Francisco Martins Magalhães

Computer Science and Engineering

Membership Service for Large-Scale Edge Systems

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

Master of Science in
Computer Science and Informatics Engineering

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

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2018

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, are now done in 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 distributed systems. 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 thesis, we aim at designing 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. For that reason, we will study existing membership protocols to be used as baselines. Some issues, such as dealing with NATs and firewalls, come along with the introduction of the new Edge Computing paradigm. Our membership service should be able to deal with those issues in a efficient way.

Additionally, we will leverage such membership service to support the operation of Cassandra, as a distributed storage system. Later, we will study how to leverage the membership service to move some of the aspects of the storage system to the edge of the network.

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 afim 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 data 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.

Afim de poder continuar a construir e gerir estes serviços, é necessário um serviço de *membership*, um componente fundamental nos sistemas distribuídos. Um serviço de *membership* 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, visamos desenhar um serviço de *membership* novo, escalável e eficiente, capaz de suportar aplicações de grande escala com componentes na berma da rede, mais perto dos utilizadores destes serviços. Por isso, será necessário estudar protocolos de *membership* já existentes para serem usados como base. Com a introdução do novo paradigma da Computação na Berma, surgem problemas como lidar com NATs e *membership*. O nosso serviço de membership deverá ser capaz de lidar eficientemente com essas dificuldades.

Para além disso, esse serviço será utilizado primeiramente para suportar a execução do Cassandra, como sistema de armazenamento distribuído. Mais tarde, estudaremos a possibilidade de utilizar esse serviço para transportar alguns dos aspetos desse sistema para 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 increases. 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 data retrieval. 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 made enormous changes in the way we live and work and in particular how we build and maintain large-scale systems. Many large-scale systems where deployed in recent years so as 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, such as 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 clocks).

However, the Cloud cannot scale indefinitely. In spite of the large amount of advantages brought by the 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 increase of latency in 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 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 is membership service, that tracks the components that are currently active and available to the system. Many membership services were designed in the previous years as to support large-scale systems [9, 19]. 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 on multiple data centers and in edge locations.

1.2 Motivation

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

Current membership services solutions in distributed systems are designed for applications that rely on Cloud-based infrastructure. Even though they have proven to be scalable and failure resilient, membership services need to be changed in order to support the Edge Computing paradigm. For instance, they have to deal with the fact that many nodes are behind NATs and Firewalls, which doesn't happen inside a data center. For this reason, membership services have now to consider the existence of those in order not to create an overhead on nodes that are behind them in the network.

Large-scale distributed storage systems, such as Cassandra, are a key support in many large-scale systems nowadays. In Cassandra's membership service, every node knows each other node in the system, which forces each node to store a lot of information about the membership. Many approaches for membership services were deployed in large-scale distributed systems in a way that nodes need to know only a fraction of the active nodes. However, those approaches don't 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. Extending such solutions to deal with large numbers of components scattered in edge locations is an open challenge.

1.3 Contributions

In this work, we plan to create or modify a membership service in order to make it capable of supporting distributed systems that run in devices located in the edge of the system.

Besides that, we plan to develop or improve an already existent membership gossip-based broadcast protocol so it can deal with the notion of different data centers for the same service. This solution should be able to ease the communication between different data centers so the dissemination of information is faster. A fraction of nodes of different data centers should know each other and therefore be included in each other's 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.

Additionally, we also aim to leverage such service to modify the membership service of the Cassandra distributed storage system, as to showcase the applicability of our approach.

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. Furthermore, it describes Cassandra's architecture in the aspects related with the work. It also discusses some solutions to improve the scalability of the distributed membership services.

Chapter 3 presents the work plan for this thesis, which includes a brief description of our solution and some observation on how we are going to evaluate our solution. Furthermore, it also presents a schedule for the main task to be conducted in the context of the planned work.

Chapter 2

Related Work

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

We start this chapter by discussing briefly 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 the work to be conducted in this thesis.

This is followed by a discussion on related work on peer-to-peer systems, which provides foundations for building decentralized membership services.

