Francisco Martins Magalhães

Licenciado em Engenharia Informática

Membership Service for Large-Scale Edge Systems

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: João Carlos Antunes Leitão, Auxiliary Professor, NOVA University of Lisbon

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2018

Membership Service for Large-Scale Edge Systems

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To my Erasmus friends.

Acknowledgements

Firstly, I would like to express my gratitude to my adviser, Prof. Dr. João Leitão, for providing me with the opportunity to embrace this challenge in the well-known and highly discussed area of Distributed Systems. I also want to thank Albert Linde, who, despite not having a direct impact in this work, influenced my interest in this area early on, together with Prof. Dr. João Leitão.

I thank my colleagues that in some way contributed to this work. In particular, I would like to thank Pedro Lopes, André Rodrigues, Pedro Fouto and Gonçalo Nunes, for supporting me with reviews and suggestions.

I also thank my family and friends who provided me with the necessary motivation to keep this project going.

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

Abstract

Distributed systems are increasing in scale and complexity. Cloud-based applications need more resources to be able to continue satisfying the user requirements. There are more users and devices with Internet access and therefore, more data to process and store. There is now the need of having additional systems closer to clients (and potentially outside of the data center), in the edge. Many distributed systems would be able to benefit from this new Edge Computing paradigm, where some of the computations that are done in data centers now, are instead performed closer to the client on edge nodes or edge devices.

In order to keep managing and building these services, there is the need of a membership service, which is a fundamental component in a distributed system. A membership service must be able to track all the correct processes in a system, as to avoid routing requests to unreachable nodes, to balance the workload in the system, etc.

In this work, we have designed a novel, scalable, and efficient membership service that is able to support large-scale applications which include components in the edge of the network, closer to the users. We present a membership service based on partial views on each node, so as to increase scalability. Furthermore, this work presents an edge membership component that is used to keep track of edge nodes in the system and a dissemination component to broadcast information between them and also to data center nodes.

Additionally, we present an API with well-defined functions that allow managing a membership with features including edge functionalities.

Keywords: Edge Computing, Cloud, Large-scale Membership Service, Distributed Storage Systems

Resumo

Os sistemas distribuídos têm vindo a aumentar em escala e complexidade. As aplicações baseadas na Nuvem precisam de mais recursos a fim de ser capazes de continuar a satisfazer as necessidades do utilizador. Existem agora mais utilizadores e dispositivos com acesso à Internet, e por conseguinte, mais dados para processar e armazenar. Existe agora a necessidade de ter sistemas adicionais mais perto dos clientes (e potencialmente fora dos centros de dados). Muitos sistemas distribuídos seriam capazes de beneficiar deste novo paradigma da Computação na Berma, onde algumas das computações feitas nos centros de dados podem passar a ser feitas nos nós ou dispositivos na berma da rede.

A fim de se poder continuar a construir e gerir estes serviços, é necessário um serviço de membership, um componente fundamental nos sistemas distribuídos. Um serviço de filiação deve ser capaz de detetar todos os processos corretos de um sistema, para evitar fazer *routing* de pedidos para nós inalcançáveis, para equilibrar o *workload* no sistema, etc.

Nesta tese, apresentamos um serviço de filiação novo, escalável e eficiente, capaz de suportar aplicações de grande escala com componentes na periferia da rede, mais perto dos utilizadores destes serviços. Este serviço de filiação é baseado em vistas parciais em cada nó, tornando-o mais escalável. Para além disso, apresentamos um componente de berma usado para manter informação de nós na berma e disseminar informação entre eles e também para os nós nos centros de dados.

Em adição, apresentamos uma API com funções bem definidas que permitem a gestão de uma membership com funcionalidades que envolvem a berma da rede.

Palavras-chave: Computação na Berma, Nuvem, Serviço de Membership de Grande Escala, Sistemas de Armazenamento Distribuídos

Contents

CHAPTER 1

Introduction

1.1 Context

As the Internet is becoming more and more accessible around the world, the number of users of systems and applications naturally tend to increase. Nowadays, worldwide users use the Internet on a daily basis and more frequently than ever. Services accessed through the Internet have become indispensable for the everyday life, from social interactions to commerce and even business. For this reason, there is an ever increasing need for availability and fast response, since users are becoming less tolerant to delays in response times. Furthermore, systems need to store big amounts of data and replicate such data in order to keep it available for users worldwide even in scenarios where catastrophic failures might happen.

The paradigm of Cloud Computing has been around for some time. It refers not only to both the applications delivered as services, but also to the hardware and systems software in the data centers. Cloud computing has had a significant impact in the way we live and work and in particular how we build and maintain large-scale systems. Many large-scale systems were deployed in recent years to handle the increasing amount of data that is now commonly manipulated by applications and services used around the world. A significant fraction of those applications rely on Cloud-based infrastructures to be able to process and manage user and application data in a scalable, consistent, persistent, and available way.

Millions, or even billions, of users have now access to those services, that include Facebook, Amazon, or Netflix. Using the Cloud computing paradigm, multiple data centers are interconnected across the globe and client applications frequently interact with remote servers deployed across those data centers. These interactions are used to process data that is generated by edge devices (such as smartphones, tablets, and even Internet of Things (IoT) devices).

However, the Cloud cannot scale indefinitely. In spite of the large number of advantages brought by cloud computing, as noted before (such as the ability to deploy highly available and scalable applications), this approach can have its disadvantages, namely in terms of high latency for transferring large amounts of data from clients to the cloud infra-structure and vice-versa. Additionally, as the system grows and more resources are required, the costs for service providers also increase. Furthermore, the need for frequent interaction between users and remote servers causes an increase in the perceived latency for the communication and data processing, which is not good for applications that require real-time response.

These drawbacks are exacerbated by the emergence of a new computing paradigm, motivated by the proliferation of the Internet of Things (IoT) domain: Edge Computing. Since information is nowadays mostly being produced at the edge of the network, processing the data at the edge can be more efficient. This concept is based on the approach of moving some of the computational load from the cloud towards the edge of the network and therefore closer to the source of data. Intermediate computations can be performed in multiple devices such as routers, switches, and even tablets, smartphones, wearables, or desktops, and laptops.

Many distributed systems would be able to benefit from this new paradigm of Edge Computing. In order to do so, there is the need of some additional (computational/storage) systems closer to clients (to relocate some of their components) and potentially outside of the data center. The key aspect in managing and building these services are membership services, that track the components that are currently active and available in the system. Many membership services were designed in the last years as to support large-scale systems [17, 32]. However, there is the need to devise highly scalable and efficient membership services that are capable of addressing the challenges placed by systems that have their components, potentially different among themselves, spread across multiple data centers and potentially, multiple edge locations.

1.2 Motivation

With computations and key support services moving towards the edge, large-scale distributed systems also need to suffer some changes when it comes to managing the system membership. One of the key aspects in this change is the membership service. A membership service needs to be able to track all system components throughout the system.

Current membership service solutions are designed for applications that rely on Cloud-based infrastructure, such as Cassandra. Even though they have proven to be scalable and failure resilient, membership services need to be changed in order to provide more adequate support for the Edge Computing paradigm. For instance, they have to deal with the fact that many nodes are behind NAT devices and firewalls, as they provide security features. For this reason, membership services allowing edge features must consider the existence of these components, otherwise connection requests to private nodes will be denied.

Large-scale distributed storage systems, such as Cassandra, are a key component in many large-scale systems nowadays. Cassandra's membership service, similar to many other distributed systems, requires each node to know each other in the system while keeping an open (TCP) connection to each so as to fasten communication between them. Many approaches for membership services were deployed in large-scale distributed systems in a way that nodes need to know only a fraction of active nodes. Such approaches can be much more efficient in terms of resource management and information dissemination. However, current implementations do not deal with the notion of multiple data centers. This makes it harder to disseminate information in a fast and reliable way between nodes in different data centers. Moreover, extending such solutions to deal with large number of nodes scattered in different edge locations is an open challenge.

Clients of these large-scale systems would benefit from having such information close to them. Edge nodes, such as personal computers located in private networks, for example, inside a company office, could act as servers which belong to the system's membership. In that sense, such an edge node would contain a small part of the information existing in the system's data centers. Clients which require reads and writes only to that small part of the data could benefit from connecting to those edge server nodes instead of establishing connection to data center nodes, therefore allowing faster access. An example of such a system would be a system containing lots of data regarding the students of a given university since it was inaugurated. One or more data centers would contain all the data. Furthermore, in private networks such as a specific department network, edge nodes, such as those in a department's cluster or even some personal computers, would store information specific of that same department. Students or professors inside that same LAN would be able to quickly establish connections to one of those nodes and access all the data, concerning grades, payments, etc. Any change would then be propagated to the data centers.

1.3 Contributions

In this work, we present a membership service designed specifically to support distributed systems that run in devices located both in data centers and on the edge of the system.

Besides that, we have used an adaptation of an already existent membership gossip-based broadcast protocol that deals with the notion of different data centers when performing data dissemination. This solution is able to ease the communication between different data centers so the dissemination of information becomes faster. A fraction of nodes of different data centers know each other and therefore are included in each other's partial views of the system, in a way that information can be disseminated in a timely and efficient manner both between different data centers and inside a data center. Our solution also provides an efficient and fast broadcast mechanism to share information between nodes, such as node's state and application data.

Our solution includes an edge component that allows devices in private networks to act as membership nodes and retain a part of the information shared across the membership. Those edge nodes can be helpful to accelerate the communication between clients and the system, by making information available closer to them. Devices such as personal computers or server nodes inside a private network can then be used as part of the membership and eventually serve client connections and operations if the application demands so.

We also present a Java API with well-defined methods that allow to manage cluster nodes and share information between them. Additionally, we have devised a simple Twitter-based application, that relies on edge networks, that can make use of our membership service, through the API we provide. This application is used to demonstrate the benefits of our solution.

1.4 Document organization

The remainder of this document is organized as follows:

Chapter 2 describes the concepts, definitions, and existing approaches to underlying problems related to membership services and edge systems. It also discusses some existing solutions to improve the scalability of the distributed membership services.

Chapter 3 presents the work done in this thesis, which includes a detailed description of our solution's architecture as well as the underlying implementation, adapted from Cassandra's membership. Furthermore, it describes the API we provide to be used by cloud and edge applications.

Chapter 4 presents an evaluation of our membership's components, as well as a discussion of the experimental results.

Finally, **Chapter 5** includes a discussion of possible improvements and future work for the membership service, as well as a conclusion of the main topics and contributions of this work.

Chapter 2

Related Work

This chapter presents a survey on relevant previous work that can be leveraged to build novel and improved membership services for large-scale distributed systems.

We start this chapter by briefly discussing the edge computing paradigm, that in part motivates the need for building large and more complex systems. We then discuss distributed membership services which are the focus of this work.

This is followed by a discussion on related work on peer-to-peer systems, which provides foundations for building decentralized membership services. Additionally, we discuss Cassandra's membership, as an example of membership services implemented in large-scale distributed storage systems. We also discuss NAT-Aware memberships, as well as existing membership services.

2.1 Nodes and peers

For better understanding of this chapter and subsequent ones, we clarify our usage of the terms *node* and *peer*. We refer to *nodes* and *peers* the same way we refer to *people* and *friends*. Friends are obviously people, although two people are not necessarily friends, as well as two nodes are not necessarily peers. When a node joins a system, it is simply a node. However, being a peer (of another node) requires some relationship. When two nodes become aware of each other and agree to share information about the membership, they become peers.

2.2 Edge-Computing

The new era of the Internet of Things brings new mobile equipment, smart devices, sensors, vehicles, i.e. edge devices, which generate large amounts of data streams. There is the need to gather additional computational power, as well as improved connectivity among all the devices. The concept of edge computing has emerged to denote a new trend in large-scale distributed systems, where computations have to be performed outside the data center boundaries, closer to the data sources and consumers (i.e. clients).

The tradicional approach of having data centers far from the users is becoming a limitation. Because of this evolution, it is necessary to have computational components of applications closer to the devices. Thus, edge computing is used to perform data processing at the edge of the network, improving and overcoming some pitfalls of cloud computing. In other words, the goal of edge computing is to search for ways of making computations on edge nodes, i.e, the nodes through which application network traffic is directed. The work presented in [42] defines five main needs motivating computing on edge nodes: Decentralized cloud and low latency computing, surmounting resource limitations, energy consumption, dealing with data explosion and network traffic, and smart computation techniques.

2.3 Membership

A key aspect in managing and building such large-scale services, that can cope with the needs to materialize the concept of edge computing is the membership service. Sometimes it is important to be aware of the most updated information about the computational resources active and available in a distributed system. We need to know which nodes are active, for example to balance the workload. As we deal with unreliable networks, we should expect and handle network partitions, packet loss, transmission delays, etc. Furthermore, we want to avoid routing requests to unreachable nodes, which would only consume resource with no benefit for the progress of the system. Hence, the membership must be able to track the components of the system. In this context, there are some important concepts to be introduced.

One solution for building a membership service, and in fact the most intuitive one, would be to provide full membership. This implies that every process must know all the other processes in the system. However, a membership service, in order to be scalable, must support a large number of processes. In a large-scale system, we can expect frequent changes in the membership, either due to failures or because we stop needing some processes (and these are decommissioned). The cost of keeping such a large information in each node makes this approach unfeasible. This solution however, has been used in pratical systems. Amazon's Dynamo [11] is highly available and its membership uses a gossip based algorithm that allows every node to know every other node in the system. This creates an overlay in which the distance between two nodes is at most one-hop (as we discuss further ahead).

However, the drawbacks of leveraging full membership protocols become more evident when considering edge-computing systems, due to the fact that such systems might have many more components, potentially very heterogeneous among them in terms of available resources and operations environment.

An alternative to full membership protocols is to use partial membership protocols. In these, each process in a system is aware of only a fraction of other processes. Such approach is inherently more scalable, as the amount of information maintained by each process is much smaller. Furthermore, each process has to keep track of membership changes for only a small portion of the whole system.

Typically, partial membership protocols operate by building and maintaining an independent partial view of the system for each process. In this work we aim to build a highly scalable and efficient partial membership solution. In the following we explain in more detail the concept of partial view.

2.3.1 Partial views

Partial views encode membership information and due to that, they are usually composed by a set of process identifiers. These identifiers contain, at least, the necessary information to reach the process identified by it (e.g. an IP address and TCP or UDP port).

However, these identifiers can also include some additional meta-data that can ease the management of the partial views maintained by each process or some information that is useful to support the operation of systems leveraging the membership service (e.g. in distributed hash tables, which we discuss further ahead, these identifiers usually are enriched with a probabilistic unique large bit string that is relevant to define the DHT topology).

While partial views should be very small with respect to the total size of the system, their concrete size depends on a number of factors, such as the maximum expected number of processes in the system and the expected dynamicity of the system membership. As we discuss further ahead, different protocols for managing these partial views (partial membership protocols) prescribe different sizes for them.

Whenever there is a change in the membership, partial views should be updated to reflect the changes. It can happen that some nodes will suffer no difference in their partial views. If a process enters the system, its identifier should be added to the partial views of some of the other processes. The contrary will happen if a node fails or leaves the system. These changes should be made in a small time window, so as to avoid incorrect operations of the system leveraging the partial membership protocol.

Partial membership protocols, that manage partial views, usually follow one of these strategies:

- **Reactive:** According to this strategy, nodes' partial views change depending on events in the membership. These events (the arrival or departure of new nodes) trigger partial view changes. In other words, if the membership does not change, partial views do not change.

- **Cyclic:** In this approach, on the other hand, partial views change periodically from time to time. Typically, every t time units nodes exchange some information which leads them to update the contents of their partial views.

2.4 Gossip-based protocols

This section addresses gossip-based broadcast protocols and how they are used for building membership services. In spite of that, gossiping can be used for other purposes, such as file sharing systems, application level reliable broadcast, publish-subscribe, etc, which are not part of the scope of this discussion.

The concept of *gossip communication* in computer science is based on real gossiping among humans. Due to the fact that biological virus can be spread in the same way as gossip spreads in human communities, this class of protocols is many times called *epidemic protocols* instead of *gossip protocols*.

In a gossip broadcast protocol, in order to broadcast a message, a node selects t nodes from the system at random and propagates the message to them. When a node receives a message for the first time, it repeats the procedure. Gossip broadcast protocols are used for dissemination of information, such as user data, control information, etc.

The same way it happens in human gossiping, some redundant messages are produced by the dissemination process. This phenomena, while consuming additional network resources, does not affect the correctness of protocols, since information is still spread throughout all nodes in the system. Actually, this redundancy makes this kind of protocols highly resilient to different kinds of failures, such as node and network failures. Less redundancy makes the protocol more susceptible to faults and hence decreases its failure tolerance requirements.

The work present in [27] defines a gossiping framework. The crucial aspects captured by this framework are peer selection, data exchanged, and data processing. Obviously, this aspects can be applied to gossip protocols used for memberships. Lpbcast [14] combined its actual dissemination algorithm with a membership service, being the first protocol to introduce a gossip-based membership service. We can model the gossiped information exchange by means of an active thread which takes the initiative to communicate and a passive thread which accepts requests from other nodes.

In order to broadcast a message, a process selects some other processes in the system at random. In many systems, it is assumed that the set of peers can be chosen uniformly at random. This choice mechanism is called **peer selection**. Gossiping protocols can differ a lot in how they select the peers for each communication step of the protocol.

Peers make decision based on **data exchanged** with each other. Depending on application domains, this data will have different types of information. In a membership protocol, the information exchanged among peers is mainly lists of peers known to be part of the system.

When a node receives new information from a peer, it should deal with the received information (**data processing**) and decide what to do. In membership services, nodes update their lists of neighbors (i.e. their partial views) according to the received list of peers, having into account some restrictions, such as maximum view size.

The number of nodes with whom a node exchanges information in a single gossip communication step is called fanout. Some membership protocols use adaptive fanout, being able to change it on-the-fly, according to the state of the membership. Usually, the fanout is defined as a global system parameter, according to reliability and redundancy.

2.4.1 Strategies

The work presented in [30] identifies four main approaches for the implementation of a pairwise gossip information exchange.

Eager push: According to this approach, as soon as a node receives new information (i.e. a new message) to be propagated, it sends it to random selected peers, according to the fanout. The degree of redundancy and overhead depends in the number of peers selected. However, this allows a fast dissemination. Furthermore, there is no need in keeping copies of the messages, unlike the following discussed approaches.

Pull: Using a periodic approach, a node asks random selected peers if there was any new information received recently. If the node receives a report about new information, it explicitly makes a request (to the peer that replied) for that information. Although this approach reduces redundancy (when information exchanged among peers is large), the time interval used for asking for new messages may affect the speed of the dissemination.

Lazy push: Instead of sending new information immediately as in eager push, nodes send only an unique identifier of that information. If a node receives an identifier that it has not received, it explicitly requests the corresponding information. This approach also reduces the amount of messages disseminated, although it takes more time for a node to get a message than in the eager push approach.

Hybrid: An hybrid approach is a combination of the strategies above. For example, eager push can be used to disseminate the messages for fast delivery. Secondly, lazy push can be used in order to recover from omissions in the first phase, ensuring failure recovery.

