João Miguel Pomar Monteiro

Degree in Computer Science and Engineering

Scaling DHTs towards millions

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

Master of Science in
Computer Science and Engineering

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

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Februrary, 2021

The work reported in this document was partially supported by
Fundação para a Ciência e Tecnologia through the Project NG-STORAGE (PTDC/CCI-INF/32038/2017)

Abstract

The P2P paradigm allows the creation of decentralized systems that overcome several shortcomings of centralized approaches (e.g., scalability, fault tolerance, operational costs, etc.). While these systems may have peaked in popularity in the previous decade with applications such as BitTorrent, they are starting to be sought after again due to their added level of security and integrity they offer to blockchain systems or the censorship resistant properties in distributed file systems. To provide efficiency in resource lookups, P2P applications usually rely on structured overlay networks that act as distributed hash tables (DHTs), thus organizing nodes in a predictable fashion which allows for participants and resources to be found usually in a logarithmic number of steps in relation to the network's size. While the number of steps needed is usually optimal, the links formed between participants can have undesirable properties when considering network latency and geographical location, which may have a significant negative impact in resource retrieval time.

While overlays that address these issues exist, there is not enough evidence that the networks they create can tolerate millions of nodes participating concurrently. Additionally, structured overlays usually suffer from performance degradation in highly dynamic settings where the system's membership changes frequently. This last problem can be addressed by using gossip protocols to reduce the dependency of the DHT topology from the distributed protocol that manages the topology itself.

In this work, we propose to create a novel multi-level DHT that organizes nodes hierarchically, creating clusters of nodes that are topologically closer to each other. By doing this, participants can efficiently find nearby resources with low latency, while retaining the desired logarithmic steps needed to find content located further way. The overlay will focus on providing unprecedented scalability and fault tolerance in dynamic networks, which will be evaluated in a large scale cluster. This work is being conducted in collaboration with Protocol Labs that operate the Interplanetary File System (IPFS).

Keywords: distributed hash tables, resource location, peer-to-peer systems

Resumo

O paradigma entre-pares (P2P, do Inglês *peer-to-peer*) permite a criação de sistemas descentralizados que superam limitações de abordagens centralizadas (por exemplo, escalabilidade, tolerância a falhas, custos operacionais, etc.). Embora estes sistemas tenham atingido um pico de popularidade na década anterior, estão a começar a ser procurados novamente devido ao seu nível de segurança e integridade acrescentado que oferecem a sistemas de *Blockchain* ou às propriedades resistentes à censura em sistemas de ficheiros distribuídos. Para proporcionar eficiência na procura de recursos, as aplicações que recorrem a sistemas P2P dependem geralmente de redes estruturadas sobrepostas que atuam como tabelas de dispersão distribuídas, organizando os nós de uma forma previsível permitindo que os participantes e recursos sejam encontrados num número de passos logarítmico em relação ao tamanho da rede. Embora o número de passos necessário seja geralmente ótimo, as ligações formadas entre os participantes podem ter propriedades indesejáveis quando se considera a latência da rede e a localização geográfica, o que pode ter um impacto negativo no tempo de obtenção dos recursos.

Embora existam redes estruturadas que abordam estes desafios, não há provas suficientes de que estas tolerem a participação simultânea de milhões de nós. Além disso, as redes estruturadas sofrem normalmente de degradação de desempenho em ambientes dinâmicos, onde a adesão ao sistema muda frequentemente. Este último desafio pode ser resolvido utilizando protocolos de rumores para reduzir a dependência da topologia do protocolo distribuído que gere a própria topologia.

Neste trabalho, propomos a criação de uma nova rede estruturada que organize os nós hierarquicamente, criando aglomerados de nós que estão topologicamente mais próximos uns dos outros. Ao fazer isto, os participantes podem encontrar eficientemente recursos próximos com baixa latência, mantendo ao mesmo tempo o números de passos logarítmicos necessários para encontrar conteúdos longínquos. A rede estruturada proporcionará uma escalabilidade sem precedentes e tolerância a falhas em redes dinâmicas, que será avaliada num grupo de computadores em grande escala. Este trabalho está a ser realizado em colaboração com a Protocol Labs que opera o *Interplanetary File System* (IPFS).

Palavras-chave: tabelas de dispersão distribuídas, localização de recursos, sistemas entre-pares ...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Peer-to-peer (P2P) systems are based on a decentralized distributed model that can surpass the shortcomings of more centralized approaches, for instance, scalability, fault tolerance, and high infrastructural costs. In this model, each participant in the network (referred to as peer) acts as both a client and server, sharing its own resources, e.g., computational power, storage space, and bandwidth, thus improving system scalability. As each peer acts as a server, it removes the problem of having a single point of failure. Infrastructural costs are lowered by removing the need for adding, upgrading, and maintaining powerful machines as the system needs to scale.

For a system to scale properly, while not imposing high overhead costs to each participant due to system affiliation dynamics, (i.e., nodes leaving and joining concurrently), only maintains a partial view of all participants in the system which is managed by a distributed membership protocol. The logical network formed by all these peer connections that are built on top of the physical network topology is called an overlay. Depending on the way the membership protocol handles connections and dynamics, overlay networks can be classified in two types: unstructured and structured overlays.

In unstructured overlays [12, 14, 35], the membership protocol is flexible in creating links between participants as they are mostly random in their nature. Due to this, they typically incur in less management overhead, and therefore are typically more resistant to failures in highly dynamic systems. Unstructured overlay networks can be used, for instance, to implement broadcast primitives and resource location services to discover and retrieve resources that are kept by one or more participants. Peers communicate with each other through the use of gossip protocols [2, 11], where they exchange messages randomly between themselves until they reach the message target.

In contrast, structured overlays create a rigid topology known *a priori*, as the membership protocol enforces constraints between the links that are formed in the network, usually using random unique identifiers from an identifier space, attributed to nodes when they join the system. Structured overlays can serve as distributed hash tables (DHT) [8, 22, 32, 33, 36] where, each node is responsible for maintaining a set of resource keys close to its identifier. By using key-based routing algorithms, any participant can efficiently, usually in a logarithmic number of hops, find resources and other participants in the network. Due to their less flexible structure, structured overlays incur on higher overheads to create and maintain overlay links and as so, performance degrades when faced with highly dynamic networks. Structured overlays can be leveraged, for instance, to create distributed file sharing applications where each node is responsible for a fraction of the files stored. Examples of such applications are BitTorrent [3], and more recently, the InterPlanetary File System [1]. The latter will serve as case study to the work conducted on this thesis

1.2 Motivation

While many systems have integrated P2P designs into their architecture, in large scale scenarios, these may face scalability challenges. While overlays offer an easier management of the overall system, links established among participants in the overlay may not reflect the underlying network. In large scale systems, this can reflect into poorly optimized networks and high latency costs. Solutions to this challenge have some tradeoffs. By enforcing distance metrics into the topology constrains, while a portion of participants might benefit from the increase in performance, other peers that do not meet connection requirements to most other participants might be left with an insufficient amount of connections which can lead some peer to become disconnected and unable to participate in the system.

One question that has not received much attention to date, is if P2P systems could leverage the coexistence of multiple overlays to enhance the system's performance, while not incurring in excessive amounts of maintenance overhead. For instance, if P2P could benefit from the use of both structured and unstructured overlays to provide more stability in dynamic scenarios. While there is some literature [21, 27] exploring the use of these overlays, it remains an open question on how to better take advantage of such an approach. Multi-Level DHTs [6] are another case of combining multiple overlays where each of the levels imposes different constraints (i.e., each level is a separate DHT that corresponds to some latency requirements).

Lastly, there is another more practical issue regarding the experimental evaluations on large scale networks. While there is a considerable amount of literature on the study of both structured and unstructured overlays, the experimentation is often imperfect as the evaluation is conducted only though the use of simulations that can only scale to a certain amount of participants and do not capture all aspects of a real environment. Therefore,

there is a lack of research on P2P overlays where the experiments take into consideration large scale settings (i.e., millions of nodes) in realistic settings.

