were performed with 200 nodes total. On network boot, we had 25 nodes join the network, and each tenth of a second another 25 nodes joined until all 200 nodes were part of the system. Additionally network monitoring were began only 60 seconds after the last node joined. This process allowed us to take measurements and infer conclusions on a network that had already stabilized.

The results of our tests were compiled and presented in Figures 5.1, 5.2 and in Table 5.1.

In Figure 5.1 we have displayed the average number of messages sent each second during the execution of the tests. We have presented the results of the three different parametrizations in the same Figure in order to better compare the impact of frequency

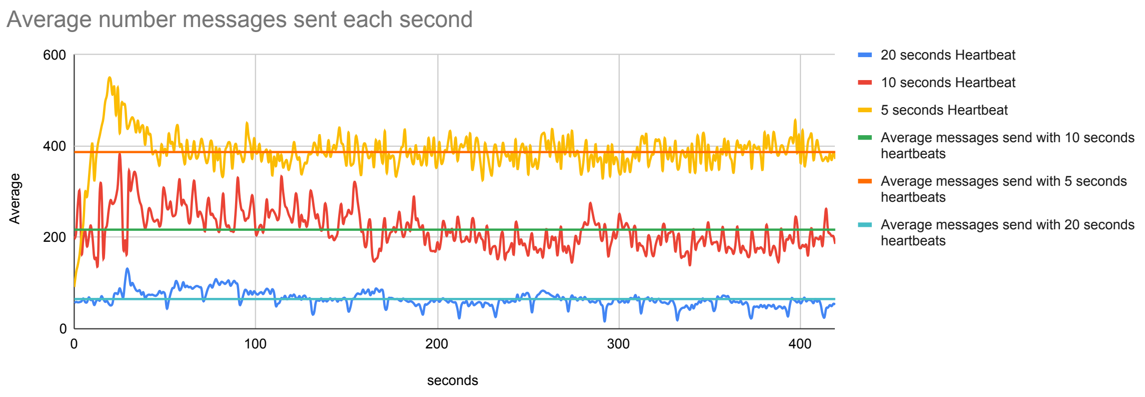

Figure 5.1: Representation of the number of messages sent each second and their relation with the frequency of the upkeep routine.

of heartbeats. As expected changing the frequency of the *Upkeep Routine* rounds, has a direct impact on the number of messages sent each second. By changing the frequency of the *Upkeep Routine*, we change the frequency that *HeartBeat* messages are sent. So we expect the number of *HeartBeat* messages to be directly proportional to the frequency of the *Upkeep Routine*. Additionally as each upkeep triggers an exponential number of messages, as every *HeartBeat* delivered, is replied and a number of *Ping* messages equal to the number of members of the cell is created. The total number of messages, however, follows a logarithmic relation with the frequency of the *Upkeep Routine*.

Table 5.1: Average size of each message type.

Message	**Average Size in Bytes**
HeartBeat	1411
HeartbeatAck	1
HeartbeatNack	44
Ping	38
PingAck	4
PingNack	38
UpdateSuccessor	889
UpdateSuccessorAck	879
UpdateSuccessorNack	44
Average of all messages	257

We use the same tests to profile the size of each message type and the number of times it was sent. In Table 5.1 we laid a comparison between each different type of message and their average size. And in Figure 5.2 we presented the relative number of times each message was sent, ordered in a way, that the most common messages are organized from left to right.

Immediately we may note that converging the cell state within all members and with neighboring cells, comes at a cost. As the *HeartBeat* and *UpdateSuccessor* messages average to be the largest. We believe this situation originated from a bad decision in the implementation process as the *HeartBeat* message it is also the their most sent message. Comparing the number of *HeartBeat* and *UpdateSuccessor* messages, allow us to confirm our intentions during the implementation process, ensuring that all nodes in a cell share the most up-to-date state, with subsequential neighbors sharing a decreased probability of receiving an up-to-date view. As each *HeartBeat* message will trigger multiple *Ping* messages and comparing with the results shown in Figure 5.2, it is possible to observe that the *Ping* and *PingAck* messages, although the most numerous are also the smallest.

As our objective we strived for a resilient and easy to maintain *Upkeep Routine*, as a result the largest messages tend to be less frequent during the execution of our system.