2.5 Overlay Network

We call *overlay network* to a logical network that operates on top of another (tipically the physical) network. This network consists on the nodes and the logical neighboring relationships between them. It is usually perceived as a graph in which the processes are the vertices and the edges are communication links. In fact, a partial membership protocol implicity defines an overlay network as the closure of the partial views maintained by each individual process (since partial views define neighboring relationships). In this section, we will aproach three categories of overlays. These categories have been analysed and compared before [5, 37], mainly when it comes to large-scale network overlays.

2.5.1 Structured Overlays

In structured overlays, the topology of the overlay follows a predefined structure, which means that nodes are required to be routed to a specific logical position in the network. This known structure allows improvement on search primitives, enabling efficient discovery of data and processes (usually identified by a probabilistic unique identifier). This is achieved by constraining both the overlay and the placement of data in a way that data objects and nodes have identifiers in the same (logical) range.

However, the advantages of having a predefined topology may come at a big price when it comes to churn [1] resilience. Under these conditions, in which there is a need to stabilize the network, the protocol must somehow handle or prevent an excessive control message overhead and degraded routing performance, due to the lack of flexibility of a structured overlay network, which is usually very hard to achieve in practice. Structured overlay are usually based on DHTs.

2.5.1.1 Distributed Hash Table (DHT)

A DHT is a commonly used distributed data structure in today's P2P systems and multiple large-scale distributed data stores. A DHT stores (key, value) pairs, allowing a lookup feature in the overlay similar to a hash table. In the same way, all possible keys are included in a keyspace, split across the nodes. This abstraction can be shared at the same time by many applications [18].

Rather than having a central node responsible for the mapping of data objects to nodes, any node in the overlay is able to retrieve data based on keys. Each node is responsible to store some section of the keyspace, based on its own identifier and the identifiers of surrounding peers.

2.5.1.2 Chord

Chord is a structured overlay (DHT-based) algorithm based on consistent hashing [25]. Each node is assigned a unique identifier in the space of the output of a hash function. Furthermore, it is implemented in a way that there is no need for updating the partial views of many peers when a node joins or leaves the system. The nodes' identifiers are usually hashes of their IP address.

Chord interprets identifiers as a circular space that is used to organize nodes in a ring. Each node also maintains a routing table with other node references, called the finger table.

[1]Churn is a period during the system activity in which there is an extremely high frequency of changes in the membership, due to arrivals and departures of nodes.

These references are used as shortcuts to navigate the ring topology in fewer numbers of hops in the overlay.

The tables maintained by the nodes are automatically adjusted to changes, ensuring that it's always possible to find a node responsible for a given key (which depends on its own identifier and the identifier of neighboring nodes in the ring). However, simultaneous failures may break the overlay, as Chord's correctness depends on each node knowing its correct successors. To make sure that the lookups are done correctly, Chord runs a stabilization protocol periodically in background, updating successor pointers (as well as entries in the finger table).

2.5.1.3 Pastry

Pastry is similar to Chord, as it also defines a DHT. It is completely decentralized, scalable and self-organizing. What sets Pastry apart from other other DHTs overlays is the routing strategy employed to navigate the overlay network. Node's identifiers are assigned randomly when a node joins, and nodes, similar to Chord, are arranged in a circle. The routing strategy used in Pastry is based on numeric closeness of identifiers. Each Pastry node maintains a routing table, based on substrings of the node own identifier. When trying to route a message, a node first chooses a peer whose ID shares the longest prefix with the target's identifier. If it cannot find a suitable candidate, it will chose a peer whose identifier is numerically closest to the target's.

When a node joins the system, it informs other nodes of its presence. It's assumed that the new node knows about a node already present in the system (called a contact node), which can be located by IP multicast or configured by the system administrator. However, this join protocol is more complex than the one employed by Chord. The routing table of the new node will be filled with information sent from nodes along the overlay path followed by the join message while it is routed to the node with the closest identifier to the joining node, which can lead to an increase in the latency of the join procedure. However, since Pastry takes network locality into account, it minimizes the distance messages travel in the underlay (i.e. the physical network).

Pastry nodes also maintain two other sets of peers, to provide back-up nodes to handle failures in a timely fashion.

2.5.2 Unstructured Overlays

In unstructured overlays, there are few constraints on how neighboring relationships (and hence, communication links) are established among nodes in the system. In order to join the system, a node needs to make a request to a contact node, which is provided by an external mechanism (similar to what is assumed in Pastry). Later, some of the other members will be notified of this change by the contact node by either flooding the overlay or using a random walk mechanism.

The resulting graph is highly randomized and hard to characterize. Not having a predefined topology, the overlay may have high resilience to churn scenarios, providing strong fault-tolerance, which derives from the fact that there are few or no restrictions on the contents of partial views maintained by nodes. On the other hand, in order to make sure a query cover enough peers, unstructured systems have to use flooding or random walk mechanisms. For that reason, this randomness brings a disadvantage on constructing this kind of communication and search primitives.

We now discuss the design of some distributed algorithms that build and maintain unstructured overlay networks.

2.5.2.1 SCAMP

Scamp [17] is a fully decentralized protocol in which each node is provided with a partial view of the membership. Its design requirements include scalability, reliability, decentralized operation, and isolation recovery.

Scamp builds an unstructured overlay, creating partial views without a fixed size, that contain on average $(c+1)*log(n)$ connections, c being a design parameter for fault tolerance. Each node maintains two views of the membership. *PartialView* is a list of nodes with whom the local node performs direct interactions (e.g. gossip exchanges). The *InView* is a list of nodes from which it receives information (i.e. nodes that have the local node in their *PartialView*).

When a node receives a subscription request from a new node joining the system, it forwards the subscription to the peers in its *PartialView*. Furthermore, an additional c copies are sent to random chosen neighbors in the *PartialView*, as a mechanism to boost fault tolerance. If c is too big, it will produce a significant number of redundant message being propagated in the network, which will impact network usage. On the other hand, it should not be too small, otherwise it will not improve fault tolerance at all. When receiving a forward subscription, a node integrates it in its list with a probability depending on the size of its local partial view. If it does not add it to the *PartialView*, it forwards the message to a random chosen neighbor. Otherwise it'll tell the new node to add it in its *InView*.

Scamp also uses a heartbeat mechanism for isolation recovery. A node assumes it is isolated when it has not received a heartbeat message in a long time. In this case, it tries to resubscribe. For this end, it sends a new subscription message through an arbitrary node in its *PartialView*, as if it was trying to join the overlay for the first time.

2.5.2.2 Cyclon

Cyclon [43] is presented as a complete framework for inexpensive membership management. One of the main points is the improved version of the shuffling algorithm, compared to what was presented in [41]. For this reason, it offers better fault tolerance than Scamp. It maintains fixed sized partial views in each node, where the size is a global parameter. If the size is too high, it takes more time and effort to maintain the view. If it is too low, there is an increased probability that the overlay becomes partitioned and nodes isolated. For this reason, this trade-off must be taken into account while configuring this parameter.

In order to join the overlay, a message is send to a previously known node, called the introducer (effectively this node acts as a contact node in the previously discussed solutions). The operation is based on fixed length random walks, which means that the messages stop being propagated in the overlay when all random walks have expired (the time to live parameter reaches zero).

Periodically, each node in the overlay executes a shuffle operation. When a node performs this operation, it selects the oldest neighbor in its partial view, according to its age [2], which is incremented at the beginning of each shuffling. The node that starts the operation provides to the selected neighbor a sample of its partial view. This node then waits for a reply by the selected peer containing a similar sample. If no reply is received, it is assumed that the node has failed, therefore it will remove it from it's partial view. Both nodes will try to fill the view with the received peers. If it's not possible, they will replace already existing ones. Because of this replacement and also because the oldest node is chosen, failed nodes will eventually be removed from all partial views.

2.5.2.3 HyParView

HyParView [32] is a highly scalable gossip based membership protocol. It is able to offer high resilience and high delivery reliability for message broadcast using gossip on top of the overlay even in churn scenarios. It relies on a hybrid approach, by maintaining a passive view and an active view in each node, which are managed through different strategies.

HyParView uses both reactive and cycle strategies, which makes it hybrid. In order to maintain the active view, it uses a reactive strategy. Nodes react to events related to the change of the membership, as join/leave/failure events. In order to join the membership, a node must previously know another node that is already in the system (and is part of the overlay), as in Scamp and Cyclon.

In a node's active view, there are the nodes to which it has a communication link. Those nodes are called its neighbors/peers. In the passive view, a node has a set of backup identifiers of nodes that are part of the system but with whom the local node does not interact actively. To manage the passive view, HyParView uses a cyclic strategy, by periodically exchanging information with other nodes. For this end, a node will perform a shuffle operation with another one from its active view. Each message sent during this shuffling operation contains a subset of nodes from both active view and passive view of the sender. Both nodes involved in the exchange should then integrate the received members in their own passive views.

The gossip strategy is based on the use of TCP, as a reliable transport protocol. This is also used as an unreliable failure detector. This approach facilitates the implementation

[2]Node identifiers are enriched with an age value, whose initial value is zero

of the reactive strategy. The detection of failures of nodes in a node's active view is also quite fast, because all members are tested at each gossip step. By using TCP as a failure detector, HyParView provides ideal fast healing properties required by gossip protocols.

In a scenario in which 80% of the nodes fail at the same time, as long as the overlay stands connected, HyParView's evaluation show that it can achieve 100% of reliability for message dissemination. This comes from the capacity of the protocol to fastly react to failures in the system.

2.5.2.4 SWIM

The SWIM [10] or the Scalable Weakly-consistent Infection-style process group Membership protocol has two main and well-defined components: a failure detector and a dissemination component.

The first component is based on the failure detector presented in [19], which is fully decentralized. It then executes a randomized probe-based failure detector (the authors later suggested an optimization to use round-robin probe target selection), meaning that in each *protocol period*, each node belonging to the membership picks one of the members at random to check whether it is alive. If no answer is received, the request is then forwarded to other k members, which will execute an indirect probe. This approach to failure detection will be later explained in greater detail.

The second component is a gossip-based dissemination component. It is used to maintain the list of peers that each node is aware about. Anytime a member joins, leaves the group or fails, the update is disseminated (i.e. infection-style dissemination). In order to make information dissemination more efficient, those updates are piggybacked on top of the messages sent during the failure detection procedure.

The SWIM paper also suggests an improvement in failure detection. The authors introduce a suspicion mechanism in order to reduce the number of false positive failure detections, which may be caused when nodes slow down (due to high load) or upon temporary unavailability due to a network partition around itself. The *suspect* state is an intermediate state between *alive* and *failed*. When a node is suspected to have failed, a *suspect* message is gossiped via the dissemination mechanism described above. If a node is able to ping a suspected member, it removes the suspicion and spreads an *alive* message. The same happens if the suspected node receives a suspiction message regarding itself. This procedure obviously reduces the number of false positives when it comes to failure detection, altough it prolongs the detection time.

2.5.3 Partial Structured overlays

Unstructured overlays, like the ones we have seen, present a low cost to build and maintain. Usually, they are more resilient to node failures, although they do not allow to exploit properties of the physical network (such as node proximity, or links speed) or other more specific properties such as content proximity, which may be required by some applications. On the other hand, structured overlays are good for performing efficient search or application-level routing, even though they lack the flexibility to maintain the structure in a scenario with many failures.

For this reason, we can try to imbue some structure on unstructured overlays. This typically means we apply an optimization procedure in unstructured overlays to leverage their good properties mentioned before. For example, some knowledge on the underlying network topology can make the overlay topology evolve to become more efficient with regard to the properties in the underlay.

According to [33], connectivity, uniform degree distribution and low clustering coefficient, are the three main properties of random overlays that must be preserved to ensure the correct operation of these overlays. Those properties will be addressed in full detail later in this section, amongst other ones which we find relevant.

2.5.3.1 T-MAN

T-MAN [24] is protocol that creates an overlay that can be classified as partially structured. It is able to eventually transform a random topology into a desired target topology by the means of views exchange and a ordering function. It starts with a peer sampling service that creates a random graph but is then improved to lead the system to converge to a given target topology.

Each node maintains a partial view. Periodically, every node exchanges its partial view with the first node of their local view, according to the ordering calculated by the ranking function (which depends on the target topology). After receiving the message, the receiver will execute the same procedure as the initiator so they can later merge their local views and apply the ranking function. Each node updates its partial view with the top ranking elements according to the ranking function. Using these procedures, nodes use their peers' views to improve their own views, so that the overlay will become closer to the desired goal of the protocol. However, T-MAN does not aim to maintain a balanced degree between the nodes in the overlay. This creates a problem in the overlay connectivity, by introducing unbalanced load distribution and in the worst case, isolated nodes in the overlay graph.

2.5.3.2 X-BOT

The X-BOT [34] protocol can be placed in this category for similar reasons as T-MAN. It optimizes a random graph overlay network according to some performance criteria. Contrary to T-MAN, it strives to preserve the connectivity of the overlay, by preserving the degree of the nodes while adjusting the contents of each node partial view. It attempts links replacement only when a partial view is full, which helps maintaining the properties of the initial overlay. It also preserves low clustering coefficient and low overlay diameter.

X-BOT is built on top of HyParView. However, it relaxes stability so as to keep improving the overlay. It uses a 4-node optimization technique, which means that each optimization round involves 4 nodes of the system. The algorithm relies on a local oracle to estimate link costs. Therefore, it can bias the network according to different cost metrics (such as latency) by using different oracles. The oracle can be implemented according to the requirements of the services or applications using the membership service.

Each round in the algorithm is composed of 4 steps, one for each node participating in the optimization procedure. The passive view is kept unbiased, which means it contains only randomly selected nodes. However, it will be updated continuously, reflecting changes in the membership. When the 4-node coordination strategy is done, nodes involved in this mechanism keep their degree, which helps maintaining the overlay connectivity. This prevents the overlay from losing fault resilience due to the changes produced by the optimization procedure.

2.5.3.3 PlumTree

In order to mitigate the redundancy problem on gossip-based broadcast schemes using the eager push strategy, PlumTree [31] uses a structured overlay that establishes a multicast tree covering all membership' nodes.

PlumTree (push-lazy-push multicast tree) was developed to leverage on the properties of HyParView. However, it's not limited to its peer sampling service. It aims to reduce redundancy, so as to achieve low overhead, without compromising the fault-tolerance and reliability properties of the underlying unstructured protocol. It reacts to changes in the gossip overlay (such as nodes joining and leaving the overlay), that are then notified to PlumTree using NeighborUp and NeighborDown primitives.

In a nutshell, the protocol combines basic flooding with a prune mechanism. It relies on two main components, which can be defined as tree construction and tree repair. The construction of the multicast tree uses a source-based strategy (because links in the tree are selected based on messages sent from a given node). Initially, all links belong to the tree. When a message is received twice, the second link from which the message was received is pruned, so it is removed from the tree. These removed links will be eventually used in case of faults to recover the tree denoted by by the links that were not pruned. This make the protocol resilient and self-healing. However, PlumTree is optimized for systems with a single sender. In other cases, each node may maintain a spanning tree optimized for itself.

The protocol was proven to deliver no redundant messages under completely stable environments. On scenarios with failures, the protocol exhibits an increase of redundant messages as a result of the healing process of the overlay, which is expected. Furthermore, PlumTree was shown to be able to regain or maintain reliability under extreme failure scenarios.

2.5.3.4 Thicket

Thicket [16] is another decentralized algorithm that allows efficient build and maintenance of independent multiple trees over an unstructured overlay network. This protocol addresses an inherent main problem of P2P overlays which rely on a single spanning tree (such as PlumTree): leaf nodes support much less load than interior nodes.

By building multiple trees, Thicket allows to achieve load distribution by striving to build T trees in such a way that each node is interior in a single tree while being a leaf nodes in the remaining ones.

Similarly to PlumTree, it makes use of an underlying peer sampling service to build the P2P overlay network. The peer sampling service must, of course, implement a reactive strategy in the maintenance of the partial views. It is responsible for detecting changes in the partial view maintained locally and notifying Thicket upon those changes. Thicket will then handle the notifications according to the algorithm.

The algorithm then ensures that all nodes are connected to every tree. It also detects and recover from tree partitions upon node failures and tries to keep the tree heights as small as possible. Each node is the root of at most one tree. For implementation purposes, the authors used PlumTree protocol for building the trees.

Thicket allows a significant improve of load balancing when comparing to PlumTree. It is also extremly resilient to node failures, as it can tolerate massive failure scenarios as well as constant rate failures. Similarly to PlumTree, Thicket exploits links of the random overlay which are not part of the spanning trees (i.e. that are not used as tree branches), allowing efficient detection of network partitions and tree reparation.

2.5.4 Comparation metrics

In order to compare the different gossip-based membership protocols, we need to define a group of metrics to gauge their quality. Here we follow the proposal that can be found in [32, 34]. Based on those metrics, we will be able to experimentally evaluate the protocols which match our interests as well as our own solution.

Connectivity: The overlay graph generated by the membership protocol should be connected, which means that every node should be able to reach any other node in the overlay. If the graph is not connected, we cannot guarantee reliability and isolated nodes will not receive broadcast messages nor be able to participate in the system.

Degree distribution: The degree of a node in a graph is the number of neighbors of that node, which is equal to the number of its edges. In an asymmetric overlay, the in-degree of a node a is the number of nodes which contain a's identifier in its partial views. On the other hand, the out-degree is the number of nodes' identifiers in a's partial view. While the in-degree represents the node's reachability, the out-degree represents the node contribution to the membership protocol. Either in symmetric or asymmetric overlays, the nodes' degree should be evenly distributed across all nodes in the system, otherwise the probability of failure is not uniformly distributed in the node space.

Average Path Length: A path length is the number of edges in a path between two nodes. The average path length in an overlay is the average of the shortest path length between any two nodes in the overlay. A reduced path length usually means that a given message will be disseminated faster between the origin and the destination(s). Furthermore interaction between nodes made through the overlay will tend to be faster and more efficient from the network usage stand point. For this reason, a low average path length is good for efficient disseminations in the overlay.

Clustering Coefficient: The clustering coefficient of a node is the number of edges between its neighbors divided by the maximum possible number of edges across the whole neighborhood. It represents the density of neighboring relations across the neighborhood of a node. The clustering coefficient in a overlay is the average of the coefficient in every node. High clustering coefficient can make the overlay susceptible to partitioning and introduces more redundant messages.

Overlay Cost: It is the sum of cost for all edges that are part of the overlay graph. Usually the costs are associated to link latency. However, overlay protocols do not usually calculate the costs of the links between nodes.

Accuracy: Average of the percentage of active (i.e. correct) nodes in each node's view, in a given moment. If the accuracy is low, messages to inactive nodes will likely be propagated (because there is high probability that failed nodes will be selected as targets), causing useless overhead.

Reliability: Measures the percentage of correct nodes that receive a gossip broadcast. If 100%, it means every active node is able to receive propagated messages.

Redundancy: Measures the number of messages received two or more times. It analyses the message overhead in the gossip dissemination mechanisms.