1.3 Expected contributions

The main contributions we expect to achieve with this work are the following:

- The implementation and evaluation of a novel hierarchical DHT design and management protocol that takes into account network locality and allows for unprecedented scaling.
- A new DHT routing scheme that takes advantage of the aforementioned design.
- An experimental comparison between our proposal and the state of the art DHT designs under distinct conditions to be conducted by simulation.

1.3.1 Research Context

The work to be conducted in the context of this thesis is pursued in collaboration with Protocol Labs in the context of their IPFS ecosystem. Protocol Labs specializes in the development of P2P systems and frameworks. Examples of their work include: libp2p, a framework for developing P2P applications; Filecoin, a cryptocoin based on distrusted storage; IPFS, a P2P file system averaging tens of thousands of participants daily.

In particular, this work, is tightly integrated in a Protocol Labs request [26] targeted at the research of novel Multi-Level DHT design and evaluation.

1.4 Document organization

The document is organized as follows:

- In Chapter 2 we discuss related work. We explain in more detail the architectural stack of P2P, different challenges that may arise, and study closer the differences between the structured and unstructured overlays. We provide an overview on different implementations of P2P networks. Finally, we study in more detail a distributed file system architecture and protocols used.
- In Chapter 3 we describe the proposed work, evaluation plan and the scheduling for future work.

Chapter 2

Related Work

In this chapter, we will start by detailing the fundamentals of Peer-to-Peer concepts, from their architecture, to the applications that leverage them and ending on some of the challenges that may arise when designing them (Section 2.1).

We will detail the different types of overlay networks that can be used when creating a P2P system, their use cases and discuss examples from the literature. First, we study unstructured overlay networks (Section 2.2), followed by structured overlays (Section 2.3) and finally, we detail possible ways of combining these overlays into a hybrid type of overlay (Section 2.4).

Lastly, we present a specific case study of a distributed system that uses P2P networks, the InterPlanetary File System(IPFS) that will be the main focus of this thesis (Section2.5).

2.1 Peer-to-Peer Systems' Concepts

In a centralized architecture, a single computer or server, holds all of the system's resources and performs all the necessary computing. These systems, while easier to maintain and implement, have an extensive list of drawbacks: the server becomes a single point of failure as all the interaction with the system depends on the availability of a single machine; the interaction with the system can heavily depend on the proximity with the server, as clients nearby can expect better services than those further away; the machine can become a bottleneck when the service is faced with a higher demand, etc. [17].

To overcome these problems, the service can be distributed through a network of machines where each one is responsible for a share of the resources. A distributed system can be of two types: centralized and decentralized.

Centralized distributed systems usually depend on a coordinator (or central server) to achieve its objectives and to distribute tasks among participants. Although this type

of architecture provides a better scalability compared to a centralized alternative, the coordinator can still become a major point of failure and a bottleneck as all operation depend on its availability. Also, as the system increases in dimension, the central server needs to keep track of all members, which can become unfeasible in large scale systems.

In decentralized distributed systems, participants rely on a distributed coordination mechanism, where each participant independently manages its tasks and connections to peers. These systems offer the most availability and scalability at the cost of a higher maintenance overhead at each machine. The Peer-to-Peer (P2P) paradigm allows the creation of decentralized distributed systems.

In the P2P paradigm, participants form a graph where each node is a participant, and the edges are formed by the network connections. Participants, which act as server and clients, dynamically divide the workload among themselves keeping the system available even in the cases where nodes fail or leave the network.

Another advantage, due to their organic growth and no need for dedicated infrastructure, is that the amount of resources increases every time a node joins the network without the need for an infrastructure upgrade [30].

2.1.1 Architecture

The architecture of a Peer-to-Peer application can be decomposed in multiple layers as shown in Figure 2.1. Each layer provides an abstraction to the one above.

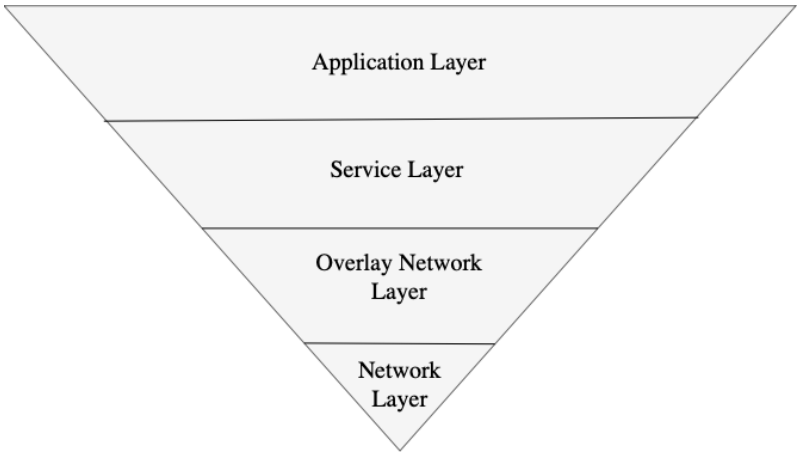


Figure 2.1: Peer-to-Peer generic architecture (adapted from [17])

Network Layer The bottom, and first layer is the Network Layer which has the responsibility to interface with a transport layer using protocols such as TCP or UDP. This layer receives and delivers messages sent by other participants, while also queuing messages to be sent using the transport layer.

Overlay Network In order to maintain information about participants in the network, P2P systems leverage the use of overlay networks [12, 22, 33]. Overlay networks are

built on top of another network and are composed of virtual links independent of the underlying network topology. As so, neighboring nodes in this abstraction, can be separated by several hops in the underlying network topology. Nodes can enter and leave the system, but the overlay has to ensure that the network always stays connected.

Each member, using a membership protocol, has to manage a set that contains node identifiers and their contact information (IP address and port) in order to be able to communicate with them. A distributed membership protocol should allow for a node to fill its neighbor set when it first enters the system, and its identifier should be added to some of the neighbor sets maintained by other nodes. When a node leaves the system, its identifier should be removed eventually from all neighbor sets. Participants join the network by contacting an already present member, usually called the *contact node*.

A simple implementation of a membership management protocol is for every node to keep a list of all network members. As the system grows, this strategy becomes unfeasible due the extensive central information that needs to be stored and managed by each node and the cost to keep that information up to date. To overcome this problem in large scale systems, each node's neighbor set contains only a subset of the network called a *partial view*.

Service Layer The Service layer [4, 11], leverages the built overlays to execute services such as lookup protocols or messages broadcasting, providing a higher level abstraction for the layers above .

Application Layer The application layer [1, 3] exposes the end interface to the user and implements the application logic using the service layers bellow.

2.1.2 Applications

Among the most popular uses of P2P is file sharing. Famously, Napster was one of the first applications for file sharing using P2P systems. Released in 1999, Napster allowed a user to search for a file that other participant had shared in the network, and directly download it from one of more peers. Rapidly, many other applications appeared that offered similar but improved services like Gnutella and Kazza [30]. Currently, one of the most famous P2P file sharing application is BitTorrent [3]. Users, using *trackers*, can find other participants sharing a specific file. Files in this protocol are divided into blocks and a user can request concurrently different blocks from multiple users. The moment a user starts downloading, it becomes part of the network, also contributing to other participants downloading the same file.

Later in this chapter we will introduce this dissertation case study, IPFS [1] that uses P2P networks to build a distributed file system that allows users to upload content to

the network (e.g., static websites, documents, media files, etc.) and access them in a decentralized fashion.

P2P can also be used in distributed computing. These applications allow users to share their machine resources to assist in computing tasks that require high computational power. Seti@Home[1] was a pioneer application that allowed users to share their computational resources to aid researchers in fields such as physics and biology.

Voice over IP(VOIP) applications can also leverage the use of P2P. For instance, in the Skype application, peers in the network assist participants in establishing connections to others, thus working around connectivity problems that usually arise due to the existence of firewalls and NAT boxes [13].

2.1.3 P2P Services

The applications described above rely on services that provide the features necessary to implement their core functionality. In this section, we will discuss some possible reusable services that can be built on top of P2P networks.

2.1.3.1 Gossip Dissemination

P2P networks can be used to support gossip-based dissemination protocols [2, 11]. In a gossip protocol, nodes collaborate to broadcast a message through the network, similarly to how rumors spread in a population. A node, in order to disseminate a message, selects a subset of its neighbors at random, a parameter commonly called fanout, from its partial view and relays the message to them. A node, upon receiving a message for the first time, repeats this same process. A common gossip dissemination strategy is a flooding strategy in which a participant sends the message to every neighbor.