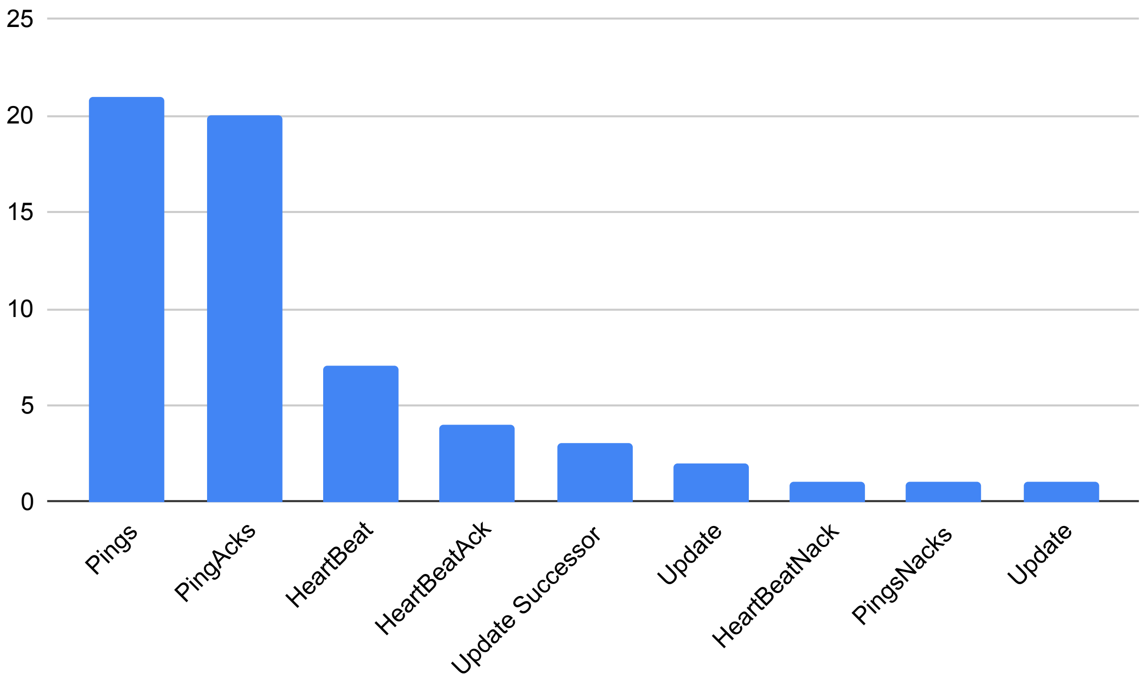

Figure 5.2: Relative comparison of the frequency of each message type.

5.3.2 Split test

Having layed down our system behavior without interruptions, we are in conditions to introduce variability to our tests. Being the ability for nodes to join in the middle of the test execution the most meaningful change introduced in this section. As nodes join the network we are expecting to observe cells reaching its full capacity, thus allowing for the *Split Protocol* to take action. Additionally in this section we still contemplate different frequencies of the *Upkeep Routine*.

We have modified the previous test scenario to better fit the objectives of this test. For this scenario we had 150 nodes join the network, 25 nodes at the time with a tenth of a second delay in between. Then we let the network to stabilize for another 60 seconds. At the end of the stabilization period, 50 nodes joined the network using random nodes already in the system as contact point.

This scenario was introduced with the ultimate goal of understand the consequences of having a node joining out system during its execution. Thus through this section we will seek to study the time it takes for node to join the network until it is considered an active element of a cell. Additionally we sought to gather enough data to understand how the system behaves when executing the *Split Protocol*, and how much time is needed to recover normal functioning after a large exodus of nodes.

In Table 5.2 we have displayed the average time nodes take to converge varying the size of data within. After any dynamic loading operation, nodes make use of *HearBeat* messages in order to synchronize the new state with their peers, we defined that a cell has converged, when all nodes agree on the same state.

Table 5.2: Average conversion time (in seconds) after a split.

HeartBeat Timer	**No data**	**5 KB**	**500 KB**	**1 MB**
5 seconds	10.8	10.4	9.8	10.7
10 seconds	20.2	19.8	19.7	20.5
20 seconds	41.2	41.7	39.2	40.9

Additionally we run a second repetition of every test, to introduce the presence of data in the cell. This round of tests have a an additional step while preparing the network. Instead of having nodes join after a stabilization period of 60 seconds, when this period ends, nodes will now beginning to insert data in the network, only after another 60 seconds, the new nodes will join the system and trigger the *Split Protocol*. With this tests, we believe we can gather enough data, to understand how the data inside the cell affects the split operation.

First, to understand the consequence's of a node joining the network, we measured the time taken since a node joins the system until it becomes an active member of the cell. For this scenario, we tested different cell sizes and our results are displayed in Figure 5.3. Due to the random nature of the upkeep routine, the time for all members of a cell to take notice of a new node deviates from the theoretical minimum, as each *Upkeep Routine* round nodes will target another random node in the cell to update, the Figure 5.3, show us that this random update takes a few rounds until all nodes have been targeted and updated.

During the executions of the tests we have measured the time that cells are marked as *Splitting* after nodes joining the system. Then we have displayed conversion time in Table 5.2. Furthermore we have measured the size of the *SplitRequest* message trough the tests, and the results are shown in Table 5.3.