2.6 Failure detection

As we know, node failures and network anomalies cannot be predicted. Clusters are growing in size and complexity. As a consequence, distributed systems must be able to handle these events by design. Those types of failures can slow down processes in data centers, as well as affect user-facing services. For that reason, memberships must implement or make use of some kind of failure detection and recovery mechanisms.

To correctly implement a mechanism for failure detection is no trivial task, mainly if we want it to be complete, quick, accurate and scalable. The work presented in[6] formally addresses two main properties to characterize failure detectors: completeness and accuracy.

Completeness requires that a failure detector eventually holds a suspicion of failure on every process that actually crashes. The degree of completeness depends on the number of non-faulty processes that suspect a faulty node in a given period of time.

Accuracy restricts the mistakes that can be made by a failure detector. The degree of accuracy depends on the number of mistakes that a failure detector mechanism has made in a given period of time.

Typically, the membership component in a distributed system performs periodic monitoring tasks itself, and then handles the failure, disseminating the update according to the protocol. However, many membership services rely on an external module that runs a failure detection mechanism.

2.6.1 Models

While we cannot guarantee that any computation, including network messages delivery will complete within an a priori known amount of time, timeouts have been widely used as a solution for failure detectors. The work presented in [15] describes *pull* and *push* models as the foundations to failure detector algorithms, which we will apply to distributed systems and, in particular, to memberships.

Push model relies on heartbeats. Processes periodically send heartbeat messages to let others know they are still alive. When processes stop sending those messages for a while (i.e. according to a specified timeout), they become suspected of failure. A slightly different way of implementing a push model would be couting the total number of heartbeat messages received from each other process [1]. Using this approach, a process is marked as suspected of failure when the counter stops increasing, which is monitored by the use of a periodic procedure as well. Traditional distributed heartbeating schemes are not scalable because every group member periodically transmits a "heartbeat" message to every other one. However, that does not have to be the case. Membership nodes can share the responsibility between themselves and then gossip the information.

Pull models rely on a *are-you-alive* approach. Liveness requests are sent periodically to processes. If an answer is not received before a certain amount of time, the process is suspected of failure. As in the push model, it is not necessary that each process monitors every other one. In order to achieve strong completeness, each process should eventually learn about every failed process. This can be achieved through information dissemination.

Dual model relies on both push and pull approaches. This model requires that some process are push-aware, while others are push-unaware. The ones in the first category use a push aproach, periodically sending *I-am-alive* messages, while the others answer to liveness requests. For this reason, this model is not usually applied in memberships, since it requires that different processes use different approaches for failure detection.

2.6.1.1 Gupta's failure detector

Push and pull models and related variants are widely used by many applications which rely on memberships. Both models are relatively simple to implement. The work in [19] presents an interesting approach based on a pull model. This approach was used as a baseline for the SWIM protocol failure detection component [10]. It presents a randomized and distributed failure detector algorithm that aims to statisfy completeness, accuracy and also speed requirements. Furthermore, it imposes an equal expected load per member. In each *protocol period*, each node belonging to the membership picks one of its peers at random to check its status. It then sends it a *ping* message. This procedure is called *direct probe*. If an *ack* message is not received in time (according to a configurable period of time), the *indirect probe* procedure is started. The probing member choses k other members and asks them to check wether the member under investigation is alive (this method distributes the decision on failure detection across a subgroup of members). They then forward the *ping* request. If any of them receives an *ack* message, they forward it to the member which initiated the check. If no *ack* message was received, either from the direct or indirect probe, the member under investigation is considered to have failed. The authors also suggested an optimization to trade off worse scale and speed properties for better accuracy, which relies on a suspicion failure mechanism, waiting for sometime before declaring that a node which was suspected of failure actually failed. This suggestion was used as a baseline in SWIM (the optimization is introduced as an extension, but in practice it is a necessary part of the protocol), as described in the previous section.

2.6.2 Adaptive failure detectors

Adaptive failure detectors are also widely used as variants of the models described above. Their goal is to adapt to changing network conditions. They are often based on a push model, relying on heartbeats. However, pull model based ones are also common. Furthermore, if a timeout is set to a value that is too high, a failure detection takes longer. On the other end, the detector can lose accuracy, which means making more mistakes, if we timeout too early. The main difference between adaptive failure detectors and traditional ones is that the used timeout can be changed on-the-fly (i.e. dynamically) according to network conditions, such as the communication latency.

Both push and pull models have variants when it comes to adaptive failure detectors. The key aspect is the use of estimation functions that can calculate better results for timeout values as the time goes on, according to past arrival times. The work in [7] presents a push-based approach in which by the use of arrival times sampled in the recent past, a function computes an estimation of the next heartbeat's arrival time. The timeout value will then be changed based on that estimation. On the other hand, the work in [23] presents a pull model approach that estimates the round-trip time (RTT). In order to create an estimation, it uses a function which gives more weight to more recently registered RTT delays when comparing to older ones. By this means, it is able to estimate an RTT according to the current network conditions and then change the timeout value.

2.6.3 Accrual failure detection

Traditional failure detectors are often based on a boolean/binary failure detection strategy where a process is considered either alive or dead (trust or suspect). However, finer interpretations may become usefull as not all applications have the same fault tolerance. On contrary to common failure detectors, an accrual failure detector [21] instead outputs suspicion information in a spectrum of values. This type of failure detector separates the monitoring part (that gathers information concerning other processes, such as messages arrival times) from interpretation part (interprets the monitoring information to decide whether a process should be considered suspected of failure), oppositely to traditional ones, in which the output is binary, so there is no need for a separated component for interpretation.

An accrual failure detector outputs $susp\_level_p(t)$ as a suspicion level at a time t for a process p. According to [21], this category of failure detectors must exhibit some properties:

Asymptotic completeness: If p fails, the suspicion level tends to infinity as t increases to infinity.

Eventual monotony: If p fails, $susp\_level_p(t)$ eventually increases monotonically, since it only decreases when it is reset to zero (by the Reset property).

Upper bound: No process p is correct unless $susp\_level_p(t)$ has an upper bound for all t.

Reset: If p is not faulty, for any t_0, there is always a time $t \geq t_0$ when the suspicion level equals zero. It starts as zero and is resetted when the detector receives a hearbeat message from p.

2.6.3.1 φ accrual failure detector

The work presented in [21] proposes the φ accrual failure detector, based on heartbeats, a failure detector that implements the abstraction explained above.

A value called φ represents the level of suspicion of a given process. The key aspect of this approach is to express φ on a scale. That scale is dynamically adjusted based on current network status, working as adaptive failure detection. The distribution of inter-arrival times is used to compute the probabilistic estimate φ by the means of previously stored (in a sampling window) heartbeat arrival times. The value of φ is calculated using the following equation:

$$\varphi(t_{now}) = \text{-} \log_{10}(\ \mathrm{P}_{later}(t_{now} - T_{last}))$$

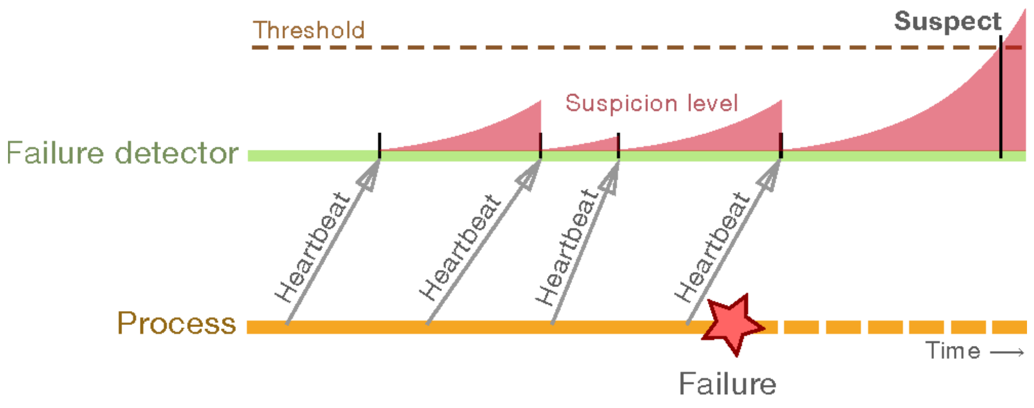

Figure 2.1: Example of execution of φ failure detector (taken from [36])

T_{last} denotes the arrival time of the most recent heartbeat, t_{now} denotes the current time and $P_{later(t)}$ makes use of a separated formula which denotes the probability that the difference between the next heartbeat and the previous one will be more than t time units.

A Φ, set by the application, is then use to interpret the monitoring information on some process. A process is suspected when $\varphi \geq \Phi$, as it is possible to understand through Figure 2.1. The use of a greater threshold improves accuracy by taking more time to consider a process as failed. A shorter threshold shortens the time it takes to detect failures by the cost of greater probability of failure. Nevertheless, as the value of φ depends on previously sampled arrival times, we are still able to achieve fairly good accuracy. The authors point out that the probability of mistake when declaring a process as suspected is 10% for $\Phi = 1$, 1% for $\Phi = 2$, 0.1% for $\Phi = 3$, and so on, due to the use of the logaritmic function. Different applications can then use different threshold values regarding their own requirements.

2.7 Cassandra — architecture and membership

We now discuss Cassandra as an example of a large-scale distributed system. We study its membership service as a possible solution for scalable memberships.

In 2004, Facebook was founded. It is currently the largest platform of social networking in the world with more than one billion users. Thus, there is the need of a huge number of servers operating at the same time, all over the world. In order to build such a big platform, reliability, scalability and availability must be satisfied, which includes dealing with a lots failures. For this reason, Cassandra was developed. In 2008, Cassandra was open sourced and became a top-level Apache project in 2010.

Cassandra [28] is a distributed NoSQL database management system designed to manage large amounts of data across hundreds of nodes in different data centers. It's data model will not be addressed in this document. Rather, we will focus our discussion on its membership service, which is a use case for applying our work.

2.7.1 Architecture

Cassandra is built-for-scale and its architecture is responsible for its scalability and availability. It relies on a ring-based membership architecture in which all nodes play identical roles, communicating via gossip. Being different from a master-slave architecture, it has no single point of failure.

2.7.1.1 Partitioning and Replication

Cassandra uses a hash function to balance the load of the nodes in the cluster. To this end, it partitions data across the cluster using consistent hashing [25], leveraging a ring topology as in to those of DHTs, although using a global membership. Nodes get assigned data items according to the hash of the item's unique key. Those hashes belong to the same output range as node's hashes. Each data item will be associated with a node by searching for the first node in the ring with a position larger than the item's identifiers. This means that each node will be assigned as many keys as those that fall between its own and its predecessor's identifiers.

Each data item is replicated at N hosts, where N is a global parameter of the system. Each node replicates the keys from which it is responsible (within its range) at N-1 nodes in the ring. The replicas are chosen according to policies that can also be configured. In Cassandra 1.2, the strategy called *SimpleStrategy* makes the N-1 replicas be chosen by picking N-1 successors of the coordinator. Furthermore, no node should be responsible for more than N-1 ranges.

Since the 1.2 release, Cassandra takes a new approach when it comes to routing read requests to replicas. It tracks the latency in the answers from replicas and routes requests to the fastest nodes whenever it's possible. In the same way, it avoids routing requests to poorly performing but alive nodes. This monitoring strategy is called dynamic snitch [13].

2.7.2 Membership

Gossip in Cassandra is used not only in replication, but also in the cluster membership management and to share control information.

In order to provide a membership service which satisfies the needs, Cassandra uses a gossip protocol based on Scuttlebutt, a anti-entropy gossip protocol [38]. In an anti-entropy gossip protocol, information is shared until it is made obsolete by newer versions. They are usually used to repair replicated data and are also useful for reliably sharing information. These protocols are able to guarantee reliable dissemination.

Scuttlebut is an efficient reconciliation and flow-control gossip protocol. It is used to propagate information across a distributed system. It uses three main structures: one to keep the state, a vector clock and a history. Peers exchange vector clocks, which keep track of logical time for a set of events. The flow control is a decentralized mechanism which allows to determine the maximum rate at which nodes can submit updates without creating a backlog of updates.

Cassandra also relies on a seed mechanism. Seeds are nodes whose addresses is previously known by the other nodes. They allow other nodes to find the cluster and then join membership during startup.

There is also an underlying failure detection mechanism, based on φ accrual failure detector [21], used to mark failing nodes and remove them from the membership. This action depends on the timing of gossip messages from other nodes in the cluster. The likelihood of the removal depends on a threshold value, which is dynamically calculated. After the removal, some nodes will periodically try to re-establish connection with the marked nodes. When a faulty node recovers, the messages stored are sent to it, by the use of the hinted hand-off technique.

2.8 NAT and firewall-aware Membership

In today's Internet, a big amount of edge nodes are behind NATs (Network Address Translation gateways) and firewalls. In order to scale memberships of large-scale systems that can run in the cloud or in the edge, we may have to deal with the presence of NATs and firewalls in the network, which block many peer-wise interactions. Since nodes cannot easily establish direct connections to these private nodes, they become under-represented in partial views and therefore this may unbalance the amount of gossip exchanges in which public and private nodes take part. In this section we will present some approaches to deal with NATs and firewalls in the network.

2.8.1 Relaying

Usual P2P protocols that handle NATs and firewalls circumvent them by the use of traffic relaying and hole-punching. These techniques are used to solve the issue of balancing gossip exchange in networks with NATs and firewalls. A relaying strategy allows routing packets to nodes behind those, through public relay nodes, as we can see in Figure 2.2. A hole-punching approach makes use of a public node (called public rendezvous node) leveraging the public node to establish a direct connection to a private node or between private nodes, by exchanging some information (Figure 2.3). After this technique is applied, both peers can send messages directly to one another, as the NATs and firewalls already have the necessary filtering information.

The work in [29] presents an approach to fairly distribute relay traffic over public nodes. It has no coordination overhead, since it requires only local information. The main aspect of this solution is that nodes sometimes participate in complete gossip exchanges while sometimes they act as routers for nodes in confinement domains in the network, forwarding the information, instead of executing the usual gossip broadcast operation.

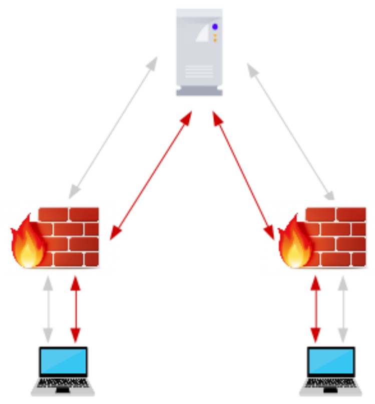

Figure 2.2: Relaying technique. Every message to be exchanged between the two peers will be firstly sent to the public relay node, which will forward it to the destination. If one of those nodes is public, the communication is even simpler, as any packets destinated to it will not be analyzed by either firewall or NAT devices.

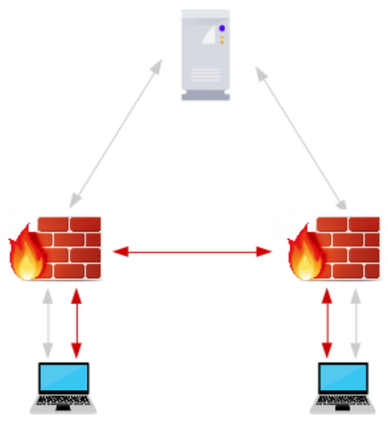

Figure 2.3: Hole-punching technique. The source peer first pings the destination and then sends a request through the public rendezvous node. When receiving the request, the destination node sends a *pong* message, also creating a new filtering rule that allows messages from that source peer [26]. Afterwards, the peers are able to communicate directly.

Each node keeps track of a value which increases when it initiates a gossip exchange and decreases when it accepts a gossip exchange. Periodically, a node initiates a gossip exchange with a peer from its view (that depends on the underlying protocol used for gossiping), increasing the value, called quota. A positive value for the quota means the node accepts gossip exchanges. Otherwise it acts only as a router, forwarding gossip requests. This solution is then able to ensure that every peer participates in a similar number of gossip exchanges.

2.8.2 Non-relaying

Using relaying mechanisms as well as hole-punching means increasing the overhead in message exchanges and the complexity in membership protocols as a result of having to maintain routing tables and mappings that explicitly deal with private nodes. The solution presented in [12] describes the first NAT-aware peer sampling service which does not rely on those mechanisms but rather on a new one. It requires the use of two partial views which are bounded in size: a public view, with public nodes' identifiers, and a private view, with identifiers of nodes behind NATs. Each descriptor contains a node's address, NAT type, and its age (number of exchange rounds since creation).

In order to generate a uniform random sample from the two views, there is the need of a mechanism to identify a node as being either public or private. Croupier includes a distributed NAT type identification protocol in order to achieve that goal. Furthermore, public nodes collectively estimate the ratio of public to private nodes in the system. Public nodes act as Croupiers, exchanging views on behalf of private nodes. By separating both public and private nodes, we can then guarantee that public nodes will not try to initiate communication with private nodes, but rather the opposite. Therefore, if a private node initiates communication with a public node, its NAT device creates a filtering rule which will make any later incoming messages from the public node be forwarded to the node that started the procedure. The opposite could not happen, has there is no previous rule allowing incoming messages from the public node to be forwarded.

Each node periodically tries to exchange and update both of its views and ratio estimations. It removes the oldest descriptor from its public view and sends a shuffle request to the same node, with a subset of its public and private view. The Croupier will then update its views and send a subset of each one. Simillary, the first node updates its views and its estimations on the ratio. Experimental results show that Croupier has good connectivity after failures when compared to relaying-based NAT-aware protocols. Furthermore, its procedures can be integrated in already existing peer sampling services.

2.9 Existing solutions

In this section, we aim to briefly describe some membership solutions as peer sampling services which can be used for supporting the operation of large-scale systems and possibly allow an edge service to be built on top of them.

2.9.1 Cassandra - Gossip 2.0

Cassandra's architecture uses a structured overlay. For that reason, it has to deal with the same issues as structured overlays, as described above, like the lack of flexibility to maintain structure in a churn scenario. Also, it was designed in a way that it tries to improve performance to make possible to complete reads and writes rapidly. In Cassandra's current design, each node maintains a global view of the nodes of the whole system, allowing fast replication. However, gossiping in a structured overlay with a global view increases the probability of exchanging messages between distant nodes. Although the global view provides some good properties as described, a partial view with the right nodes could be enough. One obvious approach would be having direct connections between nodes which replicate the same data, and they would be included in each other's partial views. This means there is no need of having open connections to every node, but rather to the ones with which communication is most usual.

Cassandra's replication benefits from consistent hashing and its one-hop DHT-structure, due to the easiness of finding the nodes which possess a given data. However, the membership service could also benefit from partial views, as noted above. Nonetheless, we should not build two completely independent overlays, one for the replication and another one for membership. This would mean increasing the number of open connections between peers in the system, which would create a significant overhead.

The issue described in Gossip 2.0 [4] identifies the need of changes in the dissemination aspects of the current Cassandra's gossip subsystem. In order to do so, Jason Brown enumerates three main requirements: a peer sampling service which is based on partial views [22], a broadcast tree protocol based on the peer sampling service [3] an anti-entropy component for dissemination.