2.1 Edge-Computing

This "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 have new 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 system, 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 doesn't work anymore. Because of this evolution, it is necessary to have data centers closer to the devices. Thus, edge computing is used to perform data processing at the edge of the network, improving and overcoming some pitfalls 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 trough which network traffic is directed. The work presented in [27] 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.

2.2 Membership

A key aspect in managing and building such large-scale services, that can cope with this concept of edge computing is the membership service. Sometimes it's 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. 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 [4] 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 trace 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.

2.2.1 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 DHTs, 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 windows, 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 don't 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.

2.3 Gossip-based protocols

This section addresses gossip-based broadcast protocols and why 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 in 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 fault tolerance.

The work present in [14] 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 [7] combined it's 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 communication 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. 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 received 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 adaptative 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.

2.3.1 Strategies

The work presented in [17] 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 next 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, altough 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.

2.4 Overlay Network

We call *overlay network* to a logical network that operated on top of another (tipically physical) network. This network consists in the nodes and their 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 [3, 24], mainly when it comes to large-scale network overlays.

2.4.1 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 large 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. A DHT is a class of decentralized distributed system. Similarly to a hash table, it provides a lookup service and a dictionary-like interface. However, the nodes are spread across the network. They are being used nowadays in large-scale services such as peer-to-peer (P2P) file sharing, and content distribution systems.

2.4.1.1 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 [10].

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.

2.4.1.2 Chord

Chord is a structured overlay (DHT-based) algorithm based on consistent hashing [13]. 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. 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).

[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.

2.4.1.3 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 as 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.

2.4.2 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 covers 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.

2.4.2.1 SCAMP

Scamp [9] 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) nodes, 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 propagates 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 doesn't 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 hasn't received a heartbeat message in a long time. In this case, it tries to resubscribe. For this end, it sends a new subscription message trough an arbitrary node in its *PartialView*, as if it was trying to join the overlay for the first time.

2.4.2.2 Cyclon

Cyclon [28] 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 [26]. For this reason, it offers better fault tolerance than Scamp. It maintains fixed sized partial views in each node, where the sized 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 becoming 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.

2.4.2.3 HyParView

HyParView [19] 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 trough 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 know another node that is already in the system (and is part of the overlay), as in Scamp and Cyclon. In a node active view, there are the nodes to which he has a communication link (its neighbors). 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.

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 of the reactive strategy. The detection of failures of nodes in a node's active view is also quite fast, because all members are testes at each gossip step.

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

2.4.3 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 don't allow to exploit properties of the physical network, like node proximity. On the other hand, structured

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

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 [20], 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.

2.4.3.1 T-MAN

T-MAN [12] 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.

2.4.3.2 X-BOT

The X-BOT [21] 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.

2.4.3.3 PlumTree

In order to mitigate the redundancy problem on gossip-based broadcast schemes using the eager push strategy, PlumTree [18] 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.

In a nutshell, the protocol combines basic flooding with a prune mechanism. 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, which becomes much more expensive.

2.4.4 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 [19, 21]. 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 a 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 trough 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 don't 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 deliver 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.

2.5 Cassandra

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 [15] 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.

2.5.1 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.

2.5.2 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 [13], creating a ring. 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. This means that each node will be assigned as many keys as those that fall between its own and its predecessor's identifiers. For this reason, when a node leaves or joins the rings, only its direct neighbors will be affected.

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 its possible. In the same way, it avoids routing requests to poorly performing but alive nodes. This monitoring strategy is called dynamic snitch [6].

2.5.3 Membership

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

In order to provide a membership service which satisfies the needs, Cassandra a gossip protocol based on Scuttlebutt, a anti-entropy gossip protocol [25] is employed. In a 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 perfect dissemination.

Scuttlebut is a 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.

There is also an underlying failure detection mechanism, 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, according to a technique called hinted handoff.

2.6 NAT-aware Membership

In today's Internet, a big amount of nodes are behind NATs (Network Address Translation gateways) and firewalls. In order to scale Cassandra's membership to 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. Since nodes cannot establish direct connections to these private nodes, they become under-represented in partial views. We can either assume that our membership will be developped to run only in data centers - and we don't have to worry with this issue - or also in the edge. In this section we will present some approaches to deal with NATs and firewalls in the network.