Gossip protocols can use several strategies to broadcast messages [17]. These include, but are not limited to the following:

Eager-Push As soon as a node receives a message for the first time, relays it as soon as possible. This strategy usually has the lowest latency but may create excessive redundant traffic.

Pull Nodes periodically query neighbors for messages they received recently. Once a node learns of a new message that has not received yet, requests the full payload. Nodes need to remember messages received which consumes additional memory.

Lazy Push As soon as a node receives a message for the first time, it only forwards the identifier of the message received. Like in the pull approach, once a node learns of a new message identifier, it requests it to the node that sent the identifier. Similarly to the Pull approach, nodes also need to store messages received.

[1] https://setiathome.berkeley.edu

2.1.3.2 Resource Location

A resource location service allows a participant to gather a set of peers' identifiers that own a given resource. The service is in charge of disseminating the query among peers, to gather the identifiers of participants that hold a resource which matches with the query and return an answer to the participant that issued it [17].

Exact match queries use unique identifiers from resources to locate them in the network. A client in order to issue this type of queries, must know the unique identifier *a priori*. In a file system, these types of queries can be used to find a specific file in the network, given its unique name.

Range queries rely on the interval over the value of some property, to return resources that fall in the provided range. For instance, in the same distributed file system, we might want to find files that have a size between two particular values.

Other type of query that a client might want to perform is keyword queries to find a list of resources associated with a tag or keyword. For instance, in a distributed network, we might want all machines that are running Linux or find files tagged as *music*, *movie*, etc.

2.1.4 P2P Overlays

P2P overlays can be decomposed in two types by considering how the overlay is maintained and created. Each of the categories has a vast literature and a manifold of approaches can be used when creating them [19, 20].

2.1.4.1 Overlay Categories

Unstructured Overlay The first approach is to build an unstructured overlay in which nodes are organized in a random topology. Neighbor sets are formed by creating arbitrary connections between participating nodes. In this type of networks, in order for a participant to locate a resource, a query needs to be disseminated among all peers to ensure that a list of all possible values is returned. These networks are useful when a resource location service needs to support range and keyword queries.

These types of overlay typically are more robust to network failures but can be less efficient when searching for a specific resource in the network.

Examples of unstructured overlays include HyParView [12], Overnesia [14], Cyclon [35], and Scamp [7].

Structured Overlay The second approach is to build a structured overlay to act as a Distributed Hash Table (DHT). These overlays have a more rigid topology that is based on unique identifiers that are assigned to participants. Resources in the system are mapped to a specific peer in the network using its own unique identifier.

In order to locate a resource in the network, a participant will use that unique identifier to find the peer that holds the value associated with it.

While the resource lookup is more efficient in these overlays, a peer needs to know the resource's unique identifier in order to locate it in the network and take advantage of the underlying overlay organized structure.

Examples of structured overlays are Chord [33], a P2P system that organized peers in a ring topology, Kademlia [22] that organizes nodes in a binary tree and Pastry [32] that creates a structured mesh.

In [10], the authors propose the T-Man algorithm to bootstrap any structured topology from an unstructured overlay using a gossip-based protocol. The proposed method is used in [25] to build a ring-like, structured topology from an unstructured one in a logarithmic number of steps which becomes similar to the Chord topology.

A third type of overlay can be created by combining two or more overlays. This creates a hybrid overlay that can take advantage of the multiple benefits offered by different P2P overlay implementations. For example, two structured overlays can be leveraged to improve node's routing information [21] .

2.1.5 Main Challenges

In [28] the authors detail some issues relevant to routing protocols used in structured overlays, which will be the main focus of this work. These challenges are sufficiently generic that may also be applied to the other overlay types. Even though the paper is from 2002, some of the raised questions remain a topic of debate.

2.1.5.1 Routing Hot Spots

When an overlay network receives an unexpected amount of traffic, there may be cases when certain nodes become overloaded creating routing hot spots. For instance, if the overlay uses some type of central peer to route traffic in order to shorten routing, this node may be a bottleneck in the system and fail, disrupting the entire network operation. Even when there is no central peer, nodes may become overloaded with routing traffic to other nodes. As pointed in [28], these hot spots are harder to tackle since a node cannot take a local action to reroute traffic.

Another example is when a participant holds a file that sees its requests spike massively. This node may become overloaded and will not fulfill all requests in a reasonable amount of time. In order to combat this issue, different types of replication mechanisms can be used to spread highly popular resources in the network. Their effectiveness however, depends on identifying such resources as being popular in a timely fashion.

2.1.5.2 Fault Tolerance

An important aspect of P2P is how tolerant the system is to failures when delivering messages or routing traffic. Churn [34], which is defined as a measure of the amount of nodes joining and leaving the system, is one of the main factors in network failures. When a high amount of nodes leaves concurrently, the membership protocol may not be able to keep up and nodes may become unreachable in some cases due all their neighbors disconnecting [23].

Unstructured overlays, due to their flexibility in managing the topology are less susceptible to churn scenarios then structured overlays which have a more rigid topology.

2.1.5.3 Topology Mismatch

While overlay networks are useful as an abstraction of the underlying network, the overlay can also become a major issue in the system efficiency as two nodes in the same physical network may be separated by a large number of hops in the overlay network, creating a topology mismatch between layers. As the network grows, this problem becomes more evident as the number of hops to locate a resource increases with the network size.

In Figure 2.2 is shown a simple example of a possible network represented on the physical level and in Figure 2.3 two possible overlay configurations. While the network represented in 2.3(a) is well configured to suit the topology and achieves minimum cost between two nodes, network 2.3(b) suffers from topology mismatch and paths are poorly optimized.

P2P systems can take into account the underlying topology to bias the neighbor sets [15]. For example, a system may measure the latency between nodes, and choose to only integrate into a given node's neighbor set other nodes that satisfy a specific latency range. A problem created by such a naive solution is that by choosing only certain nodes based on this condition, it is possible that one could create disjoint networks, as nodes would only connect to nodes in the same region.

2.1.5.4 Network Heterogeneity

In a real P2P network, it cannot be assumed that all participants hold the same resource capacity and processing power. While less powerful nodes can become a bottleneck in the network, it is possible to also take advantage of the network's heterogeneity by using more powerful nodes to route more traffic and hold more resources.

Super-peer networks are a class of overlay that leverages the use of more stable and powerful nodes to create more efficient routing protocols. In these networks, super-peers are connected between each other while regular nodes connect to a single super-peer. This solution may lead to problems where super peers fail and all of their connected nodes have to find other super-peers to connect, leading to moments of high instability.

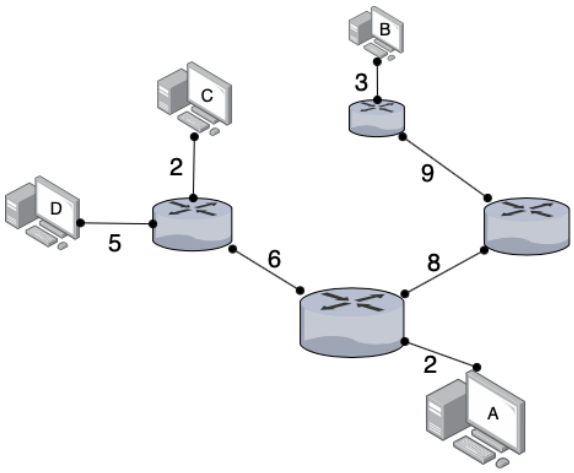
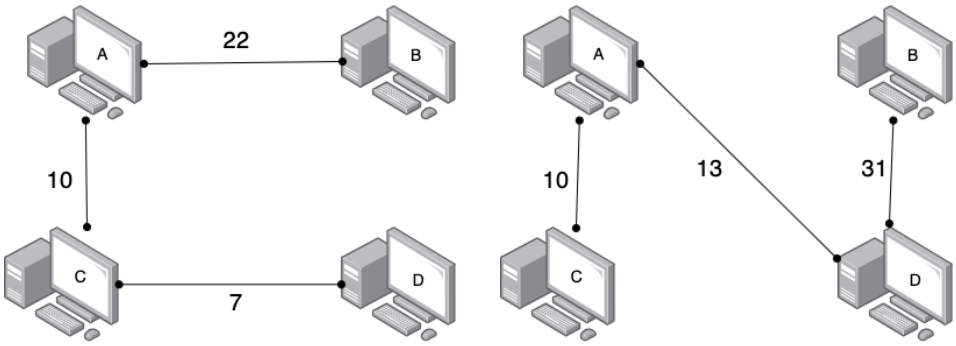


Figure 2.2: Network Topology



(a) Overlay paths are well optimized (b) Topology Mismatch in overlay

Figure 2.3: Two possible overlay configurations

2.1.6 Discussion

In this section, we gave an overview of the concepts necessary to discuss more thoroughly overlay networks architectures and protocols built on top of them. We also discussed some open problems that may arise when developing such systems. In the next section we examine unstructured overlay networks and study some examples from the literature. Even though our work mainly focuses on structured overlay networks, some of the properties that unstructured overlays provide can be used in cooperation to achieve, for instance, higher tolerance to failures.