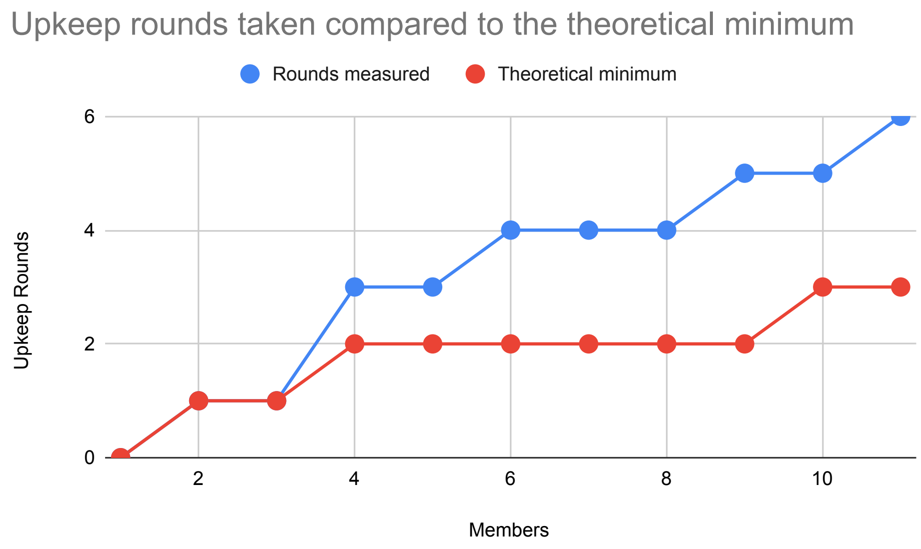

Figure 5.3: Average upkeep routine rounds for all members in a cell to take notice that a new node joined, compared against the absolute minimum.

Observing the data present in Table 5.2 takes us to conclude that the *Split* protocol is mostly independent from the amount of data stored in the cell. Since during the split only half of the keys of the data are shared, the real impact of data during the *Split* operation becomes apparent when looking at Table 5.3, were is shown that the *SplitRequest* message's size is about 10% of the size of data in the cell.

Table 5.3: Average *SplitRequest* message size (in bytes).

No data	**5 KB**	**500 KB**	**1 MB**
832	4041	64189	120253

5.3.3 Merge test

After studying the impact of data and nodes joining the network, we are now in position to study the system behavior when nodes begin to leave. For this test we took a contrary approach to the one taken in the previous section. In this scenario we boot up the network with 200 nodes, similarly to the scenario described in Section 5.3.1, then after 60 seconds of stable execution we had 50 random nodes to leave the system, thus causing the *Merge* procedure to trigger. Again this test was run against all variations previously established. We want to study the impact of the frequency of the *Upkeep Routine* in the *Merge* process, as well as monitor how the data inside the cells is taken into account.

As seen in Table 5.4 a *Merge* operation takes substantially longer to recover than a *Split* as seen in Table 5.2. Backed by the Table 5.5, we can conclude that the *Merge* conversion happens in two phases. On the first phase, the nodes leave the network. As nodes leave silently, only after another node tries to communicate with it via an *HeartBeat* message,

Table 5.4: Average conversion time (in seconds) after a merge.

HeartBeat Timer	No data	5 KB	500 KB	1 MB
5 seconds	10.2	10.4	25.4	26.3
10 seconds	18.4	19.3	40.4	39.8
20 seconds	40.1	41.7	80.5	81.4

it is detected that the node as left the cell. Then this information has to be spread among all members of the cell. To accurately demonstrate the time took by all members of the cell to know that a node has left the network, we have compiled the results of our studies in Table 5.5.

Table 5.5: Average upkeep routine rounds to detect a node left the cell.

Members in Cell	1	2	3	4	5	6	7	8	9	10	11
Upkeep Rounds	2	2	2	4	5	5	5	6	6	7	8

Comparing the average upkeep rounds for all members to take note about a node leaving against the theoretical minimum became a trivial task, as the the best possible case, would be 1 *Upkeep Routine* round. In this hypothetical scenario, after a node left the cell, on the next *Upkeep Routine* round, all other nodes would target it to receive an *HeartBeat* message. Since the node would issue no reply, all other nodes would mark it as *Left Recently* within one round of the *Upkeep Routine*.

Although the average times to spread the information through the cell is stapled in Table 5.5, in our tests we found that the *Merge* operation would start before the state of the cell actually converged. As long as the cell leader detects a low member count, it will initiate the *Merge* procedure by communicating the cell leader in a neighboring cell, thus starting the second phase of the *merge* process.