Brown has fully implemented the first two requirements of the issue described. For the first requirement, he proposes to change Cassandra's membership in a way that it implements a peer sampling service for partial cluster views through the use of HyParView. HyParView is a self-healing and self-balancing membership gossip protocol, which provides each node with a partial view of the system. Partial views combined create a fully-connected mesh over the cluster, without the need of having direct connections between every node, which is a big overhead in the system. The strategy used also adds a remote view in order to deal differently with nodes from other data centers. Otherwise, in the presence of multiple data centers, lots of connections would be stablished between both, creating a high overload in routers. The second requirement stated is a broadcast tree which creates dynamic spanning trees according to the partial views provided by the peer sampling service. This used approach relies on Thicket [16], which describes a dynamic and self-healing broadcast tree. The algorithm computes a tree for each node in the cluster (in which the root is each of the nodes). The broadcast tree allows a node to efficiently send updates to all the other nodes in the cluster with the use of a balanced, self-healing tree based on the node partial view. As for the failure detection mechanism, this approach maintains the use of an implementation based on φ accrual failure detector, as in Cassandra's current version.

2.9.2 Partisan

Partisan [35] is an open-source distribution layer for Elixir and Erlang. It provides a scalable TCP-based membership which performs better than Distributed Erlang. Furthermore, it is the solution used in order to provide Lasp with a scalable cluster membership. Lap is a suite of libraries based on a programming model that can be used to build correct distributed applications. Lasp aims at allowing programmers to develop applications which can be executed on one computer or distributed across many and still have the same behavior. Applications that are written in Lasp share a fully replicated state between each node in the system. Each node modifies its own copy and the effects of the modifications are propagated to the other nodes. In order to keep track of the nodes in the system, Lasp needs a scalable membership component.

Partisan is implemented as a library for Erlang 19.3. It is a highly-available and highly scalable replacement for Distributed Erlang. Partisan allows developers to specify the network topology at runtime. It supports five cluster topologies: static, full mesh, client-server, publish-subscribe and peer-to-peer. Partisan also provides a well-defined set of functions as part of its API:

- join(Node) - given the identifier of an other node, it joins to that node's cluster.
- leave(Node) - removes node from the cluster
- members() - return a collection of locally known cluster members
- forwardMessage(Node, RemotePid, Message) - forwards a message asynchronously

For the peer-to-peer topology, which is the most relevant one for this essay, Partisan uses a membership based on both HyParView and PlumTree. The use of HyParView provides global system connectivity in a scalable way. As we described before, it makes use of partial views that form a single connected component over the cluster. Furthermore, it uses TCP as an unreliable failure detector. PlumTree provides reliable broadcast, combining a deterministic tree-based broadcast protocol relying in HyParView as the underlying peer sampling service. The protocol is able to achieve no redundancy in stable environments through the use of spanning trees. This approach benefits from HyParView's resilience and PlumTree's efficiency. In order to communicate between non-directly connect nodes, Partisan's back-end computed a spanning tree routed at each node using an instance of PlumTree.

2.9.3 Serf

Serf [40] is a Go library for decentralized cluster membership, failure detection and orchestration. It is provided by HashiCorp [20], an open-source software company that provides tools to run and connect distributed application infrastructures. Consul [8] is one of HashiCorp's tools. It provides distributed key-value storage, together with a DNS based service discovery, RPC and event propagation. Serf provides Consul with the underlying membership, failure detection and event mechanism. However, Serf can be used together with other applications, not only those provided by HashiCorp.

Serf makes use of the gossip system provided by SWIM in order to solve three major problems: membership, failure detection and recovery and custom event propagation. In this section we focus on the first two. In order to provide an efficient and lightweight gossip-based membership, SWIM is used, along with some minor changes that help increasing measures such as propagation speed and convergence rate. Since we described SWIM before, we will focus on those changes.

Full state sync Gossip is done over UDP with a predefined fanout and interval. Nonetheless, Serf periodically does a full state sync over TCP, by the use of a push-pull anti-entropy mechanism. Also when a node joins the cluster, it does a full state sync with the contact node over TCP, and then it normally gossips its existence to the cluster. This change was implemented in order to recover from network partitions and to faster allow node joining and rejoining. Furthermore, it allows quicker convergence of the membership list. The original SWIM implementation, on the other hand, would make a joining node wait to be pinged by every other node to finally have full knowledge of the network.

Dedicated gossip layer SWIM state updates have very high latency. This is because gossip only makes use of piggybacking to send state updates together with failure detection messages. It means that, if we pick a one second failure detection interval, we are also only able to send one gossip message to one peer every second, which tends to cause a very slow convergence rate. In order to solve this major issue, Serf has a dedicated gossip layer decoupled from the failure detection component. This allows the two components to have different timers. We can still have a failure detection rate at once per second and a higher gossip rate, such as once per 200ms, allowing faster convergence rates. Serf also makes use of a fanout of k, meaning that if there is a state update, it will be gossiped to k different nodes, allowing quicker data dissemination.

Keeping the state of dead nodes As SWIM does not make use of full state syncs, it immediately deletes a node state after learning that it is dead. Serf, on the other hand, maintains the state during a set amount of time, in order to share information about dead nodes during full syncs.

Local health Lifeguard [9] is a set of other extensions to SWIM that are used in Serf. It uses the concept of local health, which is a property measured by the use of heuristics. Timeouts for failure detection are then dynamically adapted based on the node's health, turning this component into an adaptive failure detector. Lifeguard adds a *nack* mechanism to the previously described failure detector component. When an indirect probe results in no answer, the *k* selected members will then send a *nack* message to probing member. The fact that a node is missing *nack* messages, together with the lack of received *acks* as well as needing to refute suspicions against itself, contribute to possibly considering itself unhealthy. That decision is based on the use of a counter which takes all those factors into account. When a node considers itself as unhealthy, it may decrease its timeouts, according to specific Lifeguard's calculations.

2.10 Summary

In this chapter we studied the underlying topics of membership services and how important they are to build scalable and reliable memberships. We also discussed the benefits of using partial views to improve scalability in large-scale systems. Additionally, we briefly discussed how Cassandra builds and manages the system membership and also studied existing membership solutions.

In the following chapter, we will introduce our membership service solution which relies on the topics we discussed so far. We will also explain how we made use of some of the discussed solutions for specific topics to leverage our service.

Chapter 3

Membership Service

3.1 Introduction

This chapter presents the main contributions of this thesis. We describe the main topics of our solution for a scalable membership for edge systems. We start by explaining the assumed system model, followed by its architecture and implementation details.

3.2 System model

Our solution focus on large-scale distributed systems which aim to store and process data in multiple data centers (with possibly different sizes). Furthermore, we focus on systems which also aim to provide edge functionalities across edge nodes in multiple subnets by allowing them to take part in the system and contain a part of the information present in data centers. Distributed systems without any edge functionality can also benefit from our solution. The developed membership service can be integrated on already existing large-scale systems, as we provide an API with a well-defined set of functions.

Our solution assumes an asynchronous model, which means no assumptions are made when it comes to the time it takes to perform any computation or to send and receive messages. It also assumes a model where nodes may fail (i.e. crash or stop responding) and where transient network partitions may occur.

3.3 Architecture

3.3.1 Simplified view

A simplified view of our membership service is shown in Figure 3.1. In fact, many membership services follow this architecture: connections between nodes are managed by the

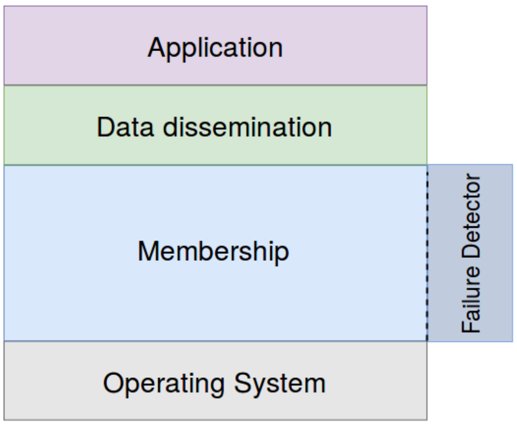

Figure 3.1: Simplified view of our membership solution

Operating System and then used by the membership in order to establish the overlay. The membership component then manages the nodes' state, by relying on the failure detector component. The failure detector component must also receive some input from the membership, such as the peers to be monitored. Having established an overlay, it is possible to provide a data dissemination component that allows the application to use the existing connections between nodes in the membership.

3.3.2 Detailed view

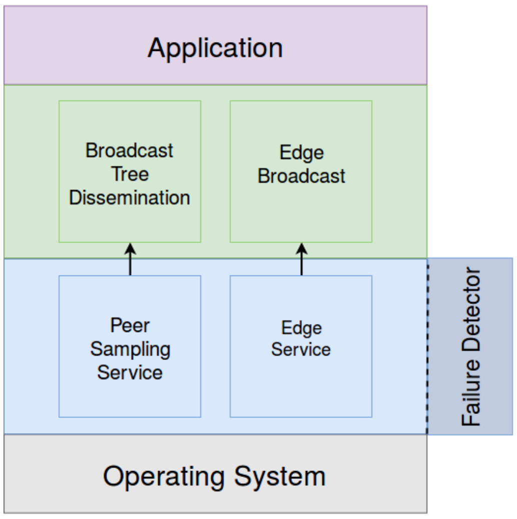

Figure 3.2: Detailed view of our membership solution

Figure 3.2 allows a deeper understanding of our membership service. Two subcomponents make part of our membership: a peer sampling service for inter-node communication inside and between data centers or subnets and another subcomponent which allows communication to be made between different subnets. Both subcomponents rely on TCP. Our peer sampling service relies on HyParView [32] (with some changes), which makes use of TCP, as demanded by the protocol. This allows us to ensure the symmetry of the partial views. Although HyParView relies on TCP as an unreliable failure detector, both of the subcomponents make use of a separated failure detection component based on φ accrual failure detection [21], so as to improve accuracy and completeness.

The subcomponent that concerns the edge communication acts in a different way. For a better understanding of the architecture, we specify some applications of it. A private network connected to a data center may contain many devices which can be used as access points for the clients in that network. A personal laptop can then be used as a node of the system, taking part in the membership and keeping track of some of the other nodes in the data centers. In the same way, a second private network inside the first one may have some devices which can also be seen as data center nodes, from a client's point of view. These networks will obviously contain only a small part of the information existing in the "main" nodes. The goal is to allow clients to faster update and receive information which is more likely accessed by them. For this reason, we build a hierarchy of networks, in which the data center overlay is at the top, and the most restrictive network is at the bottom.

By the use of both subcomponents together, we can establish a proper connected overlay across all nodes, from data center nodes to edge nodes. Inside each private network, nodes still establish an unstructured overlay based on HyParView, as we described for nodes in data centers.

The data dissemination component is also subdivided in two. For information to be broadcasted inside and across data centers, we rely on a broadcast tree protocol which makes use of the underlying peer sampling service. We use Thicket [16], which constructs a tree routed at each node. The trees are use to gossip any membership update originated in any of the nodes. As for the information broadcasted between private networks and to/from the data centers, we simply rely on the connections established in the edge subcomponent of the membership, since our architecture does not allow cycles in the network hierarchy. Both data dissemination subcomponents are used to broadcast information of membership nodes but can also be used by the application, by making use of the established connections.

Furthermore, due to our architecture, there is no need of a specific approach to deal with NATs and firewalls. Non-relaying techniques for NAT and firewall-aware memberships, such as the one presented in [12], rely on the distinguishing nodes from public networks from nodes from private networks. Nodes compute separated public and private views. This allows private nodes to be the ones trying to establish connection with public nodes. Otherwise, the connection would be blocked by any existing NAT or Firewall. Firewalls and NATs usually allow packets from the Internet to pass through if the host has previously send a packet through it to that destination. Essentially, this means that the host behind the firewall or NAT has to be the one initiating the communication. The other end will be able to communicate to it by using the private node's public IP address. Non-relaying techniques are based on that principle. However, the way we designed our architecture actually requires edge nodes (private nodes) to be the ones initiating communication, as we describe in following subsection.

3.3.3 Overlay

As we rely in HyParView to establish the overlay, we end up with an unstructured network overlay, which means that nodes and communication links define a random graph. Even though nodes from different data centers are part of the same overlay, we take into account that they are not from the same data center and we deal with them in a different way, using that as an advantage. Most peer sampling services simply consider nodes from different data centers as part of the same overlay without any special treatment. Furthermore, usually there is no way to guarantee connectivity between data centers. Our approach rather tries to minimize the overhead on routers that are used to communicate between two data centers by allowing each data center node to maintain at most one connection to each remote data center. This also deals with network partition, by ensuring there is always connectivity between data centers, and strives to recover such corrections when all of them become unavailable.

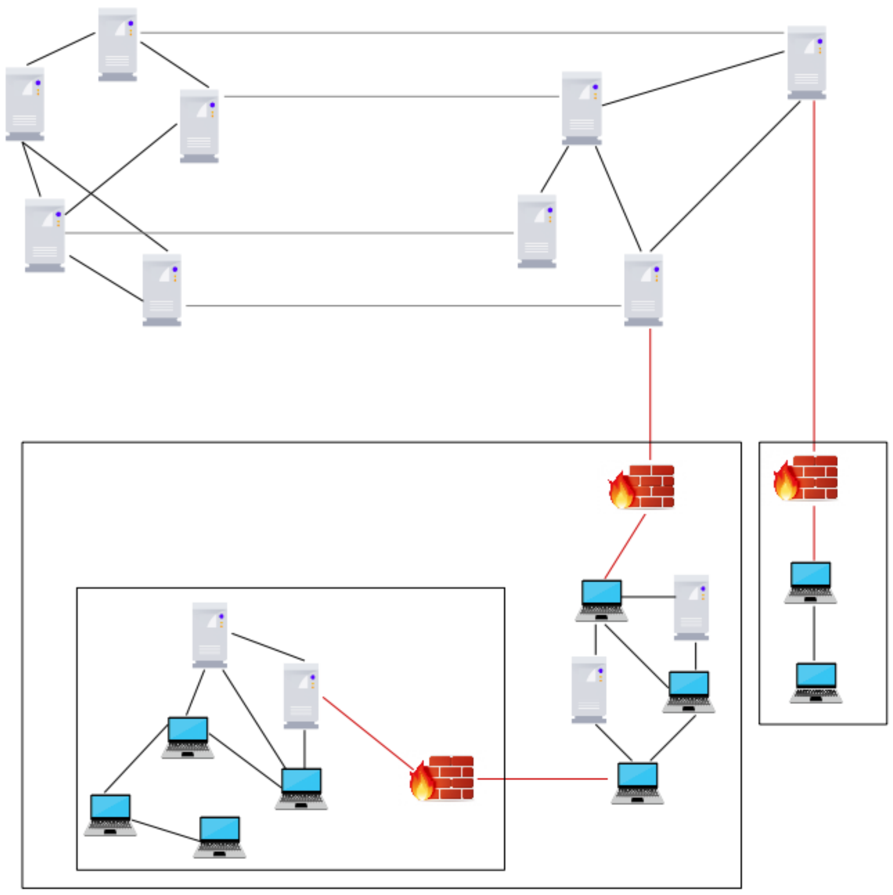

Figure 3.3: Example of a graph resulted from established connection between nodes. Black lines connecting two nodes represent TCP connections resulted from HyParView's procedures. Grey lines represent TCP connections between nodes from different data centers. Red lines represent TCP connections between nodes when there is a firewall or NAT device in the way.

As for the edge connection, networks establish a hierarchy between each other, where the data center overlay is at the top. This hierarchy is static, based on configurable parameters. At most one edge node from each network must establish a connection that we call edge connection, to one node of the parent level in the hierarchy of networks (the chosen node in the parent level is called contact node and communication with it will be initiated on startup). This property has to be guaranteed by the system administrator. Moreover, no different private networks should be connected unless one is containing in the other. By guaranteeing these two properties, we ensure that there are no cycles due to messages exchanged across all networks and data centers. We use this approach in order to simplify our implementation, however in the future, we aim at allowing those connections by adding a mechanism to prevent message redundancy across the whole overlay.

Figure 3.3. depicts a possible overlay generated by our service, which includes the hierarchy containing edge networks. Nodes in a data center establish an overlay which forms a random graph. The same happens between nodes in the same private network if they are not separated through a firewall or a NAT device. The red links, however, are based on the system's configuration. Even though they may vary over time due to failures, an edge node starts by joining with a specified parent level contact node.

3.4 Multi-datacenter and Edge Membership Service

In this section, we will describe the details of our implementation of the specified architecture in the previous section. Firstly, we used Jason Brown's approach [4] that was used for implementing changes in the current Cassandra gossip subsystem. Cassandra's membership is one-hop, allowing each node to communicate directly with any other. However, this introduces some problems in terms of scalability, due to the cost of maintaining so many connections at the same time. Brown proposed and implemented a peer sampling service to create partial views as well as a broadcast tree, based on both HyParView and Thicket respectively. Those changes were integrated directly in Cassandra's source code, which we used as a baseline. We then carefully extracted all the code regarding the membership, to make it usable by other systems, through the use of a well-defined set of functions. Additionally, we have added many more features concerning the peer sampling service, dissemination throughout the trees, as well as an edge service.

3.4.1 Peer Sampling Service

Firstly, we will describe the part of the solution regarding the communication within multiple data centers. We propose the use of a Peer Sampling Service which provides a partial view of a cluster to each node, allowing much more efficient resource management and dissemination of information. This way, there is no need to maintain a connection between every pair of node over the cluster. A combination of all the partial views creates a fully-connected overlay.

We start by summarizing the HyParView protocol, as described by the authors, and then explain the modifications of the protocol in our solution. Those modifications include the management of the passive view and the notion of multiple data centers.

3.4.1.1 HyParView — summary

HyParView is a clearly defined protocol for a Peer Sampling Service. It is used for building unstructured overlays and balances both in and out-degrees of nodes in the overlay. It is a highly scalable gossip based membership protocol. It is able to offer high resilience and high delivery reliability for message broadcast using gossip on top of the overlay even in churn scenarios. It relies on a hybrid approach (using cyclic and reactive strategies), by maintaining a passive view (used as a backup) and a symmetric active view in each node. Furthermore, HyParView ensures complete coverage by making sure all nodes are included in at least one node's view, through the use of a coordination protocol.

When a node first joins the system, it will start by contacting a seed node (the address of the seed node must come from an external source, such as a configuration file). The seed node will then proceed to send a ForwardJoin to the peers in its active view. ForwardJoin requests carry a TTL that is initially set to ARWL (Active Random Walk Length), which is a configurable value that specifies the maximum number of hops that such request is propagated.

When a node p receives a ForwardJoin request, it will accept it, adding the new node to their active view, if an only if, the TTL equals zero or p has only one node in its active view. If a random node has to be dropped (since there is a limit of nodes in one's active view), p will send a Disconnect message to that randomly chosen node, remove it from its active view and insert it in its passive view.

If a node p suspects that one of its peers has failed, it selects a random peer from its passive view and attempts to establish a TCP connection with it. If it succeeds, p will send a Neighbor request with a priority level, depending on the current size of its active view. When a node receives such request, it will identify the priority level and react accordingly (which is explained in greater detail in the original paper). Additionally, a peer may have to be dropped from its active view in order to make room for p.