2.2 Unstructured Overlay Networks

Unstructured overlays are characterized by the random neighbor connections each node establishes in the system. They give more freedom to nodes when choosing neighbors imposing a lower overhead than structured overlays. This flexibility translates to a higher tolerance to churn scenarios, since when a node departs the system it is easier for their

neighbors to find suitable replacements. These overlays are typically used to support gossip dissemination protocols and resource location services that use range or keyword queries.

Each node only maintains a partial view of the network, as containing all members of a large scale network in memory would overload a participant. These partial views can be maintained using different strategies employed by distributed membership protocols:

Cyclic In a cyclic strategy, nodes periodically exchange neighbor information with peers, updating the view each time this operation happens. In this approach, views are updated in stable systems. An example of a usage for this strategy is Cyclon [35].

Reactive In a reactive strategy, nodes act as a response to an event that affects their membership (i.e., nodes joining or leaving). Contrary to cyclic approaches, views are not updated in stable systems. This type of strategy relies on some kind of failure detection mechanism. HyParView [12] uses a reactive strategy to update a part of the neighbor state that each node maintains.

2.2.1 Examples from the Literature

2.2.1.1 Cyclon

Cyclon [35] uses a cyclic strategy to maintain the set encoding a node's neighbors (i.e., the partial view). Each entry in neighbor set has a corresponding age, indicating for how long that particular entry has been created. Cyclon uses this age to choose which peers it will use in a cyclic shuffle operation. In each cycle, a node n will increase the age of all peers in its neighbor set and select the oldest among them to perform the shuffle. Node n will create a *shuffle list* from a random subset extracted from its neighbor list, join itself to the list (with the age of zero) and send it to the oldest node n'.

Node n' upon receiving this list, will create an equivalent *shuffle list* with its own neighbors, send it back to n and perform a merge with its neighbor view. For each entry in the list, it will check if already has an equivalent one and will incorporate into the list the one with less age. Otherwise, it will add the entry to its set if it is not full. Lastly, if none of those conditions holds, it will remove an entry that it will later send with its shuffle list and add this new entry. The procedure is repeated similarly by node n when it received the set of samples from n'.

A joining node n" only needs to put the contact node (i.e., the system's participant that is reached in order to join the overlay) into its partial view and wait for the cyclic timer in order to start filling its neighbor list. Another alternative is to perform a random walk through the network, until a predefined TTL, to find a random node in the network. A node will relay the contact information of n" and decrease the TTL in each hop. Finally, n" will insert the node where the random walk ended in its neighbor list.

Cyclon strategy, in the presence of a static network, continues to exchange neighbors which causes unnecessary communication costs in these network conditions. The interval

between shuffle rounds may, in some cases, too large to timely deal with highly unstable networks (e.g., churn conditions).

2.2.1.2 HyParView

HyParView [12] combines both cyclic and reactive approaches to maintain two distinct partial views. A small active view is maintained using a reactive strategy that contains the actual nodes that will be used to create the overlay. A passive view, maintained by the cyclic strategy, contains a larger list of nodes that is used to replace links in the active view when they fail. Periodically nodes perform a shuffle operation in which they trade a list of nodes in order to update the passive view.

A node n joins the network by establishing a connection to its contact node. The contact node adds n into its active view and propagates a *FORWARD JOIN* message through the network using a random walk with two configurations parameters associated. The *Active Random Walk Length* (ARWL) is the parameter that specifies the message's maximum TTL and *Passive Random Walk Length* (PRWL) specifies at what point the node is inserted into another node's passive view.

A node that receives a *FORWARD JOIN* will add n into its active view if it only contains a single node or if the ARWL reached its end. If the view is full in the latter case, a random node will be evicted from the active view. In the case that PRWL hops have been done by the *FORWARD JOIN*, the node forwarding the message will add the new node into its passive view. Lastly, if the node was not added to the active view in the other case scenarios, the message is relayed to another node.

Entries in a node's p active view are replaced when the established connection fails. In this case, p picks a candidate c from the passive view and tries to establish a connection. If the connection cannot be established, c will be removed from the passive view. Otherwise, p sends a *NEIGHBOR* message to c with a priority level associated, depending on current size of the active view. The message has high priority if p's active view is empty, otherwise is low priority message. Node c always accepts a high priority request by adding p to the active view, even if it needs to evict a node from its view. It only accepts a low priority if it has space left in its list. A node evicted from the active view will be moved to the passive view if the removal was a local or remote decision, and not a failure from the TCP connection.

Node p upon receiving the decision of c, will move it to the active view if c accepted the request. Otherwise, it will select another node from the passive view and try again using the same process.

Periodically a node updates its passive view using a cyclic strategy, similar to the shuffle used in Cyclon [35]. A node n will select a subset of nodes from both its views, add itself to the list and send this subset using a random walk with a TTL to another node. If a node p receives this message it will only perform the shuffle if the TTL reached its end or p's active view only contains one entry. Otherwise, decreases the message's TTL

and relays it to another node. The shuffle operations consist of creating a similar list to the one sent by n and merge p's passive view with n's list after it removes nodes already present in the active and passive view from it. Due to the view's fixed sized, if necessary, p will prioritize the removal of nodes that will be sent to n first in order to free space. By integrating nodes from the active view into this shuffle, the probability of having only active nodes in the passive view increases while nodes that failed will eventually be discarded by all participants.

Experiments made on the original paper [12] show that HyParView, compared to the previously discussed Cyclon, is more reliable in high churn scenarios as it maintains a message delivery reliability of 100% with a node failure level of 90% compared with Cyclon's 0% reliability. HyParView also has a faster healing time as it needs less rounds to reestablish its reliability in highly unstable scenarios.

2.2.1.3 Overnesia

Overnesia [14] explores and expands on the concept of a super-peer network to build its overlay. Super-peer networks minimize the cost of disseminating a query through the network by having a two-tier hierarchical structure: in the bottom level, nodes connect to super peers; in the higher level, the super peers are connected between themselves. Each of these nodes connects to other super-peers forming an overlay and is responsible for maintaining an index of all regular nodes connected to it. While super-peer networks are more efficient in message dissemination, they can become less tolerant to node failures. For example, if a super-peer fails then all its low-level nodes will have to find a replacement super-peer and its index must be updated. Super-peers, if not chosen properly (e.g., more high powered nodes or with lower average latency), can become a network bottleneck.

In order to solve these challenges, Overnesia builds a network of virtual super-peers where each of these is composed of multiple physical nodes which act as replicas. Each of these replica nodes possesses links to different virtual-peers, thus creating more diverse paths increasing the tolerance to failures. Nodes inside a virtual peer (or cluster), maintain a complete view of all other members in the same cluster and keep track of cluster's size.

A new node, when joining the overlay, propagates a join message through the network using a random walk with a TTL. The node joins the cluster where the walk ended or if it finds first a cluster with a size below the desired target. The node will then, establish neighbor relations will all the cluster's members.