At this state the cell's leaders broadcast to all members of both cells the intent to converge both cell's states. Finally until the end of the conversion period, all nodes send *HeartBeat* messages at double the rate, during each *HeartBeat* reception, if the state of the sender differs from the state of the receiver, the receiver will attempt to send all data stored, using a force *UpdatePeer* message. This message is proportional to the size of the data in the cell.

Unlike the *Split* process where only the keys of the data travel the network, and half of the data is "forgotten" by the resulting cells, the nature of the *Merge* process does not allow for a light state synchronization. A merging cell does not share any previous knowledge of its counterpart's data, thus, after the two cells exchange memberships, the new cell makes use of the *HeartBeat* message to converge its state. The time of conversion is influenced by the frequency of the *Upkeep Routine* as shown in 5.4, however during that time, nodes will transfer data as shown in Figure 5.4.

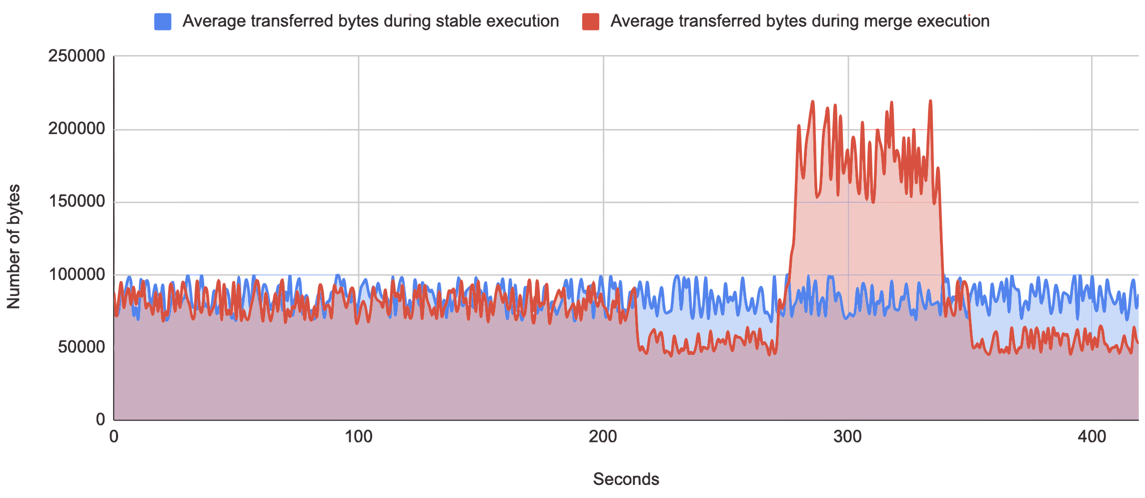

Figure 5.4: Comparison of bytes transferred between a stable execution and an execution when cells need to merge. In both executions, all cells contain 500KB of data.

5.3.4 Relocation test

As predicted in our implementation, the *Merge* operation as revealed to be very communication intensive when cells are responsible for larger amounts of data. The following tests will be performed on the *Relocation Protocol*, with the objective to study its behavior. Furthermore we have done extensive comparisons with the tests perform in the previous section. Our goal is to discover cases where the *Relocation* of nodes may be preferred to the merging of cells.

The scenario implemented for this test was similar the one described in the previous section. We had the network to boot with 200 nodes, then after 60 seconds of stable execution we had 50 random nodes to leave the system, however unlike the previous section, we privileged the use of *Relocations* instead of the *Merge* operation when dealing with node departures.

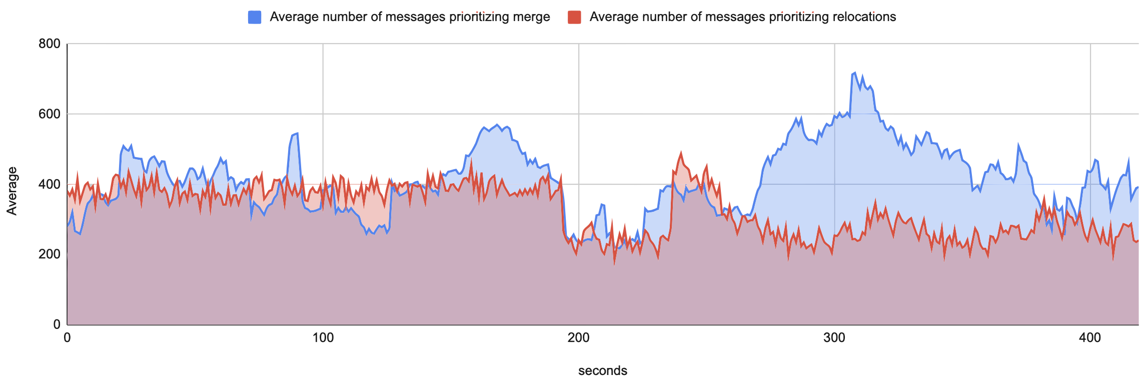

Figure 5.5: Comparison of number of messages sent each second, between an execution where the *Merge* operation is privileged and a case where cells prefer to relocate nodes.