3.4.1.2 Backup view

Despite being described as having both active and passive views, the used base implementation skips the passive view, mainly to eliminate some extra network traffic. However, the functionality of the view is still retained, as we are able to acquire knowledge of the full cluster with no significant overhead, as explained in section 3.4.3.

Information (i.e. the state) on those nodes in the backlog/passive view is maintained, although they are only used for communication when necessary, as described in HyParView's protocol.

Algorithm 1: Data structures

```
1: Local State:
2:     $N_{id}$ //Node address
3:     $N_{datacenter}$ //Node data center
4:     localView //List: Address
5:     remoteView //Map: $remoteView[N_{datacenter}] \rightarrow$ Address

6: procedure getPeers do
7:     remote ← $\emptyset$
8:     foreach $N_{datacenter} \in remoteView$ do
9:         remote $\cup$ $\{remoteView[N_{datacenter}]\}$
10:    return localView $\cup$ $remote$
```

Since it is possible to obtain knowledge of the full cluster, there is also a periodic check for full cluster connectivity that is run to ensure that a node is fully connected to the peer sampling service. If a node has less peers in its local active view than the fanout number of nodes in the same data center, it will send a HyParView Neighbor request to one of the nodes it knows about from its passive/backup view. It will also try to connect to one (at maximum) node from each remote data center if that data center has a lesser number of nodes, as explained in 3.4.1.3.

The fanout of the local data center is calculated based on the use of a logarithmic function. It replaces the use of a specified limit for the active view size, which would requires some a priori knowledge of the cluster size. This calculation is then not possible in HyParView on its own since there is no knowledge of the full cluster size. The authors, however, show that the protocol can achieve high reliability with a fanout value of $log(n)+c$, having used $c = 1$. Since n is known in our membership, we can calculate the fanout value and use it for procedures such as the full connectivity periodic check. The fanout value is also used to verify if a node can accept another peer connection, by checking if it's view is full, comparing that value with the size of the local active view.

3.4.1.3 Multiple data centers

As we mentioned, the implementation takes into account the possibility of multiple data centers belonging to the same system. This means that nodes from different data centers take part in the same unstructured overlay created using HyParView although they are handled differently. In order to do that, a remote active view is maintained (Algorithm 1, line 5), containing information on nodes from other data centers. The remote view will hold at maximum one active peer for each remote data center. The backlog containing all the nodes in the system allows each node to acquire knowledge on the number of nodes in each data center. This information is used so we can guarantee that a HyParView Neighbor request message for another data center is only sent if the number of local nodes (i.e. nodes from the local data center) is less than or equal to the number of nodes in the remote data center (Algorithm 2, line 10). This approach ensures that nodes in the smaller data center are the ones trying to connect. Otherwise, we could be sending too many messages to those smaller data centers. Let *dc1* be a data center with 10 nodes and *dc2* a bigger data center with 20 nodes. At a given time, the aggregate remote view (i.e. the combined remote view of each node) in *dc2* may contain all the 10 nodes in *dc1*. However, the reverse will not be true, as there will be at most 10 nodes from *dc2* (instead of 20) in the aggregate remote view of *dc1*. HyParView does not deal with network partitions, so this approach is used to guarantee there is always a path between all data centers and that events in *dc1* will be propagated throughout *dc2* and vice-versa. With this modifications, the information sent during membership operations in the HyParView is always sent along with the data center identifier.

Algorithm 2 depicts the most important divergences when maintaining both views when compared to the original algorithm. The procedure that handles Disconnect messages has suffered some modifications. Whenever a node receives a Disconnect message, it will now try to replace the removed node with another one from the same data center (Algorithm 2, line 14), unless the necessary conditions are not verified. This way we are actively striving to protect the connectivity between different data centers. Furthermore, in order to have at most one node from each of the other data centers in the remote view, a high priority Neighbor request from another data center will be accepted and the local node may have to disconnect from an existing peer from that same data center (Algorithm 2, line 37).

3.4.2 Broadcast Tree

We will now describe the used approach for disseminating information. Such information can be either membership updates or data from the application.

We start by summarizing the Thicket protocol, as described by the authors, and then explain the modifications of the protocol in our solution. Those modifications include the join mechanism, the possibility of local data center broadcast, dissemination and a mechanism for recovering missing messages.

3.4.2.1 Thicket — summary

The used solution relies on Thicket, which describes dynamic, balanced and self-healing broadcast trees. Thicket efficiently disseminates information in a peer-to-peer overlay by making use of an underlying peer sampling service such as HyParView, to build the overlay network.

Thicket builds T divergent trees for the whole cluster (T being a configurable value) where most nodes are interior in a single tree while being a leaf in all the other trees. Thicket allows nodes to efficiently send messages to all the other nodes in the cluster without any redundancy after stabilization (no nodes joining or leaving). When using HyParView as a peer sampling service, this means that active views are used to derive the set of downstream peers in each tree. The peer sampling service is then responsible

Algorithm 2: Neighbor handlers

```
1: upon Receive(Disconnect, peer, datacenter) do
2:     existsLocal ← peer ∈ localView
3:     existsRemote ← peer ∈ $remoteView[N_{datacenter}]$
4:     if $existsLocal$ ∨ existsRemote then
5:         localView ← localView \ {peer}
6:         remoteView[$N_{datacenter}$] ← remoteView[$N_{datacenter}$] \ {peer}
7:         sendRequest ← false
8:         if existsLocal then
9:             sendRequest ← isFull(localView)
10:        if existsRemote then
11:            sendRequest ← size($N_{datacenter}$) ≤ size(datacenter)
12:        if sendRequest then
13:            trigger sendNeighborRequest(datacenter)

14: procedure sendNeighborRequest(datacenter) do
15:    newPeer ← getRandomPeer(datacenter)
16:    Send(NeighborRequest, newPeer, $N_{id}$, $N_{datacenter}$)

17: upon Receive(NeighborRequest, priority, peer, datacenter) do
18:    existsLocal ← peer ∈ localView
19:    existsRemote ← peer ∈ $remoteView[N_{datacenter}]$
20:    if $existsLocal$ ∨ existsRemote then
21:        Send(NeighborResponse, ACCEPT, $N_{id}$)
22:        return
23:    if $priority == LOW$ then
24:        if ($datacenter == N_{datacenter}$ ∧ isFull(localView))
           ∨ remoteView[$N_{datacenter}$] ≠ ∅ then
25:            Send(NeighborResponse, DENY, $N_{id}$)
26:            return
27:        trigger addNodeToActiveView(peer)
28:        Send(NeighborResponse, ACCEPT, $N_{id}$)

29: procedure addNodeToActiveView(peer, datacenter) do
30:    if $datacenter == N_{datacenter}$ then
31:        localView ← localView ∪ {peer}
32:        if $isFull(localView)$ then
33:            Send(Disconnect, first, $N_{id}$, $N_{datacenter}$)
34:        return
35:    remoteView[$N_{datacenter}$] ← remoteView[$N_{datacenter}$] ∪ {peer}
36:    if $size(remoteView[N_{datacenter}]) > 1$ then
37:        Send(Disconnect, first, $N_{id}$, $N_{datacenter}$)
```

for notifying this dissemination layer whenever there is a change on the partial view of a node.

According to Thicket, each node maintains a set of *activePeers* per tree. Each node also maintains a set of *backupPeers* with the identifiers of the neighbors that are not being used as its downstream peers for any tree. A tree's *activePeers* set contains all the peers to which the node will propagate any message received from that tree. When a node joins the peer sampling service, the Thicket layer is notified and inserts that node in the *backupPeers* set.

When a node n receives a message from a tree t, if first checks if t was already created locally. If it was, the node simply disseminates the message to the peers in *t.activePeers*. If not, it selects f nodes from the *backupPeers* and forwards them the message, with the tree identifier. On the other hand, if n is already interior in some tree, it stops the branching processing, becoming a leaf in t.

If a received message is a duplicate (i.e. the same message identifier from the same tree was previously received), the node sends a Prune message back to the sender of the message. When receiving a Prune message from a node n, n is moved from *t.activePeers* to *backupPeers*. The branch is then removed from that tree. This mechanism is used for tree construction and prevents redundancy after stabilization.

In order to ensure all nodes eventually become connected to all the spanning trees, each node periodically sends to its *backupPeers* a Summary message with the recently received message identifiers. If the missing messages have not been received after a specified amount of time, nodes send a Graft message to possible candidates from the *backupPeers*, which will evaluate the request and act accordingly. If the candidates reject the request, they answer with a Prune message. The choice of Graft candidates and the evaluation of Graft requests is explained with detail in the protocol specification.

3.4.2.2 *N* trees

Our solution computes a tree for each node in the cluster (i.e. each node has a tree rooted on itself), which is a small change to Thicket, since Thicket works with T trees, instead of n. The authors have used T = 5 for a 10 000 node cluster. However, we rather use $T = N$, N being the number of nodes in the whole cluster. This change, already used in Brown's solution, helps in improving fault-tolerance. With this modification, whenever a node wants to broadcast any information, it simply uses the tree rooted at itself, rather than choosing one of the existing ones, making the implementation easier. Even though this approach increases the number of trees in which each node is interior, the load balancing is still maintained, but rather through the use of more trees. In consequence, each node will also be a leaf in more trees.

Both remote and local HyParView's partial views are used to derive the set of peers in each tree, so a tree should contain all nodes from all data centers. This means that in every tree, the downstream peers of a given node always belong to its HyParView's partial view.

3.4.2.3 Join mechanism

When a node x joins the cluster, it first creates its own state. As soon as the peer sampling service notifies Thicket that we added at least one peer to our partial view, x will try to broadcast its state.

The new node will start constructing its tree by sending a message to nodes in its *backupPeers* set. However, according to Thicket's specification, when a node is already interior to a tree, it stops the broadcast and considers itself a leaf of that tree. The tree will then continue to be constructed through the periodic procedure of sending Summary messages to *backupPeers* and through Graft requests. This means that the time for constructing the whole tree will depend on the time set for those periodic procedures.

Our membership service, however, aims at allowing each node to quickly learn about a new joining member. Waiting for a periodic procedure which may take many seconds to be triggered is then not the best option. For this reason, we have to use a different approach. When a node joins, we make use of its set of *backupPeers* (which will consist solely in a seed node) and broadcast the update through them. Each node, when receiving that update from a new node, executes the same procedure, whether or not it is a leaf in other trees. This procedure is fast since there is no waiting time. Even so, these messages cannot be used to construct the tree: since seeds are always the first nodes to join HyParView's active view, they will consequently be the first downstream peer for the new node's tree. The seed receiving the message would then use its peers to broadcast the update. Those peers would eventually receive duplicated messages from other nodes, which would be answered with Prune messages. This means that nodes would always try to maintain the branch that connects them to the seed, while pruning others, which would create a great overhead in seed nodes. To avoid such scenario, we cannot use those messages for tree construction. We rather execute a periodic step in each node that sends a message through its own tree. Therefore, we are able to construct the tree, with the due tree repairing periodic mechanisms of missing messages of Summary and Graft messages. This prevents a waiting time in then next membership update or application message sent through that tree. This also allows other joining nodes to create a branch to that tree without having to wait. Otherwise, they would first enter a node's Thicket *backupPeers* set, receive Summary messages and then finally send a Graft request, only when such a message was broadcasted. This mechanism, however, makes trees ready for future messages. The timer for this step is set with the help of a random constant, to prevent a message overhead when a large number of nodes start at roughly the same time.

3.4.2.4 Local broadcast

Additionally, our implementation uses a secondary instance of Thicket in order to build and maintain local trees, i.e. trees that contain only nodes from each data center. This is achieved by using only HyParView's local active view, instead of both local and remote (as explained before, the partial view we obtain from the PSS is a main factor for the nodes that will be chosen in each tree). Therefore, no tree will contain nodes from more than one data center. For this reason, we are able to propagate any message exclusively through the data center from which the message was originated. Some systems can possibly make use of a primitive which allows local-only broadcast. This result could also be obtained by having two instances of HyParView running in each node (one with only local nodes and the other with both local and remote nodes). However, we find it is more convenient to use our approach both in terms of implementation and performance. Each Thicket message contains a flag saying whether we are aiming for a local broadcast or not and it will chose the right Thicket instance to handle the request according to the value of that flag. Local broadcast "mode"will also not broadcast through edge connections, even though edge nodes may have established an edge connection to the data center where the message is being broadcasted.

3.4.2.5 Dissemination

With the introduction of an edge service, a change has to be made when receiving data from a peer. This modification consists simply in broadcasting the same data to the necessary levels of the edge hierarchy. Details on this topic will be explained in section 3.4.4.

3.4.2.6 Missing messages

Even though Thicket has a mechanism of relying in missing messages to repair trees, the protocol specification does not describe any mechanism for recovering those messages. Because of that, in scenarios that imply the initialization of multiple nodes as well as in churn scenarios, many messages can be lost due to constant tree reconstruction. Since we aim at providing an efficient broadcast primitive at the application level, we need any node to receive all the important information whenever it is possible. Due to this, we take advantage of the tree repair procedure to acquire missing data. When a target is chosen to repair the tree, we are able to piggyback a set of missing messages ids in the Graft message. Such a modification allows the potential repairing peer to send the corresponding data, if it knows about it. In spite of allowing data messages to be received even if the tree becomes unstable, such an approach requires saving the history of data messages in each node. Although we remove those messages after 40s, it can still introduce a considerable overhead depending on the size of the payloads.

3.4.3 Global view

With HyParView on its own, the nodes we know are simply the ones in the active view and the ones in the passive view. Many applications, however, require full membership knowledge. It may be important for the application to be aware of all the live nodes (for client connection, load balance, partitioning, replication, etc.), we construct a global view on top of the Peer Sampling Service component. This will then translate into one of our API's functions.

We make use of the functionalities of HyParView and Thicket to be able to construct and maintain a global view of the whole system.

3.4.3.1 Constructing a global view

In the original Cassandra and also in Brown's approach, there is a periodic procedure (set to 1s by default) in which each node tries to gossip to at most three other nodes: a random live node, a random unreachable node and a seed (unless the live node or the unreachable node were a seed). Therefore each node will eventually establish a TCP connection with every other node. These messages contain the states of all the peers those nodes know about. Furthermore, they are also used as heartbeats, for the failure detection component, which means that this approach helps solving two issues: the global view requirement and failure detection. Since the heartbeats use a versioning system, each heartbeat has a bigger version than the previously sent one. When a node starts a gossip exchange (i.e. the gossiper), it sends the list of all the endpoints it knows and the corresponding last received version of the heartbeat through a GossipDigestSyn message. The receiving node (i.e. the gossipee) then compares the versions and requests the outdated nodes states and also sends the list of states which are more updated, through a GossipDigestAck message. The gossiper then replies with the requested endpoint states through a GossipDigestAck2 message. By using this procedure, which is part of the anti-entropy reconciliation protocol Scuttlebutt, the information on every node will eventually converge. Moreover, when a node receives a state referring to a new node, it also sends an echo message and asynchronously waits for the response, to make sure the node is reachable, before marking it alive.

In our approach, we still rely on the gossip exchange mechanism, but we rather make use of the Peer Sampling Service to choose with which nodes those messages are exchanged. Since we rely on HyParView, we can make use of a node's active view to reduce the number of nodes each node gossips with. Therefore, each member of the membership periodically gossips to a random peer from its active view, which contain both local and remote peers. Since HyParView provides a fully connected overlay, by only gossiping with the peers in the active view, each node will eventually learn about the state of all the live endpoints, which are also used to provide the same functionalities as the passive view, as previously described in section 3.4.1. Moreover, whenever a node joins the membership, it makes use of the Thicket spanning tree to broadcast its state. Therefore, the dissemination of this event is much faster than solely relying on the periodic gossip exchanges. As well as in Cassandra's membership, our solution also includes trying to gossip with an unreachable node in the same cycle.

Due to our edge mechanism, we also need to gossip to the contact node in the above layer. However, we do not need edge nodes to be aware of the entire membership, therefore, contact nodes only gossip the states of the nodes in its partial view, by making use of HyParView's active view, with both the local and remote views. This will be further explained in greater detail on section 3.4.5. On the other hand, the contact nodes do not have to be aware of other edge nodes except the ones with which they established an edge connection. This is due to the fact that nodes in lower levels will try to establish connections (if needed) and not the opposite.

Concerning the gossip exchanges with seeds, we added a mechanism that benefits from using partial views. Since one of the goals is to lower the number of open connections, a node x only gossips to a given seed y if there is no peer from y's data center in x's active view, wether the data center is the same one or a remote one. Although nodes always communicate with seeds in the beggining and therefore establish TCP connections with them, due to a procedure of closing unused connections described next, those connections are eventually closed if not needed anymore.

3.4.3.2 Closing connections

Even though we try to minimize gossip exchanges by making use of a node's partial views and by avoiding connections to seed nodes when it is not needed, it is possible that some useless TCP connections (created before, when needed) remain open after some time. It is in our interest, however, that a small number of TCP connections is kept open, since they may slow down processes, due to TCP's mechanisms to maintain connections, such as *keep-alive* messages. In scenarios with large number of nodes, this mechanisms might have an impact in the node's performance and network saturation.

In case a high number of nodes join the system at the same time, many HyParView messages are exchanged. Since each node maintains a partial view of the system in its active view, it cannot accept all the incoming join requests made through TCP connections. Furthermore, in this cases, many HyParView Disconnect messages are sent due to replacements in active views. However, those connections remain open even though the two nodes involved are not peers anymore. One solution to avoid this increasing number of open connections is to reactively close them after a node realizes it does not need them anymore (for example, when such messages are received). However, in case of unstable environments, where we can find many nodes joining and leaving the membership, those open connections are useful to avoid many simultaneous procedures of establishing a TCP connection. Such previously useless open connections may help saving time upon failure recovery, when trying to communicate with backup nodes. Therefore, our approach uses a timeout mechanism by which sockets are closed if not used for more than one minute and fifteen seconds, which most likely means that we are facing a stable environment. We have used that value since it is higher than HyParView's connectivity periodic step, which was set to one minute. On that step, nodes may have to contact other nodes from their backup views. With a lower timeout value for closing connections, this would mean opening and closing the same connections frequently.

This solution also allows seeds to close unused connections and therefore have the same average number of open connection as any other node, in stable environments. Since a node's active view keeps in average *log(n)* peers, we are able to reduce the number of open connections per node from n to approximately *log(n)*, when comparing to Cassandra's membership (or any TCP-based full-mesh membership) and also to Brown's modifications.

Algorithm 3: Data structures

```
1: Local State:
2:     $N_{id}$ //Node address
3:     $N_{datacenter}$ //Node data center
4:     upView //List: Address
5:     downView //List: Address
6:     filter //BloomFilter
7:     filters //Map: $Address \rightarrow$ BloomFilter
```

3.4.3.3 Shutdown mechanism

When a node intends to exit the system on its own, for example by closing the process manually or due to some scheduled procedure, it first communicates its decision throughout the membership. It announces its new shutdown state by using the spanning tree rooted at itself. Therefore, every node will eventually be aware of this state change and will then mark the node as shutdown. Moreover, future periodic gossip state exchanges will contain this new shutdown state of that node, if for some reason (such as failures, resulting in broken tree branches) not every node learned about that through the spanning tree. Furthermore, a node also communicates the new state through its edge connections.