2.6.1 Relaying

Usual P2P protocols that handle NATs circumvent them by the use of relaying and hole-punching techniques. This allows to route packets to nodes behind NATs. These techniques were used to solve the issue of balancing gossip in networks with NATs by sending a message via a relay node, which forwards it to the private node, preventing private nodes to receive and process an unbalanced number of gossip messages.

The work in [16] 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 has routers for nodes in confinement domains in the network, forwarding the information, instead of executing the usual gossip broadcast operation.

Each node keeps track of a value which it 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.

2.6.2 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 [5] describes the first NAT-aware peer sampling service which doesn't rely on those mechanisms but rather on a new one. It requires the use of two bounded in size partial views: 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.

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. Futhermore, its procedures can be integrated in already existing peer sampling services.

2.7 Existing solutions

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.

2.7.1 Partial Views

In Cassandra's actual 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 to have 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 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 opened connections between peers in the system, which would create a big overhead.

The issue described in [2] identifies the need of changes in the dissemination aspects of the current Cassandra's gossip subsystem. In order to do so, the author enumerates three main requirements: a peer sampling service which is based on partial views [11], a broadcast tree protocol based on the peer sampling service [1] an anti-entropy component for dissemination (to which there is not an implemented solution yet).

For the first requirement, the approach presented in [11] 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 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 [8], 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.

Some membership solutions were introduced to scale large systems. Partisan [23] is a distribution layer for Erlang. It proves to be better than Distributed Erlang (which relies in an incomplete click between all nodes). when it comes to latency. Partisan allows developers to specify the network topology at runtime. For the peer-to-peer topology, it uses a membership based on HyParView and PlumTree. The use of HyParView provides global system connectivity in a scalable way. Furthermore, PlumTree provides reliable broadcast, combining a deterministic tree-based broadcast protocol with a gossip protocol. In order to communicate between non-directly connect nodes, Partisan's backend

computed a spanning tree routed at each node using an instance of PlumTree.

2.7.2 Discussion

Two approaches were introduced as possibilities to scale Cassandra's membership. Both approaches use HyParView, which removes the need of having direct connections between every pair of nodes in Cassandra's system, by the use of partial views. However, HyParView assumes that every node is in the public network. When it has to deal with NATs and Firewalls, it is not possible to ensure that the properties keep being valid, because an assumption is not satisfied. Moreover, these approach does not deal with the notion of different data centers (it assumes a flat uniform node distribution), which is one of the goals of this thesis.

Either with [11] or [23], both of which use HyParView, nothing guarantees that nodes which replicate the same data blocks will be in each other's partial views. Cassandra benefits from its full mesh because it can fastly send messages between the nodes which replicate the same data. Neither of the solutions can guarantee that nodes which share data have a direct connection at the membership/overlay level.

Chapter 3

Work Plan

In this chapter, we establish a summary of our solution and work plan for this thesis. This work is expected to last for around seven months and consists in a set of tasks which duration can vary according to many factors. However, we present an estimated duration for each of the tasks.

Main tasks include solution design, implementation, evaluation and writing.

Table 3.1: Task schedule.

Task	Start Date	End Date	Duration (weeks)
Solution Design	19 February	26 March	5
Implementation			
Multiple Data Centers	26 March	30 April	5
NAT and Firewall-Aware	30 April	28 May	4
Cassandra's integration	28 May	9 July	6
Evaluation	9 July	13 August	5
Writing	13 August	24 September	7

In the **solution design phase**, we aim at designing a solution which can satisfy the requirements we established previously in this document. We aim at designing a membership component which can leverage the notion of data centers in a membership service, support applications running according to the Edge Computing paradigm, and we also aim at designing this solution in a way it is compatible with Cassandra's membership and architecture.