Right from the start, we noted the increasing number of *JoinCellRequest* messages, as the referred messages take part in the *Relocation* process. We measured the *JoinCellReply* resulting from this process, and as expected, the size of the message is proportional to the size of the data contained in the cell. For these tests, we chose to execute the operations when cells contain $1KB$ of data stored, and the results of these tests are presented in Graphs 5.5 and 5.6.

First by analyzing the results shown in Figure 5.5, where we displayed the number of the messages sent each second and their variation when both strategies take action, we can start making our conclusions. We are faced with a scenario, where relocating nodes allows for less messages to be sent during the period of stabilization. Furthermore after the node is relocated to a cell with low members count, meaning that will take less rounds in the *Upkeep Routine* to become a full member if the cell.

By analyzing the Figure 5.6 we can construct better conclusions. Here we are facing a situation, where the *Relocating* nodes requires a smaller window for synchronization within the new cell. During this window we face a period where the amount of data transferred reaches almost the double of the data transferred via *Merge* operations. However when no data is present in the cell, this spike in data transfer does not occur, making the *Relocation* the preferred method.

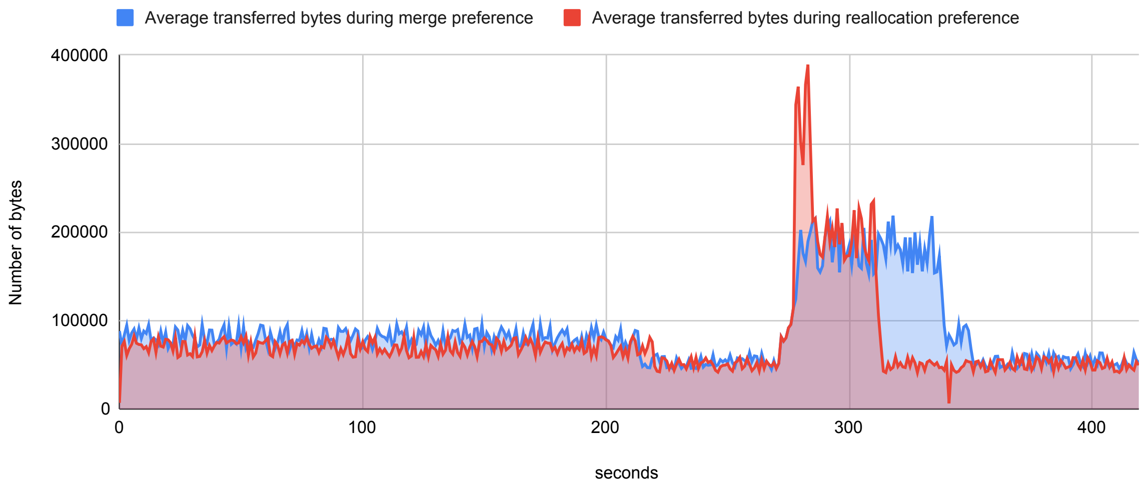

Figure 5.6: Comparison of number of bytes sent each second, between an execution where the *Merge* operation is privileged and a case where cells prefer to relocate nodes.

6 Conclusions

6.1 Conclusion

In this work, we have presented our approach to upgrade Thyme system with a *group-based distributed hash table*, allowing for devices running our system to be able to form a network without being restricted by the physical distance to said devices.

As seen in Figure 4.4 we have augmented the previous Thyme cells. In past iterations, these cells were stateless and statically created on the network boot. The presented solution allow nodes to be created without any configuration from the users. Furthermore, the creation, deletion and management of cells are now in the responsibility of the system, and the system is able to manage cells and their membership in order to better react to the changes of the network.

We argue that the results presented and studied in Chapter 5 allow us to confirm our intentions. And concluded that we have indeed created a network capable of withstanding high levels of churn, while ensuring fairness to all members of the network.

6.2 Future Work

Even though we have developed a DHT capable of reacting to changes in the system membership in running time. Thus completing our initial goals. However, we believe that, while the objectives of this work were achieved, there are still some areas that could be improved as well as completely new features

Cell dynamic loading based on data. The proposed implementation augments Thyme's cell to react accordingly to the number of members present with the goal of preserving as much data as possible. However we think that our implementation would be greatly improved if the decision making process would also take into consideration the data inside the cell and it's popularity. As at the moment of this writing cells may be overloaded with requests and will not take any action to mitigate this condition if the membership inside the cell is stable.