This approach for the shutdown procedure allows each member to quickly learn that a node left the membership. If no solution was provided for this case, nodes would have to wait until the failure detection mechanism considered that node as failed. The longer it takes to detect a failure, the more time we think that a given node is alive and the more time we believe we can use its resources. However, in this case, we can be assured that the membership cannot make use of that node anymore, so it is important to take advantage of such an event.

3.4.4 Edge Service

We will now describe the details of our edge service operation, one of the main contributions of this thesis. The implementation goes according to the hierarchy-based solution described in the previous section.

The same way a node joining the system needs the address of a seed node to initialize the HyParView join procedure, an edge node needs a contact node from a network above its location in the hierarchy (i.e. its parent network), also provided by an external source.

An edge node also runs HyParView and Thicket as a common data center node. Consequently, each network has an overlay network built on top of it, where nodes communicate through it.

Each node maintains a bloom filter [2] and a map of received ones. Each bloom filter is used to keep track of the information to which each node is subscribed. A node being subscribed to a keyword means it will replicate information mapped to that keyword. At first, an edge node inserts in its bloom filter the keywords to which it will be subscribed. An example with a test application will be explained later for better understanding. Bloom filters are used to lower the amount of information that has to be kept by each node. Without having to exchange the keywords directly, we still achieve the same goal.

Bloom filter Bloom filters are space-efficient probabilistic data structures. A bloom filter is basically a bit array that makes use of hash functions in order to test whether an element is member of a set. Any time an element is added, the hash functions are applied to it and the corresponding bit array positions (the results from applying the functions) are set to 1. To check if an element may be present in the set, the same functions are applied and the query returns true if and only if those positions are set to one. This probabilistic method has a set error rate depending on the number of used hash functions. False positives are possible, although false negatives are not. This means a query to a bloom filter will never falsely return that an existing member is not on the set. On the other hand, as false positives are possible, an answer of *true* shall be seen as "possibly in the set". The probability of mistake can be negligible if the proper number of hash functions is used. To properly make use of bloom filters, we use Google's Guava library for Java, which automatically calculates the adequate number of hash functions to be used given the maximum desired false rate. The use of bloom filters as data structures makes this approach really lightweight, due to how information is stored in a bit array, allowing faster dissemination of information.

3.4.4.1 Join Mechanism

When an edge node tries to join the system, in addition to starting the HyParView join procedure, it will first load its bloom filter and then send an EdgeJoin message to the specified contact node from the above level in the hierarchy, as described before. The contact node will then answer with an EdgeJoinResponse containing the map of bloom filters it has received until the moment. The local node will then use the broadcast protocol (Thicket, in this case) to disseminate that same response through its network, so the other nodes can update the filters information as well (this allows every node to keep track of the location of application data). The whole procedure is repeated periodically, to make sure every node has updated information as well as to ensure that nodes that recently joined the membership have access to that as well. Moreover, any node in the hierarchy is subscribed only to a subset of the keywords to which the nodes in the level right above it are subscribed. The top level contains all the information existing in the system.

Each edge network can have only one connection to the superior level. On the other hand, two different edge networks in the same level can maintain an edge connection two the same data center, and even to the same node.

Algorithm 4: Edge operations

```
1: every $T$ seconds do
2:     fillFilter(filter, keywords)
3:     Send(Join, contactNode, filter, $N_{id}$, $N_{datacenter}$)

4: upon $Receive(Join, filter, newNodeId, datacenter)$ do
5:     filters ← filters ∪ {(newNodeId, filter)}
6:     Send(JoinResponse, newNodeId, filters, $N_{id}$, $N_{datacenter}$)

7: upon $Receive(JoinResponse, remoteFilters, node, datacenter)$ do
8:     Thicket.broadcast(JoinResponse, remoteFilters, node, $N_{id}$)
9:     filters ← filters ∪ remoteFilters

10: upon $Receive(Data, data, root, node, datacenter)$ do
11:     Thicket.broadcast(Data, data, $N_{id}$)
12:     foreach $address \in$ upView do
13:         if $address \neq$ root ∧ address ≠ node then
14:             Send(Data, data, address, root, $N_{id}$, $N_{datacenter}$)
15:     if $interested(data)$ then
16:         foreach $address \in$ downView do
17:             if $address \neq$ root ∧ address ≠ node ∧ interested(data, address) then
18:                 Send(Data, data, address, root, $N_{id}$, $N_{datacenter}$)

19: procedure $interested(data, address)$ do
20:     keywords ← getKeywords(data)
21:     foreach $word \in$ keywords do
22:         if $filters$[address].contains(word) then
23:             return true
24:     return false

25: procedure $interested(data)$ do
26:     return filter.contains(word)

27: procedure $broadcast(data)$ do
28:     foreach $address \in$ upView do
29:         Send(Data, data, address, $N_{id}$, $N_{datacenter}$)
30:     if $interested(data)$ then
31:         foreach $address \in$ downView do
32:             if $interested(data, address)$ then
33:                 Send(Data, data, address, $N_{id}$, $N_{datacenter}$)
```

3.4.4.2 Dissemination

Any payload needed to be sent by the application is encapsulated in an EdgeData message. Every EdgeData message is propagated to the upper levels of the network hierarchy (Algorithm 4, line 12), until the data center level, which contains/is interested in all the information, although not always propagated to the levels bellow (Algorithm 4, line 17). For this reason, each node has to maintain two data structures: one for contact nodes (which will be only one at the time, but we'll keep it as a collection for further improvements) and another one for nodes in a level bellow to which it maintains an edge connection. Having access to each level's bloom filter, any node is able to understand if lower levels are interested in receiving some payload. There is no need in sending those levels a message they are not interested in. For instance, the dissemination of a message coming from a node in the data center can stop at some level in the hierarchy or not even reaching any level at all (excluding the data center). Additionally, if a receiving node understands that it is interested in a given payload received inside an EdgeData message, the payload is extracted and encapsulated in a ThicketData message and it will then be disseminated through the respective tree. In the same way, any node that received a ThicketData message will extract the payload and encapsulate it in an EdgeData message and send it through its edge connections (if the other ends are interested), if any.

3.4.5 Failure detection

Although HyParView protocol relies on TCP as an unreliable failure detector mechanism, we find that using it together with an accrual failure detector is suitable to a large-scale system. We use φ accrual failure detector, as already used by Cassandra. Due to its adaptiveness to network performance, workload and historical conditions, we are able to provide a better accuracy on failure detection.

3.4.5.1 On HyParView and Thicket

In spite of Brown's approach making use of HyParView, each node still gossips with every live node in order to check whether they are dead or alive, as well as to create the global view. Periodically, each node gossips to a random live node and tries to gossip to an unreachable member (with some probability to check if it is back up). This gossip messages contain the state of all known nodes. So the gossip process for failure detection tracks node's state both directly or indirectly. It means that the accrual failure detector implementation considers heartbeats received directly from the node or received by other nodes, which share the information. Therefore, a node is only considered as dead if no node can reach it. Everytime a new heartbeat is gossiped, the version of the endpoint state is updated. This version is local, so it allows two nodes to learn which one of them has the most updated information about some endpoint. This heartbeat versioning mechanism was already used in Cassandra, as described before.

As explained in section 3.4.3, in our approach, nodes periodically gossip with a random peer from the active view (instead of gossiping with a random peer from the set of live endpoints of the whole cluster). Due to the connected overlay provided by HyParView, we can still monitor every node even though each node only monitors a small part of the membership. Furthermore, since each node maintains a symmetric relationship with each peer in it's active view, a node's state is monitored only by those other nodes which monitor itself. Otherwise such an approach would be much more difficult to implement. When a node fails, every peer in its active view will be able to identify the failure and remove the failed node from the active view (furthermore, the node is removed from the set of live endpoints and added to the set of unreachable nodes). The other nodes, that are not connected to it, will also eventually be able to learn that it has failed, due to the gossip subsystem described in section 3.4.3.

As previously explained, due to the periodic exchange of the full cluster knowledge, if two nodes x and y cannot communicate but still both of them get information saying the other node is alive, they cannot consider each other dead. A physical link failure between both nodes my cause such scenario. The TCP connection can achieve its timeout and report both x and y that the connection is broken. Both nodes will then remove each other from their own active views, altough they will not remove each other from the set of live endpoints since consistency between all nodes must be achieved (and other nodes can reach both x and y, so they are alive). As they are removed from each other's active views, the trees are updated by the Thicket protocol.

The φ accrual failure detection has high accuracy levels, due to its capability of adapting to external conditions. This means that the likelihood of mistakenly suspecting a node is low. However, if a suspected node was in fact alive, but performing poorly, it may still receive the Disconnect messages sent to it after its active view peers considered it dead. When a Disconnect message is received, the receiving node sends neighbor requests to nodes from its backup view. In this case, the misjudged node can eventually be part of the membership again. Even if for some reason, those Disconnect messages are not received, the periodic procedure to check full connectivity (previously described) will make the misjudged node contact some backup nodes.

3.4.5.2 On the Edge Service

The edge service also benefits from the φ accrual failure detector. When a connection is established with an upper level node, that node shares information about the state of some other nodes in its level. Whenever the detector considers the parent level node as suspected, the edge node in the lower level must try to establish a connection with an alive node from the parent level. The edge service makes use of the peer sampling service partial views to provide a backup view for edge nodes: a node's partial view (in HyParView's case, the active view, which contains both the local and the remote partial views) and a seed's address is shared with the edge nodes connected to it, and then disseminated through those levels' spanning trees, starting in the receiving nodes. In order to take into account networks with NATs and firewalls, the lower levels always have to initiate connection with upper levels when trying to recover from failures, and not the opposite, as explained before. Due to that requirement, it is the lower level's jobs to initiate failure recovery, by trying to establish connection to one of those backup nodes from the contact node's active view, and lastly to the provided seed, in case the others are unreachable. Although it is highly unlikely that all peers from the active view fail at more or less the same time, together with the contact node, it is important to keep a seed in that backup view, since the membership assumes seeds are always alive. The lower level node also sends an EdgeDisconnect message so the previous contact node does not keep broadcasting any new information to it, because it may still be alive but performing poorly.

On the other side, when there is a failed node on the other end of the edge connection, i.e. when the node in the lower level fails, the peers in its partial view (from the peer sampling service) will try to re-establish connection between the two levels. Since the partial views are symmetric, a node's peers will know if they are contained in its partial view. Each of the peers will then try to connect to the same exact upper level contact node that the failed node was connected to. When the contact node receives a EdgeJoin request from a new node, it will verify if it suspects the previous one as failed. If so, it will then accept the request and continue with the normal procedures for the edge join mechanism. Therefore, the first node to have its message arriving at the contact node, will establish the bridge between the levels.

3.4.5.3 Extensibility

Other failure detection algorithms, including others that follow the rules of an accrual failure detector, such as [39], could easily replace φ accrual failure detector. *IFailureDetector* is an interface which allows any application using our membership service to query the liveness of a given node through simple methods. Therefore, we provide extensibility to the service, allowing other failure detectors to be implemented if needed.

3.5 API

Our Java API provides a well-defined set of functions. This API a key contribution of this thesis, as it is important that the main functionalities are mapped into methods that can be used by applications needing a membership service which allows edge connections.

- **initServer()** Allows each underlying membership service (HyParView, Thicket and Edge Service, etc.) to be initialized.

- **shutdown()** Shuts down any underlying membership service running in the local node.

- **getMembers()** Returns an iterator for members of the membership of which the local node is aware of.

- **getPeers()** Returns an iterator for all nodes in both the local and remote active views.

- **getLocalPeers()** Returns an iterator for the nodes in the local active view, i.e. only nodes from the local data center.

- **broadcast(IPayload payload)** Broadcasts a given payload both through the tree and edge connections, in order to reach every end of the membership.

- **datacenterBroadcast(IPayload payload)** Broadcasts a given payload only through the local datacenter.
- **interested(IPayload payload)** Returns true if the local node is interested in a given payload, considering its keywords. Otherwise, returns false.
- **where(Object keyword)** Returns one address which is interested in the given keyword.
- **addListener(BroadcastServiceClient listener)** Adds a listener to Thicket service as well as to Edge service.

IPayload defines an interface with only one method: *getKeywords()*. Any ad-hoc payload object must implement *IPayload* and in consequence implement *getKeywords()*, to be used by the Edge service to check whether a node is interested in the data.

BroadcastServiceClient<IPayload> defines an interface for listeners of messages received through broadcast. Such a listener must implement *BroadcastServiceClient.receive(IPayload payload)* to specify a behavior upon receiving a message either through the tree or through an edge connection.

3.6 Summary

This chapter presented a membership service for edge systems with a notion of multiple data centers. We described its subcomponents and how they work together to provide the necessary properties of a scalable membership. Our solution makes use of existing protocols that we have adapted and also of an edge component that allows to keep information on nodes from other networks and disseminate information through edge connections.

We also presented an API which relies on our membership service. This API allows applications to access membership information, such as the global view, to use the edge component, and also to specify different message handling behaviors depending on the application data being exchanged throughout the dissemination components.

CHAPTER 4

Evaluation

In this chapter we present and analyze the results of our experimental work, comparing our final solution with others. We also have deployed a client-server application that makes use of our API to meet its requirements. We describe and evaluate the application and also suggest some improvements to similar applications when using our API.

4.1 Experimental work

In this section we start by describing our setup, which we used for all the tests with our membership service. We then present our results and analyze them.

We compare three approaches in this chapter: our solution (MS-DCS), our solution but with no data center distinction (MS-DC1), and Cassandra's membership (CM). MS-DC1 solution simply removes the use of remote views from the solution and handles nodes as if there are no different data centers (multiple data centers are treated as one). In order to make a fair comparison between the different solutions, we do not use any edge nodes in the first set of experiences. After the comparison between the three solutions, we then evaluate some properties of our membership when including edge nodes, as it is one of the main goals of our work.

4.1.1 Experimental setup

We have run our experiments in Microsoft Azure, a well-known platform which provides cloud computing services. As our membership service aims to take into account the use of different data centers, we have used 4 different data centers, located in Netherlands, the United States, France and Canada. In each data center, we have created a virtual network.

In order to simulate a scenario of a large-scale system, we wanted each data center to have a number of nodes relatively greater than the expected number of peers in a node's partial view, in order to be able to test the peer sampling service with great reliability. However, we had to limit to number of nodes per data center to 10 due to budget limitations on using Microsoft Azure. With this number of nodes, each node is expected to have less than 4 peers in its local active view, which represents the fanout value. We then used 10 *A2 v2* virtual machines in each data center to emulate each of the 10 nodes, by running an instance of our membership in each one (with the same source code). An *A2 v2* type virtual machine has two CPU cores as well as 4 gigabytes of RAM, which we find suitable for our experiences.

Each node was provided with the public address of four seeds, one from each data center. The seed list is the same in every node. Each node also knows its own public address beforehand. Port 7000 (the default port for our membership) had to be previously open, for TCP connections, for membership nodes to be able to receive packets from the other virtual networks.

For the following tests, unless stated otherwise, we started the membership service at the four seed nodes. After thirty seconds, we started the service in the remaining 36 nodes. However, since we access the virtual machines through *ssh*, the starting times can vary as it depends on the latency between the local experience orchestration machine and the data centers. Furthermore, we have disabled the shutdown notification dissemination mechanism so we could simulate real failures. Otherwise closing a node would simply notify the membership through its tree.

4.1.2 Multi-data center

4.1.2.1 Open connections

We first present our results for the average number of open TCP connections throughout time of in experience. We have separated this information in Fig. 4.1 and Fig. 4.2 for both seed nodes and non-seed nodes respectively.

As for seed nodes, we can observe that after we have started the other 36 nodes, the number of open connections starts to increase quickly. Due to our mechanism of preventing excessive gossiping to seed nodes, by checking if a node is already gossiping to that data center through its partial view, we are able to reduce the number of open connections when comparing to Cassandra. In Cassandra's membership, as expected, the number of open connections is constant over time after it reaches the total number of nodes (minus one, itself) for the first time. In this case, that value is 39. Both MS-DCS and MS-DC1 solutions present roughly the same number of open connections over time. One minute and fifteen seconds (75 seconds) after we have initialized the other nodes, we can see that the number of open connections starts to drop. Eventually, the number of open connections reaches the number of peers in a node's partial view, which is smaller in MS-DC1 since $log(40)+1 < log(10)+1+3$. The value $log(N)+1$ represents the average number of nodes in local partial views, as explained in the previous chapter. Since in MS-DC1 there is no distinction between data centers, then $N=40$, while in MS-DCS $N=10$. The constant 3

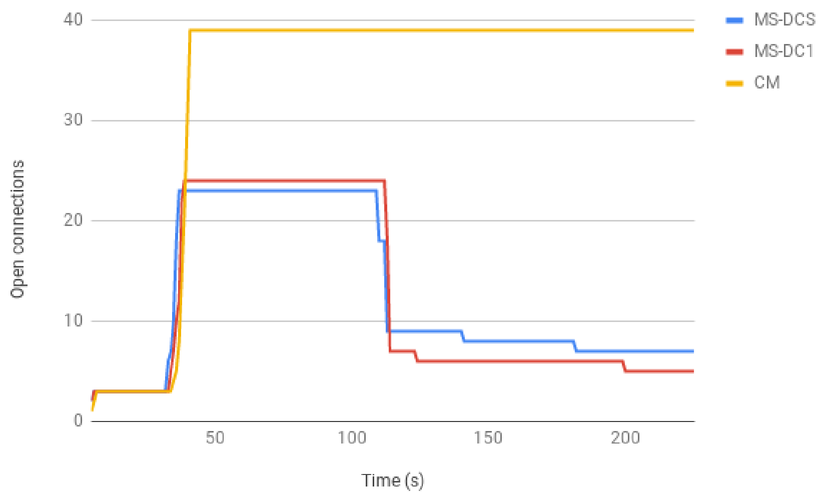

Figure 4.1: Open TCP connections through time in seed nodes

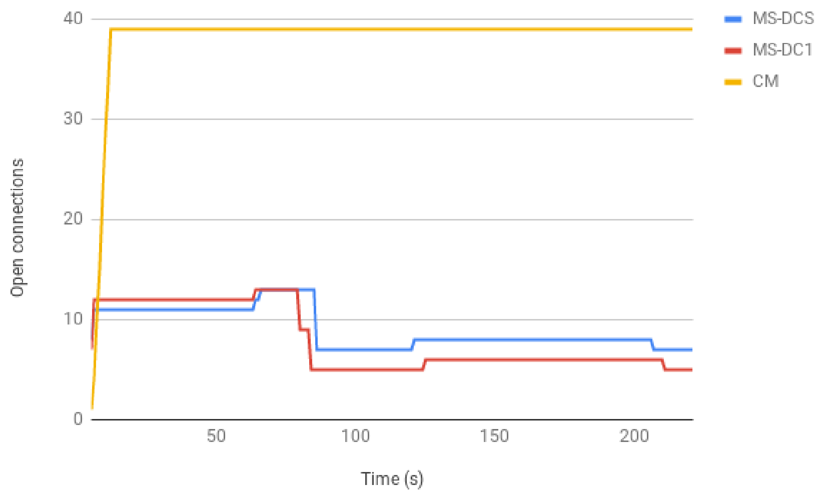

Figure 4.2: Open TCP connections through time in non-seed nodes

refers to the number of peers in the remote active view, which corresponds to one per each remote data center, which is only applied to MS-DCS.