Nodes maintain a constant size neighbor set containing members of external clusters. To acquire these neighbors, a participant uses a random walk to find new nodes that also got empty space in their external neighbor set. If the random walk terminates without a finding a suitable peer, the last visited node becomes the new neighbor.

Clusters may need to split due to their increasing size. To achieve this, the node

inside a cluster with the smallest identifier will trigger a split procedure that includes generating two different cluster identifiers and assign each half of its view to each cluster. A node upon receiving this message, will wait a predefined amount of time and if no other division message was sent by other smaller identifier node, it triggers the joining procedure to the new cluster.

A collapse procedure may be used to migrate nodes from a cluster whose size has fallen below a predefined threshold. Nodes will periodically check the cluster's size and trigger a procedure similar to the aforementioned where a new node joins the network . If during this process the cluster grows sufficiently, the node may cancel the join request.

To maintain a consistent state between all nodes in a cluster, an anti-entropy procedure is used periodically where nodes trade their cluster views and external neighbor sets. A node that receives this message will add missing cluster neighbors to its list. Finally, if the node receives two consecutive messages that contains entries referring to a cluster that is also present in its list, it disconnects from that peer.

In the experiments detailed in the original paper [14], Overnesia is able to handle high amounts of churn compared to other unstructured overlays and the Chord DHT [33] which will be discussed further ahead.

2.2.2 Discussion

Unstructured overlays, as mentioned, provide higher resistance to churn scenarios than their structured counterpart. Even though they are random in nature, there are some ways to bias the topology to optimize some efficiency criteria while preserving the original properties [16].

One of the drawbacks of having a random network topology is that when searching for a specific resource (e.g., file), it may be necessary to flood the entire network to find the node storing it. In this use case we might benefit from having nodes connecting in a predictable fashion that allows lookup mechanisms to make efficient decision on their routing path. We are now going to discuss structured overlays designs that approach routing and geographical topologies in different ways which will be the main focus of our work.

2.3 Structured Overlays

Structured Overlays are characterized by their organization using a strict topology enforced during their creation by the careful construction participants' routing tables. To each participant, a uniformly random unique identifier is assigned that defines the position of the node and allows to locate in the network. This normally done by using a consistent hashing function.

Structured Overlays are usually used as DHT (Distributed Hash Tables) where resources are also assigned unique identifiers similarly to nodes. Each participant therefore, is responsible for maintaining a set of keys distributed uniformly across the network promoting load balance. These overlays provide an efficient structure for the use of exact match queries (i.e., queries to locate resources where the identifier is known *a priori*).

Each node only keeps a small number of routing entries to neighbors in comparison with the overall network, and when a node enters or leaves the system, only a small number of nodes are affected by that event.

Due to these characteristics, structured overlays provide a scalable infrastructure to build file sharing applications and distributed file systems where files can be found efficiently using their identifier.

2.3.1 Examples from the Literature

Classifying a specific overlay into a category is not trivial as many features are shared by multiple classes of overlays while other are very specific to a single overlay implementation. In this section we will classify overlays by taking into consideration the way they treat the underlying topology into the lookup protocols and routing tables, similar to what was proposed in [28].

2.3.1.1 Proximity Routing

A Proximity Routing based DHT optimizes the routing path based on the progress towards the looked-up key. This type of routing can be extended by taking into consideration the latency of neighbors when choosing the next hop. This type of overlay does not take network distances into account when constructing routing tables, thus it is possible that a single hop may travel the entire underlying network diameter.

Chord Chord DHT [33] is a P2P network design that organizes nodes in a ring topology where the node's assigned identifier, with m bits, defines its position in the ring. In a basic Chord implementation, each node needs to know only its successor (defined as the first node in a clockwise direction from itself) and predecessor. Nodes are responsible for keys with identifiers in the interval between their identifier and their predecessor identifier.

In a simple implementation, a node performing a lookup will iterate all successors until the correct node is found thus, in the worst case scenario, a lookup will need to transverse the whole network in order to find the value. To improve lookup performance, a Chord node also maintains a routing table with a maximum size of m, containing pointers to distant nodes in order for lookups to take shortcuts. This routing table, also called a finger table contains, at each entry i, the routing information for the next node that succeeds itself by at least 2^{i-1}, thus each hop increases by double the distance. It is important to note that in a sparse network,

multiple entries can point to the same node due to absence of nodes in a large interval. Predecessors are used to support join and leave procedures.

In lookup queries, a node will check the entry in its finger table corresponding to the searched key's interval and route the query to that node. This process continues until the routing finds a node whose successor is responsible for the key's interval, which in turn will return the contact for its successor. In this forwarding mechanism, the number of hops necessary will be $O(\log N)$, being n the total number of participants.

A new node joining the system, will ask its contact node to lookup its finger table and its predecessor. The contact will then, by each entry i, find the corresponding node by looking up the $n + 2^{i-1}$ node being n the new node's own identifier. This process is optimized by also checking if a given entry's node is also the next entry. This optimization reduces completion steps from $O(m \log N)$ to $O(\log^2 N)$ steps. In order to add the new node to already present nodes' finger tables, a Chord node will go backwards through the network m hops and, at each hop, add the node into the i^{th} entry of the closer preceding node to the $n - 2^{i-1}$ identifier, being i the hop number. Lastly, the application layer will need to transfer the resources which became responsibility of the new node from its successor node.

A stabilization protocol runs in the background periodically in each node to guarantee correctness. It consists of a given node n asking its successor n' for its predecessor. If the returned node is not n then this node will become n's successor. The node n' can also change its predecessor to n if its current one is further away than n. To stabilize finger tables, nodes query random entries to find the node that is responsible for that key-space and replace the current entry with the new one if necessary.

When a node n fails, all nodes that contain n in their finger tables must update the entry with n's successor, which is done eventually by the stabilization protocol. Each node maintains a list of its next successors to use as backups in case its current one fails. The predecessor to n will replace it with the first entry closer to it during the stabilization procedure. Nodes that are running lookup queries that encounter a failed node, will have to retry after a timeout in order to wait for the stabilization procedure.

Chord DHT is well balanced in terms of key distribution and provides a logarithmic number of steps to perform lookups and join/leave procedures which are valued attributes which can be used in collaborative distributed file systems. Chord inspired other algorithms that base themselves on its design to offer different types of services and guarantees such as Self-Chord [5] that improves load balancing and the possibility to execute range-queries.

Kademlia Kademlia [22] is currently one of the most popular DHTs and its algorithm is

used in multiple applications. To each node, a m-bit node identifier is assigned that determines a node's position in the network by its shortest unique prefix. Participants are organized in a binary tree that is divided into multiple successive subtrees where a node knows at least one other node in each of the subtrees it does not belong to. Any node that wants to find another peer given its identifier, will only have to successively query the closest node in knows of, until it converges to the target.

Kademlia uses the XOR metric as a notion of distance between nodes. A XOR metric takes two identifiers and the resulting XOR bitwise operation is interpreted as an integer to provide the distance between two nodes. Using this metric, the closest node of a given n will be the node that shares the longest common prefix with it.

Each node keeps a routing information list for each $0 \leq i < 160$ of the identifier bits, containing nodes with distance between 2^i and 2^{i+1}, called k-buckets, ordered by last time seen. The buckets are organized in a binary tree and are identified by the longest common prefix shared by its members. Initially, a node only has a single k-bucket that covers the entire identifier space. When a node n gets to know a new node, it will check if the assigned k-bucket is not full and if that's the case it will simply insert it. Otherwise, if the assigned k-bucket space covers n identifier it will split the bucket into two and then insert the new node into the appropriate one. This splitting happens until a node knows at least a neighbor in all subtrees.

K-buckets provide a Least Recently Used eviction policy. When a node receives a message, it will check if the sending node is already assigned to the appropriate bucket, and if present, it is it will simply move it to the bottom of the list. If it receives a message from an unknown node, and the bucket is full, it will put this node into a replacement cache. When a neighbor becomes unresponsive, it will move a node from cache and replace the unresponsive one. By doing this, Kademlia prioritizes older neighbors which are more likely to stay in the network for even longer periods of time minimizing the k-buckets management overhead. The Least Recently Used policy also provides resistance to denial of service attacks, as a large quantity of attacking nodes will not have an immediate impact on participants' routing tables. Nodes periodically ping entries in buckets that have not received lookup requests recently, in order to keep the bucket list with the minimum of failed entries possible.