Study the impact of increase memory usage to reduce messages size. Although in this work we strived to create a system capable of synchronization without exchanging large amounts of data. We still believe that some performance upgrades are worth studying. The most pressing focus on expanding the storage catalog, in order to create an extensive log of data items joining and leaving the storage component. We believe that the described process can result in less data travelling the network when synchronizing out-of-date nodes, as by comparing the catalog of two nodes, we can order the operations performed on data, allowing for a finer grain, when updating nodes.

Integration with Thyme GardenBed. Through the course of this work we have implemented the proposed solution into Thyme. However as presented in Section 3.2 and 3.3, Thyme is augmented to use the edge servers of GardenBed. This fusion allows for Thyme to increase data availability across many regions. However we do ask how would our implementation benefit from having an edger server to coordinate cells operations.

Bibliography

[1] S. A. Abid, M. Othman, and N. Shah. “A survey on DHT-based routing for large-scale mobile ad hoc networks”. In: *ACM Computing Surveys (CSUR)* 47.2 (2014), pp. 1–46 (cit. on p. 3).

[2] E. Adar and B. A. Huberman. “Free riding on Gnutella”. In: (2000) (cit. on pp. 7, 8).

[3] E. Anceaume et al. “Peercube: A hypercube-based p2p overlay robust against collusion and churn”. In: *2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE. 2008, pp. 15–24 (cit. on pp. 9–11).

[4] S. Bianchi et al. “Adaptive Load Balancing for DHT Lookups”. In: *Proceedings of 15th International Conference on Computer Communications and Networks*. 2006, pp. 411–418 (cit. on p. 14).

[5] C. Blake and R. Rodrigues. “High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two.” In: *HotOS*. Vol. 3. 2003, p. 1 (cit. on p. 13).

[6] M. Castro et al. *Proximity Neighbor Selection in Tree-Based Structured Peer-to-Peer Overlays*. Tech. rep. MSR-TR-2003-52. 2003, p. 11. URL: https://www.microsoft.com/en-us/research/publication/proximity-neighbor-selection-in-tree-based-structured-peer-to-peer-overlays/ (cit. on p. 11).

[7] F. Cerqueira. “Um Sistema Publicador/Subscritor com Persistência de Dados para Redes de Dispositivos Móveis”. MA thesis. FCT NOVA, Sept. 2017. URL: http://hdl.handle.net/10362/28553 (cit. on pp. 37, 38).

[8] F. Cerqueira et al. “Towards a persistent publish/subscribe system for networks of mobile devices”. In: *Proceedings of the 2nd Workshop on Middleware for Edge Clouds & Cloudlets, MECC@Middleware 2017, Las Vegas, NV, USA, December 11 - 15, 2017*. ACM, 2017, 2:1–2:6. DOI: 10.1145/3152360.3152362. URL: https://doi.org/10.1145/3152360.3152362 (cit. on pp. 4, 16).

[9] Cisco. *Cisco Annual Internet Report (2018–2023) White Paper*. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html (cit. on pp. 2, 3).

[10] P. Costa et al. “Epidemic algorithms for reliable content-based publish-subscribe: An evaluation”. In: *24th International Conference on Distributed Computing Systems, 2004. Proceedings.* IEEE. 2004, pp. 552–561 (cit. on p. 8).

[11] I. B. Damgård. “A design principle for hash functions”. In: *Conference on the Theory and Application of Cryptology*. Springer. 1989, pp. 416–427 (cit. on p. 11).

[12] K. Dolui and S. K. Datta. “Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing”. In: *2017 Global Internet of Things Summit (GIoTS)*. 2017, pp. 1–6. DOI: 10.1109/GIOTS.2017.8016213 (cit. on p. 2).

[13] U. Drolia et al. “Krowd: A Key-Value Store for Crowded Venues”. In: MobiArch ’15. Paris, France: Association for Computing Machinery, 2015, pp. 20–25. ISBN: 9781450336956. DOI: 10.1145/2795381.2795388. URL: https://doi.org/10.1145/2795381.2795388 (cit. on p. 3).

[14] A. Elgazar et al. “Towards Intelligent Edge Storage Management: Determining and Predicting Mobile File Popularity”. In: *2018 6th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*. 2018, pp. 23–28 (cit. on p. 14).

[15] M. J. Fischer, N. A. Lynch, and M. Paterson. “Impossibility of Distributed Consensus with One Faulty Process”. In: *J. ACM* 32.2 (1985), pp. 374–382. DOI: 10.1145/3149.214121. URL: https://doi.org/10.1145/3149.214121 (cit. on p. 6).

[16] L. Glendenning et al. “Scalable consistency in Scatter”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011, pp. 15–28 (cit. on pp. 13, 14).

[17] K. T. Greenfeld and K. Taro. “Meet the napster”. In: *Time Magazine* 2 (2000), pp. 998068–1 (cit. on p. 7).

[18] B. Hayes. *Cloud computing*. 2008 (cit. on p. 1).