In non-seed nodes, i.e. the remaining 36, the behavior is not significantly different. However, in both MS-DCS and MS-DC1, nodes reach a smaller number of open TCP connections than in seed nodes. That number is still higher than the number of peers in partial views due to the constant replacement of nodes in those views in scenarios with lots of nodes joining the overlay at the same time, as it is the case. We can also verify that the number of connections may even increase and decrease again. This is due to the periodic mechanism for ensuring node's full cluster connectivity: when a node considers its partial view is too small considering the total number of nodes, it sends a Neighbor request to a node from its passive/backup view.

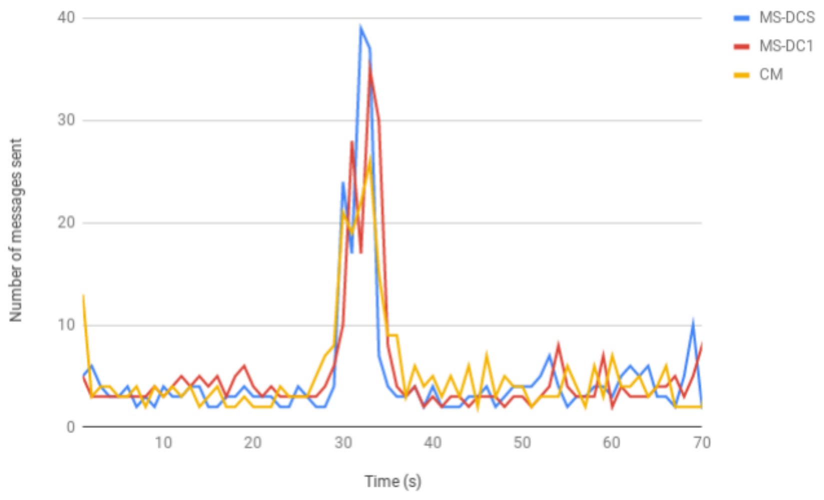

Figure 4.3: Messages sent per second in seed nodes

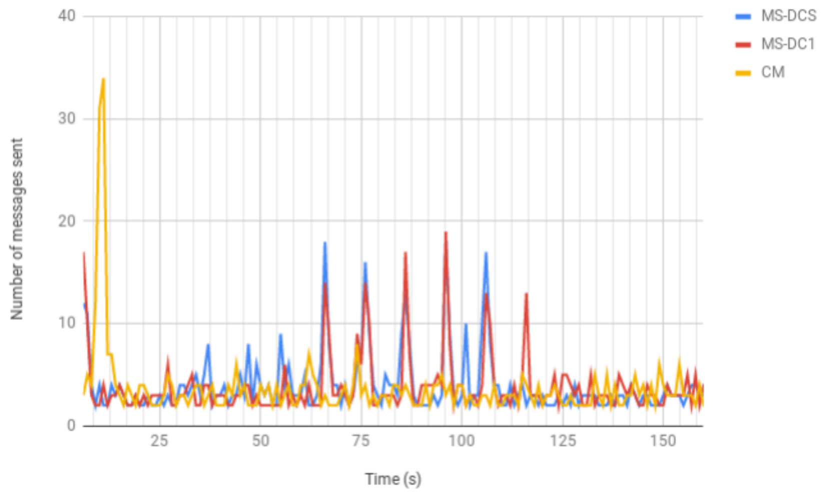

Figure 4.4: Messages sent per second in non-seed nodes

Although latency is not influenced by this number of open connections, we are preventing eventual network saturation in large-scale scenarios, with data centers with thousands of nodes.

4.1.2.2 Messages sent

In this experiment we have measured the number of messages sent (to maintain the membership and dissemination components) by each solution over time. We are also able to see how this number changes when a high number of nodes join the membership. This measurement is highly relevant since too many messages may slow down other important processes in the machine.

As the results show in Fig. 4.3, CM introduces more messages sent per node right after start up. This is due to the echo messages (both the requests and the confirmations) sent

	MS-DCS	MS-DC	CM
Global cluster information	1.8	1.8	4.7
New node information (average)	0.60	0.80	5.3
New node information (maximum)	0.67	0.9	8.7

Table 4.1: Times (s) for global view construction when joining node

to confirm nodes' reachability after learning about them. When the other 36 nodes are joined, the number of messages sent per second increases quickly regardless of the used approach. While in the CM solution, this is due to the echo messages, in MS-DCS and MS-DC1 solutions, this is due to the state propagation mechanism that is triggered in each of the 36 nodes, which originates more sent messages. Roughly the same number of messages is distributed in both MS solutions through seconds 30 to 35.

Fig. 4.4 shows that non-seed nodes send a lot of messages in CM a couple of seconds after joining them, due to the echo messages. Since a node takes longer to be aware of the whole cluster in CM than in both MS solutions, those messages will only be sent 3 or 4 seconds after the nodes were joined (as we will show and explain in detail in the test in section 4.1.2.3). In both MS-DCS and MS-DC1, the messages concerning nodes' state propagation through periodic gossip exchanges are sent earlier and are a lot less than in CM. This is due to the fact that through HyParView's joining procedures, nodes will be aware of others much faster.

There is no significant difference in terms of the number of messages sent between MS-DCS and MS-DC1. On the other hand, we can observe that a periodic increase of that number. This is due to tree construction, which requires SUMMARY and GRAFT messages to be sent periodically. Although they are not needed for the construction of the global view, that procedure accelerates eventual data dissemination, as detailed in 3.4.2.3. After every tree is stable, all three solutions present roughly the same number of messages per second. CM presents slightly more messages, since nodes still gossip to seeds if they have not gossiped to one on that same second.

4.1.2.3 Join

As one of our goals is to quickly make nodes aware of changes in the membership, we joined a node in a stable environment. We firstly started 39 nodes and after a couple of minutes we have started the 40th node. The 40th node was the same in the tests for the three approaches. One goal of this test is to measure how long it takes for a node to be aware of the whole membership (knowing the state of every other node). The other goal is to measure how long it takes for every other node to learn about the new node.

The first row of Table 4.1 refers to the time it took (in seconds) for a node to learn the state of the whole cluster. The second row is the average time it took for other nodes to learn about the state of the new one. Finally, the third row represents the time it took to the last node receiving the update to learn about the state of the new node.

In both MS solutions, the new joining node is able to learn about the whole cluster more than two times faster than the CM solution. As soon as the node starts, it tries to reach a seed. Quickly after, it will have new nodes in its active view, which is symmetric. For that reason, at least one of its new peers will likely (since the choice is random between the peers of the active view) gossip the cluster state on their next periodic gossip exchange, set to trigger every second. On the other hand, in Cassandra's solution, the new node has to first gossip to a seed, which will mark it as alive. In the next of seed's gossip cycle, it will gossip to a random live node, which means it can either gossip the cluster state to the new node or gossip with another node, already including the state of the new one. The joining node then has to wait one of the nodes to initiate a gossip exchange with it. The relation between the time it takes and the number of nodes in the cluster is then logarithmic and also influenced by the gossip cycle timer.

We can also verify a big difference between the MS's and the CM solutions while measuring how long it takes for each node to be aware of the joining node. As we are making use of our broadcast service built on top of Thicket, the dissemination of the update is much faster than Cassandra's periodic gossip approach. There is also a slight difference between the times of MS-DCS and MS-DC1 solutions. This is due to the fact that MS-DC1 does not use any criteria relative to the physical distancy (i.e. the data centers) for choosing peers for the active view. In consequence, a node might propagate the update to too many remote peers and too few or none local peers. Therefore, the update message might travel many times between data centers, taking longer to reach every node.

4.1.2.4 Failure detection and network partitions

For this test, we have simulated a cross-continent network partition between Europe and America, separating the data centers in Netherlands and France from the ones in Canada and the United States. This simulation was done through an adaptation of the source code, by making each node ignore messages received from the data centers from which it will be separated, after a given time. Since connections are closed after a while if they are not being used, the TCP sockets between the unreachable data centers will be closed.

We first started the 40 nodes. After 150 seconds, we introduced the network partition between continents. Fig. 4.5 represents the average number of live nodes through time from the point of view of a single node. The dots represent changes in the number. We can note after we have introduced the partition, it took from 20 to 25 seconds until every node from the other continent was considered as unreachable and therefore dead. The average time of failure detection depends on how often a node is reported as live which in turn depends on the timer for gossip exchanges.

After 100 more seconds, we have removed the network partition. Each node took approximately one second to completely recover from it, i.e. too mark the nodes from the other continent as alive. Since there is an attempt to gossip with an unreachable node

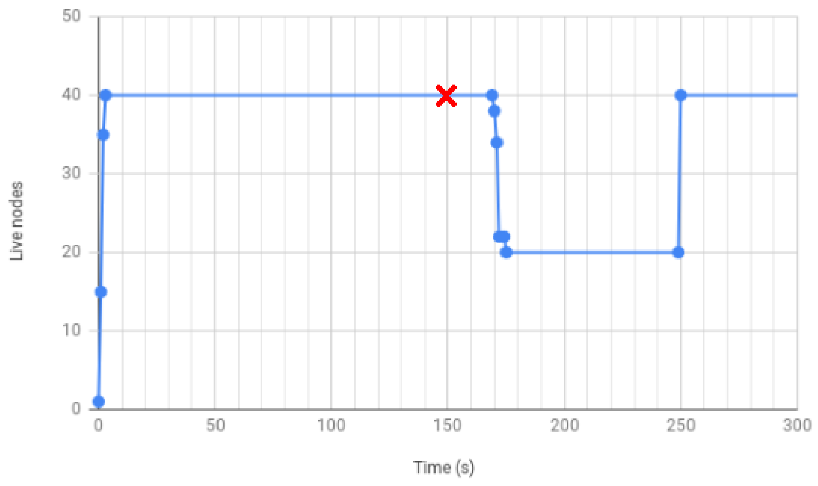

Figure 4.5: Number of live nodes from the point of view of one node

every second, nodes will exchange the states they know with only one gossip exchange.

These results do not depend on which of the three solution is used, since the same mechanism is implemented without any differences in all of them.

Additionally, even though it is not likely that every peer in a node's partial view fails at more or less the same time, such scenario may happen. In that case, any message originated by that node before it detects the failures will be completely lost. Since the recovery summary mechanism in Thicket makes use of a backup view that is derived from the partial view, the node will try to send both the message and later the summary to members of its partial view, which are inactive. One solution for this issue would be retrying to send the message when connected to another node, although it could require keeping the data during longer periods of time.

4.1.3 Edge Scenario

4.1.3.1 Experimental setup

For the tests concerning the edge functionality, we used the same data center configuration as in the previous tests. We also had two main edge networks protected by firewalls and NATs. The network in *dce1* has only one node. The network in *dce2* has four nodes, in which two of them were simulated to be inside a network contained in *dce2*. The node in *dce1* uses a node from *dc1* (Netherlands), while *dce2* uses a node from *dc4* (Canada) as contact node. *dce2.1* also uses one of the nodes of the outer network as a contact node. We made sure that the chosen contact nodes in *dc1* and *dc4* were not connected to each other directly (i.e. with a TCP connection), but rather only through the overlay. In order to send data messages through the whole overlay, we have implemented a simple application that sends *string* objects. Finally, since edge networks are subscribed to keywords, as we previously described, so they retain only some part of the data exchanged,

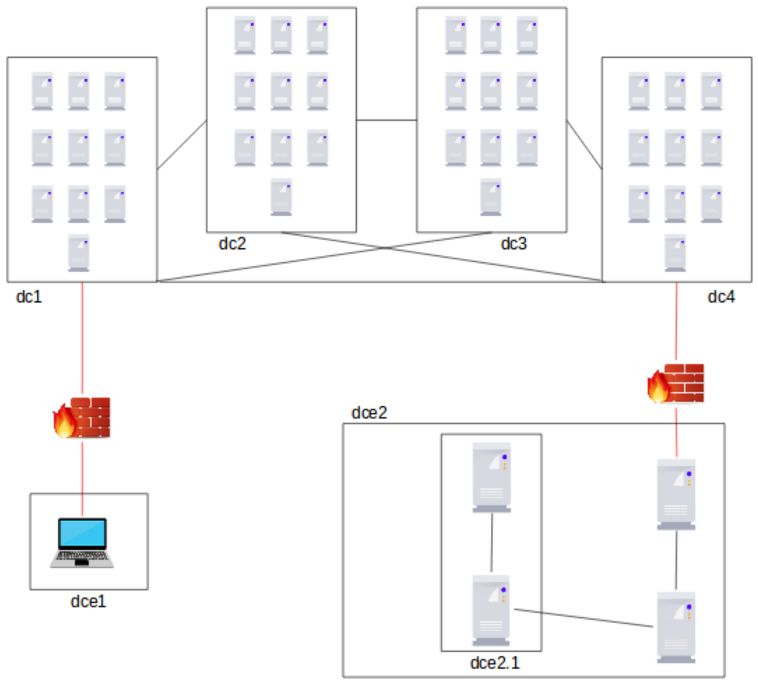

Figure 4.6: Initial setup for edge tests

we have subscribed each edge node to the keyword "test" only. This was done by using the configuration file in each node. A more complex application using our API was also implemented and will be described and tested for correctness in section 4.2.

The machines used for the whole *dce2* edge network have four CPU cores as well as 16 gigabytes of RAM. The machine used for the *dce1* was a personal computer with two CPU cores and 3 gigabytes of RAM.

4.1.3.2 Join

An edge node tries to communicate with the specified contact on start. If it fails, it'll try again after 10 seconds. In this tests, we made sure the contact nodes were initiated before the lower level nodes.

We measured the average time it took to exchange data messages with the contact nodes. To be able to obtain a representative value, each edge node connected to a contact node executed an edge broadcast every second. Every edge message is then answered with an ACK, whose time of reception is used to calculate the latency.

The latency of the first request from the node in *dce1* to the contact node in *dc1*, Netherlands, was 476ms in average. Basically, this means establishing a TCP connection to the contact node, sending it an EdgeJoin request and receiving an EdgeJoinResponse. For the contacting node in *dce2*, it took 386ms until receiving the EdgeJoinResponse from the contact node in *dc4*, Canada. On the other hand, it takes 70ms for the node in *dce2.1* to receive the EdgeJoinResponse from *dce2*. This was much faster since both *dce2* and *dce2.1* are in the same physical network.

4.1.3.3 Latency and Failures

Data messages subsequent to the join procedure take, in average: 43ms for the communication between *dce1* and *dc1*, 133ms for the communication between *dce2* and *dc4*, and finally 2ms between *dce2.1* and *dce2*.

Data messages take in average 193ms from *dce1* to *dce2.1* and vice-versa. However, they do not necessarily take the same time (in average) in the same run. In one of the runs, data messages sent from *dce1* took 195ms to reach and be processed by *dce2.1*. On the other hand, messages sent by *dce2.1* took in average 120ms in the same run. This is due to the fact that different trees are used to disseminate the information. When the node in *dce1* sends a message and contacts the contact node in Netherlands, the contact node used the tree rooted at itself to disseminate the information. The same happens for the contact node in Canada when receiving data from *dce2*. Since the trees are different, it results in different dissemination times.

After measuring those values, we registered one active view from the contact node in *dc1* and another one from the contact node in *dc4*. We also made sure those two nodes were not in each other's active views. We then stopped every other node. As the two remaining nodes are not in each other's active view, they will not receive any information about each other via gossip, therefore they will eventually consider them dead. As we previously described, each node attempts to communicate with a random unreachable node every second. Since 38 data center nodes are now dead, it can take a while until they attempt to communicate with each other.

As soon as the edge nodes detect the failures in the contact nodes, which can take up to 20 seconds, they attempt to establish an edge communication with the backup peers (from the contact's active view) previously received. However, until the two data centers' contact nodes consider each other live, no data can be exchanged between them, and in consequence, between *dce1* and *dce2*.

After total recovery, there are two nodes in the data centers and two edge networks. We were able to measure that messages between *dce1* and *dce2* took, in average, 120ms to reach the other edge data center. Since there are only two data center nodes operating, the data center trees are symmetric, which means that latency is the same for both ways.

4.1.3.4 Impact of membership changes in edge connections

We have also measured the impact of a large number of changes in the upper level of the overlay in the communications between the edge networks and the contact nodes.

Firstly, we started only the contact node in *dc4*. After 20 seconds, we started the node in *dce2* that was going to communicate with the contact node. $t = 0$ represents the moment that the edge node was started. At the $t = 50$ we have joined the other 39 data center nodes. Finally, at $t = 240$ we have stopped those. We repeated this test 5 times and calculated the average latency or each t. As we can see, the trend line in the Fig. 4.7 is going down with time. The sudden increases in latency can be verified not only while the 39

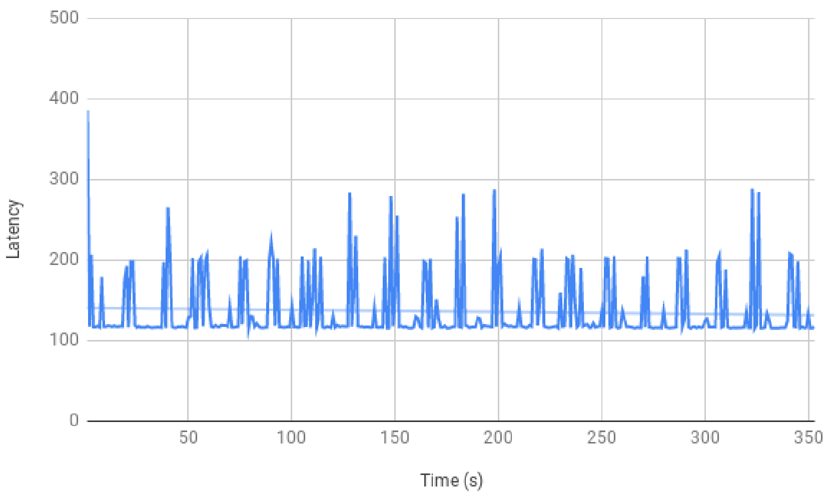

Figure 4.7: Latency in the communication to a contact node

nodes were up but also in the beginning of the run and some time after they were stopped and after they were detected as failed. Although those higher values can be reached even in stables environments, periodic tree construction and repairing mechanisms may have a slight impact in the latency with a much higher number of nodes. During these runs, we have registered an average latency of approximately the same value when compared to a run with the same number of nodes but without membership changes on-the-fly.

4.2 Application example

4.2.1 Architecture

In order to test the functionalities provided by our API as well as our edge service correctness, we implemented a simplified version of Twitter.