Kademlia exposes four Remote Procedure Calls (RPC): *FIND_NODE*, *FIND_VALUE*, *PING* and *STORE*. *FIND_NODE*, given a node identifier, returns contact information for the k closest nodes it knows. *FIND_VALUE* is similar to the first one as it returns the k closest nodes it knows of except if it is holding the value associated with the searched key. In that case, returns the stored valued. A node that receives a *STORE* command stores the key and value contained in the request. *PING* simply probes a node to check its status.

To find a node with a given identifier, an initiator node starts by picking α entries from the closest k-bucket and sends simultaneous *FIND_NODE* calls to those peers. Recursively, the node will inquiry every new α nodes by sending the same RPC. This procedure ends when the node receives responses from all closest nodes. A possible optimization suggested by the authors, is to augment k-bucket's entries with round trip estimates in order to better choose the α nodes. To find a value, the followed procedure is the same, expect it stops when the value is returned. Lastly, the requesting node will cache the resource at the closest node it knows of that didn't return a reply with the resource. XOR's unidirectionality causes lookups to converge along the same path, and as so, these searches are likely to hit cached entries first alleviating hot spots. It is expected that a lookup request will take $O(\log n)$ steps. Cached entries need to have an expiration time in order to avoid accessing outdated date.

A node joining the network will insert the contact node into its k-bucket and perform a node lookup for itself to gather its closest neighbors. The node will refresh all k-buckets with an identifier space further away than its closest neighbor to populate them and to be inserted into other nodes' buckets in order to be reachable.

A client node that wants to store a value in the network, will use the *FIND_NODE* RPC to find the closest possible nodes to the value's key and send to k of them a *STORE* RPC. Nodes storing a key must republish periodically it to compensate for nodes leaving the and new nodes arriving with identifiers closer to the value's key.

Kademlia compared with Chord [33], uses less configuration messages for nodes to learn about each other. Also, by using lookups to spread configuration information makes the network more efficient. As mentioned, the use of K-Buckets also provides some resistance do denial of service attacks. Kademlia implementations appear in file sharing networks such as BitTorrent [3] and distributed file systems such as IPFS [1].

2.3.1.2 Proximity Neighbor Selection

Proximity Neighbor Selection DHTs select nodes to build routing tables based on distance measurements to them. In scheme however, is still possible that a single hop may travel the whole network distance, as a node still has only a partial view of the whole system when filling its routing table.

Kelips The Kelips [8] DHT organizes nodes into a fixed size number of *affinity groups* or clusters. The protocol assigns each node to a cluster, based on its identifier while ensuring that groups are balanced in size.

Each node maintains a partial view of neighbors in the same affinity group, with each entry containing the node's contact information and additional fields such as round trip time and a heartbeat count. In order for nodes to communicate with

peers from other groups, each node also keeps contact information for a few nodes in other affinity groups. In the implementation detailed by the authors, both lists have a preference policy for nodes which are closest in terms of latency.

To replicate routing information in the same *affinity groups*, nodes maintain a partial list of tuples, containing a resource identifier and the node that is currently storing the file. For each tuple, a heartbeat is also associated and if not updated over a specified time, the entry is deemed invalid and removed from the list.

Within a cluster, nodes periodically use a gossip dissemination protocol to update the resource list. In each round of gossip, a node selects a constant number of nodes and sends a batch that contains file tuples and membership entries. Selected nodes in each round are based on which nodes are closer in terms of latency to the initiator node. As heartbeats for contact entries will also need to be sent to another clusters in order to remain valid, a gossiping node will also select contacts in other clusters and send them the information.

Kelips strives to limit bandwidth usage, and as such, gossip messages cannot grow indefinitely. In order to work within this limit, messages can only carry a maximum number of resource entries which are divided equally for old and new ones. For each group, entries are selected at random and if all of slots reserved to new cannot be fully filled, old ones fill what is left.

New nodes joining the system are mapped into the appropriate *affinity groups* and can start to participate in the network once they fill their neighbors' views and resource lists. As the number of contacts each node can have is limited, once a node learns of a new node and its contact list is full, it will prioritize nodes which are closer.

A node that is inserting a resource, will map it to an affinity group and send the resource to the closest node in terms of latency from that affinity group. The node that receives this request, will choose a random node from the affinity group transfer the resource to it. Insertions therefore, have a time and message complexity of $O(1)$. When a node receives a new entry, it gossips about this message for a predefined number of rounds for other cluster's members to know its location. In order to also keep the entry from expiring, the origin node periodically refreshes the entry.

Resource lookups work similarly to the insertion procedure. A node will map the resource identifier to the appropriate affinity group and send a lookup request to the topologically closest node in that group. If the receiving node contains the identifier in its list, will send back its location in the cluster. Similarly to insertions, this lookup method will have a time and message complexity of $O(1)$.

As there is also a possibility that file insertions and lookups fail, there are multiple strategies proposed in order to retry the request. One strategy is to, from the start, send multiple concurrent requests to different nodes. Another viable strategy is for

queries to be propagated through the affinity group until a fixed TTL and in case of insertions, the file is inserted where the TTL expires. Strategies like these have impact on the complexity. For instance, the normal case for insertions in the second example will have a complexity of $O(\log \sqrt{n})$.

Since Kelips nodes keep more information about neighbors in storage as compared with other DHT, it has a higher memory usage. The amount of storage required for each node is $\frac{n}{k} + c \times (k-1) + \frac{F}{k}$ where n is the number of nodes in the system, k the number of clusters, c is the external cluster neighbor entries, and F is the number of files. Authors claim that even with this increased memory usage, the requirements are moderate for medium sized systems. Kelips also has an increased background overhead due to the constant gossip messages. Authors argue that this overhead is acceptable in order to support faster lookups and insertions. Kelips tolerance to failures also benefits from the background overhead due to the constant exchange of neighbors, as nodes that are affected by high churn can recover faster.

Tapestry Tapestry [37] uses a small constant sized routing table to create an overlay mesh to route messages. Tapestry routing works similarly to a longest prefix matching, as at each hop, the number of identifier digits matched is incremented digit by digit (e.g., ***3 → **23 → *123 → 0123) using a multi-level map, called neighbor map, where each level matches the identifier up to the level's position. In each hop i the protocol will look into the ith level of the map and find the entry which corresponds to the digit in that position. This entry's node is the closest neighbor known which ends with the searched prefix. Surrogate routing is used when the entry to be used is empty, which will choose the next best entry option. The routing protocol guarantees that every node can be found in $O(\log_b n)$ steps. For purposes of routing fault-tolerance, each entry in the map is backed by two additional entries that will be used when the main fails. Nodes keep backpointers in order to send heartbeats notifying neighbors that they are still a viable routing path.

When a connection to a node fails and this event is detected by its neighbors, they mark it as invalid and start routing messages through alternatives. As connections can be reestablished, a Tapestry node periodically probes to check if the node is active for a period of time. After this period ends, the neighbor is removed from the map and replaced by a backup node, if one is available.

Tapestry's replication mechanism stores pointers to objects residing on a client's node, and not the content itself. When a client wants to publish an object that it is currently holding, it sends a message containing a copy of the pointer to the object's root node, which is the node that shares the same identifier *id* as the object. As it cannot be assumed that there is a node in the network with the same identifier, the root node will be the node with the closest identifier to it. In order to find this node, a message is routed to that *id* using the method described above and terminates

when a neighbor map is reached that the only available next hop is the node itself. At each hop during this process, nodes cache the pointer to reduce the probability of overloading the root node and to improve fault tolerance. If two different nodes publish the same identifier, nodes storing those pointers will prioritize the one with lowest latency to themselves.

New nodes, after contacting their contact node, will start populating their neighbor map. Iteratively, for each digit i in its identifier, the new node N will route a message to nodes that share the same prefix until at least digit i. Nodes that receive this message, will try to add N to their tables, and send back the level i of their routing table. N will compare, for each entry received, primary and secondary backups and choose the best node based on network distance measurements to integrate into its own routing table.