[19] D. R. Karger et al. “Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web”. In: *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*. Ed. by F. T. Leighton and P. W. Shor. ACM, 1997, pp. 654–663. DOI: 10.1145/258533.258660. URL: https://doi.org/10.1145/258533.258660 (cit. on p. 11).

[20] K. Kenthapadi and G. S. Manku. “Decentralized algorithms using both local and random probes for P2P load balancing”. In: *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. 2005, pp. 135–144 (cit. on p. 15).

[21] M. A. Khan et al. “MobiStore: A system for efficient mobile P2P data sharing”. In: *Peer-to-Peer Networking and Applications* 10.4 (2017), pp. 910–924 (cit. on p. 10).

[22] J. Ledlie and M. Seltzer. "Distributed, secure load balancing with skew, heterogeneity and churn". In: *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.* Vol. 2. IEEE. 2005, pp. 1419–1430 (cit. on p. 15).

[23] S. Lee and P. Narasimhan, eds. *Software Technologies for Embedded and Ubiquitous Systems, 7th IFIP WG 10.2 International Workshop, SEUS 2009, Newport Beach, CA, USA, November 16-18, 2009, Proceedings*. Vol. 5860. Lecture Notes in Computer Science. Springer, 2009. ISBN: 978-3-642-10264-6. DOI: 10.1007/978-3-642-10265-3. URL: https://doi.org/10.1007/978-3-642-10265-3 (cit. on p. 41).

[24] J. C. A. Leitão and L. E. T. Rodrigues. "Overnesia: A Resilient Overlay Network for Virtual Super-Peers". In: *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*. 2014, pp. 281–290 (cit. on p. 8).

[25] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[26] Y. Mao et al. "A Survey on Mobile Edge Computing: The Communication Perspective". In: *IEEE Communications Surveys Tutorials* 19.4 (2017), pp. 2322–2358. DOI: 10.1109/COMST.2017.2745201 (cit. on p. 3).

[27] P. Maymounkov and D. Mazières. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". In: *Peer-to-Peer Systems, First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002, Revised Papers*. Ed. by P. Druschel, M. F. Kaashoek, and A. I. T. Rowstron. Vol. 2429. Lecture Notes in Computer Science. Springer, 2002, pp. 53–65. DOI: 10.1007/3-540-45748-8_5. URL: https://doi.org/10.1007/3-540-45748-8%5C_5 (cit. on pp. 9, 10).

[28] D. S. Milojicic et al. "Peer-to-peer computing". In: Technical Report HPL-2002-57, HP Labs, 2002 (cit. on p. 7).

[29] J. Paiva, J. Leitão, and L. E. T. Rodrigues. "Rollerchain: A DHT for Efficient Replication". In: *2013 IEEE 12th International Symposium on Network Computing and Applications, Cambridge, MA, USA, August 22-24, 2013*. IEEE Computer Society, 2013, pp. 17–24. DOI: 10.1109/NCA.2013.29. URL: https://doi.org/10.1109/NCA.2013.29 (cit. on pp. 9, 12).

[30] Y. Qiao and F. E. Bustamante. "Structured and unstructured overlays under the microscope". In: USENIX. 2006 (cit. on p. 8).

[31] S. Ratnasamy et al. "A scalable content-addressable network". In: *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 27-31, 2001, San Diego, CA, USA*. Ed. by R. L. Cruz and G. Varghese. ACM, 2001, pp. 161–172. DOI: 10.1145/383059.383072. URL: https://doi.org/10.1145/383059.383072 (cit. on pp. 9, 10).

[32] H. B. Ribeiro and E. Anceaume. “Datacube: A p2p persistent data storage architecture based on hybrid redundancy schema”. In: *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE. 2010, pp. 302–306 (cit. on p. 13).

[33] A. I. T. Rowstron and P. Druschel. “Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems”. In: *Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12-16, 2001, Proceedings*. Ed. by R. Guerraoui. Vol. 2218. Lecture Notes in Computer Science. Springer, 2001, pp. 329–350. DOI: 10.1007/3-540-45518-3_18. URL: https://doi.org/10.1007/3-540-45518-3%5C_18 (cit. on pp. 9, 10).

[34] Y. Saad and M. H. Schultz. “Topological properties of hypercubes”. In: *IEEE Transactions on Computers* 37.7 (1988), pp. 867–872 (cit. on p. 10).

[35] *A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications*. 2001, pp. 101–102. DOI: 10.1109/P2P.2001.990434 (cit. on p. 7).

[36] J. A. Silva, P. Vieira, and H. Paulino. “Data Storage and Sharing for Mobile Devices in Multi-region Edge Networks”. In: *2020 IEEE 21st International Symposium on "A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*. 2020, pp. 40–49. DOI: 10.1109/WoWMoM49955.2020.00021 (cit. on pp. 4, 5, 19).