As we mentioned, there is the need that peers from different data centers can be included in each other's partial views so we can disseminate information quickly between data centers. There are two main aspects we need to consider. Firstly, we probably want to maintain less information about remote neighbors than about neighbors in the same local

network. However, we will want to gossip to remote neighbors with higher probability, so we can accelarate the dissemination. If too many messages are sent to distant neighbors, a overhead is potentially created in the routers in the path. However, if a small number of messages are disseminated to distant neighbors, some parts of the network may never get the information. The approach presented in [22] aims to minimize the overhead in the routers between distant nodes, while garanteeing that the message is disseminated in a fast way. Some membership protocols were studied in **Chapter** 2. Based on the analysis of those protocols and also on previous knowledge about HyParView and X-BOT, those protocols will be used as baselines for the membership service. X-BOT [21] may also be used with a oracle related to nodes' location in the network, as to deal with the problem of distributing nodes identifiers in partial views according to their location in the network (e.g. in which data center they are). Furthermore, an approach for firewall and NAT-aware membership services (as the ones we presented) should be integrated in the membership protocol used as baseline, so as to support applications with components running in the edge, and not only inside the public network.

Later, we will try to incorporate the membership service with Cassandra in a way that it doesn't compromise Cassandra's requirements, such as fast replication. In order to leverage such service to support the operation of Cassandra, we propose a solution based on two layers. The bottom layer would follow the characteristic of an unstructured overlay, maintained with the use of partial views. The bottom layer would be used to exchange membership related information, while the top layer would be used for Cassandra's replication and partitioning. The two layers would not be independent. Instead, the top layer would be able to make requests to the bottom layer when it comes to creating connections to nodes which replicate the same objects. In this way, the membership service would benefit from the scalability of the partial view and the nodes would still be able to replicate information quickly.

In the **implementation phase**, we aim at implementing the mechanisms mentioned above with the use of existing protocols as baseline.

In the **evaluation phase**, we aim at comparing our membership service with existing ones and study the results according to the metrics defined in previous chapter. Furthermore, we will evaluate the new membership service integrated in Cassandra and compare it to the current release.

Finally, the document will keep being adjusted slowly according to the progress. However, the effective period for **writing** will start after the evaluation phase.

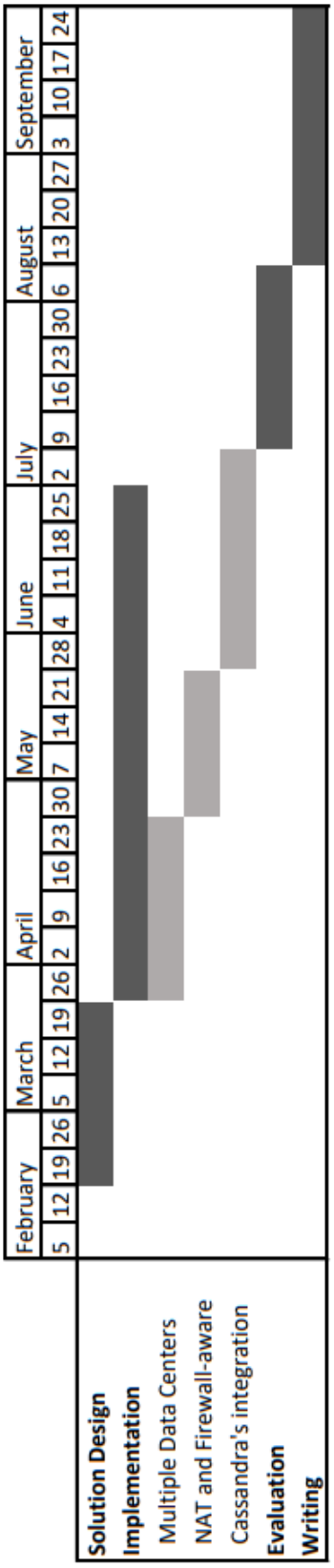

Figure 3.1: Gantt chart of the proposed work plan

Bibliography

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

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

[3] 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/.

[4] 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.

[5] 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.

[6] *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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] M. Lin and K. Marzullo. *Directional Gossip: Gossip in a Wide Area Network*. Tech. rep. La Jolla, CA, USA, 1999.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.