While periodic republishing of a resource key can serve as a solution to keep pointers up to date, this approach may consume a significant amount of bandwidth. Optionally, when a node leaves the network, it notifies all servers which it has resource pointers for. These server nodes will then use a republish message containing the key, pointer and the departing node's identifier. A new path to the root will be taken as the older is no longer available. In the routing of this new message, when the protocol encounters a node that contained the older pointer, it will replace the entry and continue until the root is reached. Concurrently, in the moment an older pointer is found, this node will use its backpointers to send a message backwards to the older path in order to delete it.

2.3.1.3 Geographic Layout

Geographic Layout DHTs build routing tables that resemble the underlying network topology by grouping closer nodes together. This type of DHT normally employs a hierarchical structure to map the geographical space into the overlay.

Coral Coral [6] organizes nodes in a hierarchical structure of clusters, identified by a unique id, increasing in diameter which are composed of nodes whose round trip between any of them is below a certain threshold. By doing this, Coral tries to organize nodes in a geographical topology where nodes closer to each other belong to the same low level regional cluster, with multiple of these composing higher level clusters and so on. Coral was built to support large scale Content Distribution Networks (CDN) with multiple writers and readers of the same content. Coral in the original implementation described in the paper, is built as a layer on top Chord [33] lookup service although authors argue that other DHTs can be used.

This system implements a *distributed sloppy hash table* or DSHT, optimized to find replicated resources. In the DSHT, a key can have multiple values stored under it, in the form of a list, and when retrieving a resource, only the closer values stored

under a given will be returned, thus sacrificing consistency for frequent fetches. A node that wants to publish content that it stores locally, will try to insert a pointer to the content it in the closest node to the resource's key.

A lookup searching for a specific key, will transverse the overlay until it finds a node that contains a full list for that key or reaches the node whose identifier is close to the key. In the first case, when the list is full, the procedure is to backtrack one hop and store the pointer there. It is important to note that an insertion will occur in every cluster level thus making higher level clusters contain all data that their lower level counterpart possess.

To retrieve a value, lookups follow the hierarchical structure. A node will first try to find the value in its low level cluster by searching for the closest node to the key. If the key is found in that node or in a cached entry in its path, then the protocol halts. Otherwise, the query will jump to the higher level clusters until the searched key is found.

A new node wanting to join the network needs to first find an acceptable low level cluster. This process is done by collecting round trip times for a subset of nodes in a cluster. If no cluster meeting the round trip requirements is found, the node creates a new one with a new identifier. After this process, the node will insert itself into the higher levels using the same process. The join and leave procedures used depend on the underlying protocol used.

When a node learns of a larger cluster which round trip times requirements are acceptable, it will move to the new one while also maintaining its old routing table. This node will still respond to queries from the older cluster, and when responding to requests it will send additional parameters such as the cluster's size and the time of its creation. By doing this, nodes in the older cluster will learn of this new one and create a merge effect where all nodes eventually join the new one. Each Coral node only has a rough approximation of its cluster size. If two clusters are similar in size, nodes will prefer the newest cluster.

Clusters can also split due to the increasing size affecting the round trip threshold. To support this operation, to each cluster a center node is assigned to whom nodes will compare the distance measurements. When splitting, two clusters are created. One is identified by hashing the center's node identifier *cnidA* , while the other is identified by hashing the center's node identifier with the higher order bit flipped *cnidB* . To avoid overloading the center node with measurements, if a node decides that its cluster is no longer acceptable, then it will perform a lookup for *cnidA* which will resolve directly to the cluster's center. If the measurement is acceptable then it will perform a put operation with *cnidA* as the key and its node address as the value, in order for nodes in the older cluster to have the possibility to contact it if they want to move clusters. If the measurement is not acceptable then it will join

the other cluster and perform the same procedure. In the possibility that a node does find both clusters unacceptable, it will generate a new cluster.

2.3.2 Discussion

Structured overlays topology strictness is useful to implement efficient lookup protocols but can also be a disadvantage due to its sensitivity to churn scenarios. When a peer leaves the system, in contrast with unstructured overlays, neighbors' routing entries cannot be replaced randomly needing first to find an acceptable candidate that suits the infrastructure. Due to this intermediate step, in high churn scenarios, the network could become partitioned and the routing infrastructure can therefore fail. In [23], the authors show that under heavy churn conditions, using parameters that balance performance and stabilization overhead, Chord successful lookups can fall below 30% when the system membership changes by 1,5% every second in a network with 1000 peers and a successor backup list of 8. Kademlia on the other hand, under the same conditions, even when using an α of 1 (asynchronous parallel requests), does not fall below 60% and when using an α of 5, always stays above 90% of success. Tapestry, as pointed in [37], is also unsuitable for networks where high churn scenarios may happen as routing paths may become unavailable.

Kelips fixed sized clusters make the system less scalable as the overlay size become larger the amount of information each node has to maintain grows significantly. The system also does not take into account the popularity of resources when caching them, as all resources will be cached the same way in the group.

Coral DSHT while providing a geographical notion using hierarchical clusters, may impose a high overhead on nodes switching between clusters as one needs to perform the join protocol used by the lookup service and the procedure to switch clusters. Coral also needs to balance two factors: as the number of nodes in a single DSHT level grow the DHST capacity increases and the miss ratio decreases while at the same time, the speed of lookups in the lower levels tends to grow in these conditions. The number of levels used is also a variable which can impact the performance of Coral.

As we discussed, structured overlays are often less suitable for highly dynamic networks as the overhead needed to reconfigure paths is high. It would be useful to combine some properties of unstructured overlays (e.g., tolerance to a high number of failures) with the logarithmic lookup times of structured overlays. Next, we study this possibility in the context of hybrid overlays.

2.4 Hybrid Overlays

Hybrid overlays are a third type of overlay that combines aspects of structured and unstructured overlays to achieve different and specific goals (e.g., better routing performance, more stability under churn, etc.). Hybrid overlays can, for example, rely on the

unstructured overlay to find resources that are highly replicated in the network while leveraging the efficient lookup protocol of a structured overlay to find less replicated files in file sharing networks [18].

2.4.1 Example for the Literature

2.4.1.1 Rollerchain

Rollerchain [27] is a DHT that uses Chord [33] structured, ring-like topology to achieve efficient lookups and to assign resources to each node. In order to perform the data replication procedure and to maintain the overlay connected it combines Overnesia [14] virtual peer concept.

As mentioned previously, the overlay is composed of virtual nodes which are made of multiple real nodes that replicate data among themselves. All real nodes can answer queries that target their cluster improving load balancing and increase availability. Virtual nodes in the overlay form a ring structure but in contrast to Chord, node identifiers are assigned to achieve even distribution of data across all nodes. A virtual node similarly to a Chord node, has routing information about its closest successor and a finger table with shortcuts to other nodes exponentially further away in the network. Furthermore, clusters can merge together if one of them is below the minimum stable or divide if grows to an unacceptable large size. When one of these scenarios happens, the protocol needs to ensure that each cluster maintains the correct set of keys. To facilitate these procedures a cluster divides into two consecutive virtual peers and merges only with its successor.

To connect two virtual nodes, the protocol creates links between all real nodes from both clusters. Each node maintains a subset of links to the other cluster's members, assuring that if a node fails, the connections are still maintained by the others. In highly dynamics systems, multiple nodes may join or leave, creating an unbalance in link distribution. To re-balance links, the cluster representative (i.e., node with the smallest identifier) is able to activate a procedure that re-distributes connections.

In the experimental evaluation conducted in the original paper [27], the overlay management is higher than compared to Chord's. This is expected as Rollerchain has to maintain two types of overlay. When comparing load balancing results in scenarios where keys are distributed uniformly and other where keys are skewed to a specific part of the identifier space, Rollerchain shows a better balance in both scenarios as its protocol adapts to the keys' distribution.

2.4.2 Discussion

In [21], the authors experiment with building a P2P overlay using Pastry DHT [32] and an unstructured overlay where peers maintain connections to other participants based on similar interests in resources. In their experiments, combining the two approaches led to a reduction in overall overlay maintenance without no performance drawbacks.

This is explained by the two overlays complementing each other by mutually supplying neighbors.