[37] J. A. Silva et al. “Ephemeral Data Storage for Networks of Hand-Held Devices”. In: *2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, August 23-26, 2016*. IEEE, 2016, pp. 1106–1113. DOI: 10.1109/TrustCom.2016.0182. URL: https://doi.org/10.1109/TrustCom.2016.0182 (cit. on p. 3).

[38] J. A. Silva et al. “It’s about Thyme: On the design and implementation of a time-aware reactive storage system for pervasive edge computing environments”. In: *Future Gener. Comput. Syst.* 118 (2021), pp. 14–36. DOI: 10.1016/j.future.2020.12.008. URL: https://doi.org/10.1016/j.future.2020.12.008 (cit. on pp. 4, 16).

[39] J. A. Silva et al. “Time-aware reactive storage in wireless edge environments”. In: *MobiQuitous 2019, Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, Houston, Texas, USA, November 12-14, 2019*. Ed. by H. V. Poor et al. ACM, 2019, pp. 238–247. DOI: 10.1145/3360774.3360828. URL: https://doi.org/10.1145/3360774.3360828 (cit. on pp. 4, 16).

[40] J. A. A. e Silva. “Data Storage and Dissemination in Pervasive Edge Computing Environments”. PhD thesis. NOVA University of Lisbon, 2021 (cit. on p. 5).

[41] M. Steiner, T. En-Najjary, and E. W. Biersack. “Long Term Study of Peer Behavior in the kad DHT”. In: *IEEE/ACM Transactions on Networking* 17.5 (2009), pp. 1371–1384 (cit. on pp. 3, 10).

[42] I. Stoica et al. “Chord: A scalable peer-to-peer lookup service for internet applications”. In: *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 27-31, 2001, San Diego, CA, USA*. Ed. by R. L. Cruz and G. Varghese. ACM, 2001, pp. 149–160. DOI: 10.1145/383059.383071. URL: https://doi.org/10.1145/383059.383071 (cit. on pp. 8–10, 15, 23, 31).

[43] D. Stutzbach and R. Rejaie. “Understanding Churn in Peer-to-Peer Networks”. In: *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*. IMC ’06. Rio de Janeriro, Brazil: Association for Computing Machinery, 2006, pp. 189–202. ISBN: 1595935614. DOI: 10.1145/1177080.1177105. URL: https://doi.org/10.1145/1177080.1177105 (cit. on p. 7).

[44] T. Taleb et al. “Mobile Edge Computing Potential in Making Cities Smarter”. In: *IEEE Communications Magazine* 55.3 (2017), pp. 38–43. DOI: 10.1109/MCOM.2017.1600249CM (cit. on p. 3).

[45] P. Vieira. “Data Storage and Sharing for Mobile Devices in Multi-region Edge Networks”. MA thesis. NOVA University of Lisbon, 2019 (cit. on pp. 19, 20, 37).

[46] S. Wang et al. “Edge server placement in mobile edge computing”. In: *Journal of Parallel and Distributed Computing* 127 (2019), pp. 160–168. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2018.06.008. URL: https://www.sciencedirect.com/science/article/pii/S0743731518304398 (cit. on p. 3).

[47] X. Wang and D. Loguinov. “Load-balancing performance of consistent hashing: asymptotic analysis of random node join”. In: *IEEE/ACM Trans. Netw.* 15.4 (2007), pp. 892–905. DOI: 10.1145/1295257.1295270. URL: http://doi.acm.org/10.1145/1295257.1295270 (cit. on p. 15).

[48] T. Zahn and J. Schiller. “DHT-based unicast for mobile ad hoc networks”. In: *Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW’06)*. IEEE. 2006, 5–pp (cit. on p. 3).

[49] T. Zahn and J. Schiller. “MADPastry: A DHT substrate for practicably sized MANETs”. In: *Proc. of ASWN*. 2005 (cit. on p. 3).

[50] B. Y. Zhao et al. “Tapestry: a resilient global-scale overlay for service deployment”. In: *IEEE Journal on Selected Areas in Communications* 22.1 (2004), pp. 41–53. DOI: 10.1109/JSAC.2003.818784. URL: https://doi.org/10.1109/JSAC.2003.818784 (cit. on pp. 9, 10).

[51] B. Zhao, Y. Wen, and H. Zhao. “KDSR: An Efficient DHT-Based Routing Protocol for Mobile Ad Hoc Networks”. In: *2009 Ninth International Conference on Hybrid Intelligent Systems*. Vol. 2. 2009, pp. 245–249. DOI: 10.1109/HIS.2009.160 (cit. on p. 3).

2021 Managing Population and Workload Imbalance in Structured Overlays Ricardo Leitão

NOVA SCHOOL OF SCIENCE & TECHNOLOGY