Each server node is part of the membership and also handles client connections. Additionally, it is subscribed to a given set of keywords. As we explained before, a node has to be subscribed only to a subset of keywords from the keywords of the nodes in its parent level. The keywords used in this example will be hashtags (e.g. *#fct*, *#lisbon*, etc.). A data center node will be subscribed (and therefore interested) to all the used hashtags. An edge network may then be subscribed to the hashtags *#lisbon, #unl, #fct*, and an edge network inside the first one may be subscribed to *#fct*. The set of keywords to which a node is subscribed is provided beforehand in each node's configuration file. Therefore, it will keep track of every tweet containing those hashtags, maintaining a structure that maps hashtags into lists of tweets.

4.2.2 API usage

Before calling *initServer()* in a server node, a listener has to be added using *addListener()*. The listener class to which we called *GossipTweetListener<Tweet>* implements *Broadcast-ServiceClient<IPayload>*. It will then implement *receive (Tweet t)*. In our application's case, upon receiving a tweet in which it is interested (*interested(tweet)*), the node will just update its map of hashtags and tweets and then send the tweet *t* to the clients with whom it has an opened connection. The dissemination through the membership is handled elsewhere and shall not be the concern of the application.

Each client communicates with only one server at a time. A client has two possible operations: tweet and search.

- **tweet(Tweet t):** sends *t* to server.
- **search(String hashtag):** asks server for all the tweets containing *hashtag*.

Whenever a server receives a tweet from the client (class *Tweet* — which contains a String, a timestamp and a list of keywords in the string — implements *IPayload*), it uses the API function *broadcast()*. The tweet will then reach every other server node which is either interested in the payload, in the same overlay as that server, or higher in the hierarchy. Consecutively, if a node received the disseminated tweet and it is interested in it, the tweet will reach any client connected to that same node. Otherwise it will just relay the message.

Whenever a server receives a search request, it will first try to understand whether it is subscribed to *hashtag*. If so, it will just send to the client all the tweets it stored with that hashtag. Otherwise, it will use the result of the API function *where()* to tell the client the address of a server which is subscribed to that keyword. Upon receiving the address, the client's behavior consists solely in closing the current connection, opening connection to the referred server and automatically repeating the operation.

4.2.3 Correctness

We have used our sample application to verify the correctness of our edge service. We have used the configuration files to specify to which keyword is each edge node subscribed. We have used the following configuration:

- *dce1* - "#lisbon, #portugal"
- *dce2* - "#lisbon, #unl, #fct"
- *dce2.1* - "#fct"

We then connected one client to the node in *dce1* and another client to the node in *dce2.1* that is directly communicating with *dce2*.

We verified that sent messages that did not include any of the specified keywords were not displayed for the other client. Messages sent with the keyword "#lisbon", on the other end, would only be displayed for the client in *dce1*.

We then used the *search* feature in the *dce2.1* client. As we searched for the keyword "#lisbon", the server node to which the client was connected verified that such keyword was not in its domain. Thereafter, the server answers the client with the IP address of its own contact node, which contains that information. As the client disconnects from the previous server node, contacts the new one and repeats the request, a list of previously sent tweets with the hashtag "#lisbon" are now retrieved to it. After that, any message with keywords in *dce2*'s domain is received and displayed to the client. The same happened to the client in *dc1* when we *searched* for the *hashtag* "#europe". In this case, that client received the IP address of the contact node in *dc1*. The client then closes the open connection and makes the same request to its new server, which retrieves the requested information. Thereafter, since the data center level's domain contains every possible keyword, any subsequent messages, regardless of the keywords, are received by that client.

4.2.4 Improvements

Clients in an edge network are expected to request data contained in the server to which they are connected. As we described, if that is not the case and a client searches for something else (even for only once), a new connection has to be made. This scenario can repeat itself, as we keep searching for other information. As we can see given this implementation, the system will eventually converge to a state where every client is connected to a node in the data center. Although this is a problem that concerns the application itself and not the membership service we described, we still propose three main solutions for this, which can even be used together:

- Keeping track of each client's request pattern. As we explained, a client that was once connected to a lower level of the network hierarchy may have requested some information that made him connect to a higher level. If that client goes back to the previous request pattern where he requests information contained in the lower level, we can close the current connection and open another one with a node from that level.

- Keeping track of the general request pattern. If a server receives n requests containing the keyword k, it can subscribe to that keyword and start replicating the corresponding information. However, this implies that the higher levels also subscribe that keyword. This will eventually converge to a scenery where every node is interested in every payload, increasing network traffic and node overload. Nonetheless, this can be mitigated using a maximum number of keywords per node.

- Not closing connections. When needing to connect to a higher level node, the client simply keeps the current connection opened. The client can then use one of the opened connections when requesting. Although this allows to keep fast access to the server when possible, we still don't mitigate the convergence problem.

4.3 Discussion

In this chapter we were able to evaluate the most important characteristics of our membership service.

The comparison between MS-DCS and MS-DC1 solutions showed us that, even though we have obtained similar results in most of the tests and only slightly better in others, we can still recover from network partitions as fast and maintain a stable membership with only one connection (per node) to each remote data center.

In the same way, we have showed that our membership can fulfill Cassandra's membership requirements in spite of relying only in partial views. Despite of not being able to show a significant change in the drop of open connections, such mechanism may help in larger-scale scenarios. We took advantage of Cassandra's membership best mechanisms and adapted some others to efficiently maintain a membership with partial views. Moreover, we were able to present significantly low times for nodes joining, which can be very helpful to quickly construct a global view in each node.

We have also verified the correctness of our edge component, which relies on the data center overlay to share information across the nodes and to the edge nodes. Changes in the membership had no impact in the latency of edge message exchanges in our experiments. However, highly larger number of nodes will probably cause an increase of the time per request in edge connections.

Chapter 5

Conclusion

5.1 Discussion

In this work we proposed a membership service that uses the notion of multiple data centers for its own advantage and also introduces an edge service to be used alongside it. Finally, we also defined an API that can be used by large-scale systems that may or may not require edge functionalities.

We based our solution for the data center overlay in a variation of Cassandra's membership, which we have corrected, completed and improved and to which we added new functionalities, to be mapped into the API's functions. Due to the protocols we have used through the work, our membership service ensures reliability, scalability, load-balancing and fault-tolerance.

We have shown that our membership service is able to quickly propagate updates even though it maintains only a small portion of open TCP connections, even in seed nodes, when comparing to Cassandra's full-mesh membership. The broadcast layer used to propagate such updates is also usable for application data, allowing it to quickly reach every node in the cluster. Furthermore, the use of partial views increases the scalability of the membership service, allowing an increase in the number of used nodes.

The overlay of the whole system, when including edge networks, was set to behave like a hierarchy, with what we call "edge networks" inside other edge networks (such as department networks inside a university network), in which the top of the hierarchy is the data center level. In those networks, devices such as server machines or personal computers can be used as system nodes, to replicate certain parts of the data in the whole system. This allows data which is usually accessed to be closer to those other devices in the network. We also have shown that, although we have to deal with firewalls and NATs, our membership avoids that problem by making sure the nodes lower in the hierarchy are the ones establishing connection.

Our API provides the functionalities which we find the most important for applications requiring this type of services. The methods were aimed to be clear and the integration is meant to be as simple as possible.

5.2 Future work

We now discuss possible improvements or new features that can be added to our membership services. We make some assumptions in chapter 3 in terms of the hierarchy. However, our approach would benefit from removing such assumptions.

Edge networks from different locations should be able to connect between themselves to fasten dissemination. However, in order for those networks to establish a connection, this requires some mechanism for relaying or hole-punching because of NATs and firewalls. The data center nodes could also act as relay nodes for these kind of connections. This would benefit the system in a way that information could be shared across different private networks without the need of waiting for such information to be propagated to data centers and then finally to the other edge locations.

Besides, it should be allowed more than one edge node from a private network to be connected to a node in the level above. This would provide better fault-tolerance, although it would require a mechanism to reduce redundancy in messages exchanged between networks and prevent cycles.

In the future, we also aim at implementing a mechanism for recovering messages lost through edge connections, due to failures in those connections. This mechanism could be similar to the one we implemented for recovering missing messages on Thicket upon tree repair, although with some adaptations.

Due to limited budget we were not able to evaluate the effect of open connections in a full-mesh membership when comparing to ours, which relies on partial views. In the future, we would also like to study such effect with a greater number of nodes, to be able to understand the negative impact of a number of TCP connections in the order of n.

Bibliography

[1] M. K. Aguilera, W. Chen, M. Kawazoe, A. Wei, and S. Toueg. *Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication.* 1997.

[2] J. Blustein and A. El-maazawi. *Bloom filters. a tutorial, analysis, and survey.* Tech. rep.

[3] *Broadcast tree for data dissemination.* `https://issues.apache.org/jira/browse/CASSANDRA-12347`. Accessed: 2018-01-23.

[4] *Cassandra - Gossip 2.0.* `https://issues.apache.org/jira/browse/CASSANDRA-12345`. Accessed: 2018-01-23.

[5] M. Castro, M. Costa, and A. Rowstron. *Peer-to-peer overlays: structured, unstructured, or both?* Tech. rep. 2004. URL: `https://www.microsoft.com/en-us/research/publication/peer-to-peer-overlays-structured-unstructured-or-both/`.

[6] T. D. Chandra and S. Toueg. "Unreliable Failure Detectors for Asynchronous Systems (Preliminary Version)." In: *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing.* PODC '91. Montreal, Quebec, Canada: ACM, 1991, pp. 325–340. ISBN: 0-89791-439-2. DOI: `10.1145/112600.112627`. URL: `http://doi.acm.org/10.1145/112600.112627`.

[7] W. Chen, S. Toueg, and M. K. Aguilera. "On the quality of service of failure detectors." In: *IEEE Transactions on Computers* 51.1 (2002), pp. 13–32. ISSN: 0018-9340. DOI: `10.1109/12.980014`.

[8] *Consul.* `https://consul.io/`. Accessed: 2018-07-09.

[9] A. Dadgar, J. Phillips, and J. Currey. "Lifeguard : SWIM-ing with Situational Awareness." In: *CoRR* abs/1707.00788 (2017). arXiv: `1707.00788`. URL: `http://arxiv.org/abs/1707.00788`.

[10] A. Das, I. Gupta, and A. Motivala. "SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol." In: *DSN.* 2002.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-value Store." In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 205–220. ISSN: 0163-5980. DOI: 10.1145/1323293.1294281. URL: http://doi.acm.org/10.1145/1323293.1294281.

[12] J. Dowling and A. H. Payberah. "Shuffling with a Croupier: Nat-Aware Peer-Sampling." In: *2012 IEEE 32nd International Conference on Distributed Computing Systems*. 2012, pp. 102–111. DOI: 10.1109/ICDCS.2012.19.

[13] *Dynamic snitching in Cassandra: past, present, and future*. https://www.datastax.com/dev/blog/dynamic-snitching-in-cassandra-past-present-and-future. Accessed: 2018-01-18.

[14] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. "Lightweight Probabilistic Broadcast." In: *ACM Trans. Comput. Syst.* 21.4 (Nov. 2003), pp. 341–374. ISSN: 0734-2071. DOI: 10.1145/945506.945507. URL: http://doi.acm.org/10.1145/945506.945507.

[15] P. Felber, X. Defago, R. Guerraoui, and P. Oser. "Failure detectors as first class objects." In: *Proceedings of the International Symposium on Distributed Objects and Applications*. 1999, pp. 132–141. DOI: 10.1109/DOA.1999.794001.

[16] M. Ferreira, J. Leitao, and L. Rodrigues. "Thicket: A Protocol for Building and Maintaining Multiple Trees in a P2P Overlay." In: *2010 29th IEEE Symposium on Reliable Distributed Systems*. 2010, pp. 293–302. DOI: 10.1109/SRDS.2010.19.

[17] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié. "SCAMP: Peer-to-Peer Lightweight Membership Service for Large-Scale Group Communication." In: *Proceedings of the Third International COST264 Workshop on Networked Group Communication*. NGC '01. London, UK, UK: Springer-Verlag, 2001, pp. 44–55. ISBN: 3-540-42824-0. URL: http://dl.acm.org/citation.cfm?id=648089.747488.

[18] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. "Scalable, Distributed Data Structures for Internet Service Construction." In: *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*. OSDI'00. San Diego, California: USENIX Association, 2000. URL: http://dl.acm.org/citation.cfm?id=1251229.1251251.

[19] I. Gupta, T. D. Chandra, and G. S. Goldszmidt. "On Scalable and Efficient Distributed Failure Detectors." In: *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*. PODC '01. Newport, Rhode Island, USA: ACM, 2001, pp. 170–179. ISBN: 1-58113-383-9. DOI: 10.1145/383962.384010. URL: http://doi.acm.org/10.1145/383962.384010.

[20] *HashiCorp*. https://www.hashicorp.com/. Accessed: 2018-07-09.

[21] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. "The /spl phi/ accrual failure detector." In: *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.* 2004, pp. 66–78. DOI: 10.1109/RELDIS.2004.1353004.

[22] *Introduce a Peer Sampling Service for partial cluster views.* https://issues.apache.org/jira/browse/CASSANDRA-12346. Accessed: 2018-01-23.

[23] V. Jacobson. "Congestion Avoidance and Control." In: *SIGCOMM Comput. Commun. Rev.* 18.4 (Aug. 1988), pp. 314–329. ISSN: 0146-4833. DOI: 10.1145/52325.52356. URL: http://doi.acm.org/10.1145/52325.52356.

[24] M. Jelasity, A. Montresor, and O. Babaoglu. "T-Man: Gossip-based Fast Overlay Topology Construction." In: *Comput. Netw.* 53.13 (Aug. 2009), pp. 2321–2339. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2009.03.013. URL: http://dx.doi.org/10.1016/j.comnet.2009.03.013.

[25] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. "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 Theory of Computing.* STOC '97. El Paso, Texas, USA: ACM, 1997, pp. 654–663. ISBN: 0-89791-888-6. DOI: 10.1145/258533.258660. URL: http://doi.acm.org/10.1145/258533.258660.

[26] A. Kermarrec, A. Pace, V. Quema, and V. Schiavoni. "NAT-resilient Gossip Peer Sampling." In: *2009 29th IEEE International Conference on Distributed Computing Systems.* 2009, pp. 360–367. DOI: 10.1109/ICDCS.2009.44.

[27] A.-M. Kermarrec and M. van Steen. "Gossiping in Distributed Systems." In: *SIGOPS Oper. Syst. Rev.* 41.5 (Oct. 2007), pp. 2–7. ISSN: 0163-5980. DOI: 10.1145/1317379.1317381. URL: http://doi.acm.org/10.1145/1317379.1317381.

[28] A. Lakshman and P. Malik. "Cassandra: A Decentralized Structured Storage System." In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922. URL: http://doi.acm.org/10.1145/1773912.1773922.

[29] J. a. Leitão, R. van Renesse, and L. Rodrigues. "Balancing Gossip Exchanges in Networks with Firewalls." In: *Proceedings of the 9th International Conference on Peer-to-peer Systems.* IPTPS'10. San Jose, CA: USENIX Association, 2010, pp. 7–7. URL: http://dl.acm.org/citation.cfm?id=1863145.1863152.

[30] J. Leitao. "Gossip-based broadcast protocols." Master's thesis. Universidade de Lisboa, 2007.

[31] J. Leitao, J. Pereira, and L. Rodrigues. "Epidemic Broadcast Trees." In: *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems.* SRDS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 301–310. ISBN: 0-7695-2995-X. URL: http://dl.acm.org/citation.cfm?id=1308172.1308243.

[32] J. Leitao, J. Pereira, and L. Rodrigues. "HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast." In: *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* DSN '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 419–429. ISBN: 0-7695-2855-4. DOI: 10.1109/DSN.2007.56. URL: http://dx.doi.org/10.1109/DSN.2007.56.

[33] J. Leitão, N. Carvalho, J. O. Pereira, R. Oliveira, and L. Rodrigues. "On Adding Structure to Unstructured Overlay Networks." In: *Handbook of Peer-to-Peer Networking.* Ed. by X. Shen, H. Yu, J. Buford, and M. Akon. Springer, 2010. URL: http://www.springer.com/engineering/signals/book/978-0-387-09750-3?detailsPage=toc.

[34] J. Leitão, J. P. Marques, J. Pereira, and L. Rodrigues. "X-BOT: A Protocol for Resilient Optimization of Unstructured Overlay Networks." In: *IEEE Transactions on Parallel and Distributed Systems* 23.11 (2012), pp. 2175–2188. ISSN: 1045-9219. DOI: 10.1109/TPDS.2012.29.

[35] C. Meiklejohn and H. Miller. "Partisan: Enabling Cloud-Scale Erlang Applications." In: (Feb. 2017).

[36] M. Phan. "A literature review of failure detection Within the context of solving the problem of distributed consensus." In: 2015.

[37] Y. Qiao and F. E. Bustamante. "Structured and Unstructured Overlays Under the Microscope: A Measurement-based View of Two P2P Systems That People Use." In: *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference.* ATEC '06. Boston, MA: USENIX Association, 2006, pp. 31–31. URL: http://dl.acm.org/citation.cfm?id=1267359.1267390.

[38] R. van Renesse, D. Dumitriu, V. Gough, and C. Thomas. "Efficient Reconciliation and Flow Control for Anti-entropy Protocols." In: *Proceedings of the 2Nd Workshop on Large-Scale Distributed Systems and Middleware.* LADIS '08. Yorktown Heights, New York, USA: ACM, 2008, 6:1–6:7. ISBN: 978-1-60558-296-2. DOI: 10.1145/1529974.1529983. URL: http://doi.acm.org/10.1145/1529974.1529983.

[39] B. Satzger, A. Pietzowski, W. Trumler, and T. Ungerer. "A New Adaptive Accrual Failure Detector for Dependable Distributed Systems." In: *Proceedings of the 2007 ACM Symposium on Applied Computing.* SAC '07. Seoul, Korea: ACM, 2007, pp. 551–555. ISBN: 1-59593-480-4. DOI: 10.1145/1244002.1244129. URL: http://doi.acm.org/10.1145/1244002.1244129.

[40] *Serf - Decentralized Cluster Membership, Failure Detection and Orchestration.* https://consul.io/. Accessed: 2018-07-09.

[41] A. Stavrou, D. Rubenstein, and S. Sahu. “A lightweight, robust P2P system to handle flash crowds.” In: *IEEE Journal on Selected Areas in Communications* 22.1 (2004), pp. 6–17. ISSN: 0733-8716. DOI: 10.1109/JSAC.2003.818778.

[42] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos. “Challenges and Opportunities in Edge Computing.” In: *2016 IEEE International Conference on Smart Cloud (SmartCloud).* 2016, pp. 20–26. DOI: 10.1109/SmartCloud.2016.18.

[43] S. Voulgaris, D. Gavidia Simonetti, and M. van Steen. “CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays.” In: *Journal of Network and Systems Management* 13.2 (2005). steen2005.03, pp. 197–217. ISSN: 1064-7570. DOI: 10.1007/s10922-005-4441-x.