The research on P2P networks that leverage multiple overlays have not received as much attention to date as the other types [20]. Hybrid overlays need to be carefully designed as it may lead only to higher maintenance costs and no returns in performance benefits.

Next, we discuss a specific case study in distributed file systems that will be the main focus of this thesis. It is based around a single structured overlay and there is some arguing that it may benefit from the use of hybrid or a more complex structured overlay.

2.5 Case Study: IPFS

The InterPlanetary File System (IPFS) is a community driven peer-to-peer distributed file system designed by Protocol Labs, that seeks to connect all computing devices with the same system of files [1]. IPFS aims at creating a decentralized web, censorship-resistant, where a member can easily publish data (e.g., documents, photos, websites) and be accessed by anyone. At its core, IPFS backbone is built around a Kademlia DHT [22] based implementation, which we will detail further ahead.

We are now going to briefly introduce the main ideas behind IPFS's architecture focusing on the routing and publishing protocols as detailed in the original paper [1] and in [9].

2.5.1 Architecture

When participants are instantiated for the first time, their identifier is created by a cryptographic hash of their public key. To provide some flexibility in creating hashes, IPFS supports multiple hashing functions by adding a header to identifiers specifying the function and parameters used. In the network layer, similarly to identifiers, a participant can explicitly announce which protocols it can use to communicate.

Data in IPFS is modeled using Merkle Directed Acyclic Graphs (DAG). In a Merkle DAG, each node has a content identifier (CID) that results from the hashing of its content (e.g., payload or children node identifiers). As so, data represented in a DAG is immutable. If data changes in the graph, a new hash needs to be computed resulting in an entirely different object [29]. Another feature of Merkle DAGs is that two different graphs, sharing two similar files, can reference the same data node which can be useful when transferring different versions of the same data. More concretely, in IPFS there are four types of file: A Blob, that represents a single piece of data; a List, that can contain multiple blob or other lists that represents a large file that needed to be dived into multiple blocks; a Tree, that can contain references to all other data types and represents a directory; Commits contain versions of objects. An example of an object is illustrated in Figure 2.4.

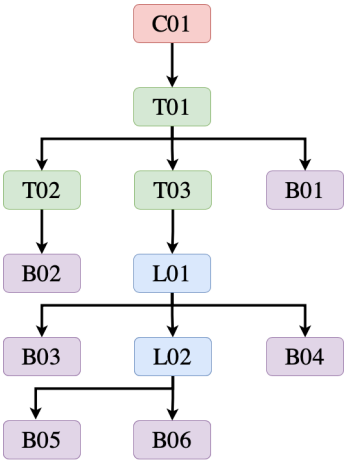


Figure 2.4: Object architecture example in IPFS

Due to the hashes' immutability, if data changes its address also changes. This property is useful as it can be leveraged to verify the integrity of an object, but it is not ideal when a file needs to be updated frequently, as the files need to be published again changing its location making difficult the task of file sharing. IPFS solution to this challenge is to create a level of indirection with the Interplanetary Naming System (IPNS) [1]. The naming system allows a participant to publish content under the hash of its public key. This hash will link to a record pointing to the actual file's hash stored under it. Now, a user that want to access content published by another user, will only have to access the IPNS link to redirected it to the more current data.

The overlay structure of IPFS is built around a Kademlia based implementation. Participants can choose to not participate in the DHT, only retrieving content by directly contacting other members. IFPS routing protocol maintains most of the characteristics detailed in the original paper [22] and previously in Section 2.3.1.1. IPFS's Kademlia implementation optimizes the creation of K-Buckets by reducing it to only the necessary minimum in order to a node not to store mostly empty buckets, as nodes with identifiers very similar (i.e., with a longest common prefix) are probabilistic more difficult to find. Another optimization can be encountered in the eviction policy of K-Buckets, as nodes do not ping entries to check if the connection is still available in order to replace them with a new one. Instead, the new node is rejected and only evicts an entry when the connection is terminated. IPFS also does not maintain the replacement buffer suggested by the original authors of Kademlia. Another important change is in the way data is stored. When a participant publishes a file, it only stores pointers to its local file in network. A file is only replicated once a node retrieves it actively.

2.5.1.1 Bitswap protocol

To transfer data between nodes, IPFS uses a protocol named Bitswap [29] based on BitTorrent algorithm for data transferring [3]. A node, when starting a new session of this

protocol creates a *want list* of root identifiers for files that its currently searching for. Bitswap architecture is represented in Figure 2.5. The ledger is used to check if the node can serve a request to other node by checking its local store, or blockstore, and maintains the status of other nodes' requests. The session manager is responsible for handling each protocol session and can use different systems for looking up content. The connection manager keeps track of all current connections to neighbors and is responsible for the message exchange between peers.

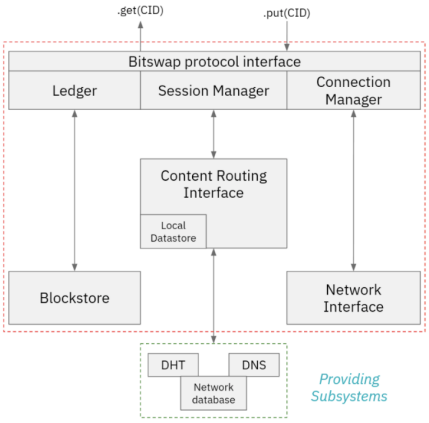


Figure 2.5: Bitswap architecture (adapted from [29])

In the baseline implementation, a node initiates the protocol by broadcasting a *WANT-HAVE* message to all connected peers with the current *want list*. A node that receives this message, will add it to the ledger, and respond back with a *HAVE* if it possesses any of the items or a *IDONT-HAVE* otherwise. The initiator, upon receiving a *HAVE*, will only forward subsequent requests to these nodes as the probability of having the rest of file's blocks is higher. The initiator to request the next block, sends a *WANT-BLOCK* message containing the item's CID to all nodes that confirmed having it.

Upon receiving the first block from the other node, the initiator will use the pointers retrieved from the root's block to request the child blocks iteratively until the full DAG is requested and downloaded. To minimize the reception of duplicate blocks from nodes, once it received a certain block, the initiator can cancel a concurrent request by forwarding a *CANCEL* message. In the possibility that none of the current peers possesses a requested data item, the initiator needs to use a lookup service (e.g., Kademlia DHT, DNS, or a connected database) to find the content location in the network and to create connections to new peers.

2.5.2 Routing at Scale

This thesis focuses on scaling problems of IPFS as detailed in [31]. The authors explain how the use of a single DHT layer for looking up content, as the network grows, is expected to become a bottleneck in the delivery times. They highlight that a complete

solution should be scalable enough to handle efficient routing with millions of active nodes while also reducing the time needed for lookups to return a result. The solution should be able to guarantee that all content is reachable by routing at all times, even in high churn scenarios. The system should also be able to distribute the load with high traffic demand.

To solve this problem, we plan to explore the use of a multi-layer DHT as starting point for devising a solution. The DHT design should try to mitigate the topology mismatch (detailed in Section 2.1.5.3) by topologically embedding it, where geographically closer nodes tend to participate in the same DHT layer.

2.6 Summary

In this chapter we studied the fundamentals of P2P networks and the motivations to leverage its uses in distributed environments. Then, we discussed the multiple types of P2P overlays.

First, we studied unstructured overlays and their usefulness in highly dynamic scenarios due to the low overhead they impose in the linking process between neighbors. We moved on to the study of structured overlays and seen how their efficient lookup protocol may be useful in distributed file systems. Due to their strictness, these overlays may suffer in high churn conditions unlike their unstructured counterparts. This motivated us to study hybrid overlays that can combine multiple networks to improve lookup performance while not creating intolerable maintenance costs.

Lastly, we focused on the particular use case for structured overlays of IPFS, a distributed file system that due to its use of only a single leveled DHT may face scalability issues due to its current continuous growth.

In the next chapter, we briefly discuss how we plan to tackle these challenges by creating a novel DHT.

Chapter 3

Proposed Work

As we have discussed in the last chapter, designing a DHT is not a trivial task as it comes with many associated challenges in real world scenarios, namely susceptibility to churn, participants' reachability, topology mismatch, etc. In this work we are proposing a novel Multi Layered DHT architecture that leverages the use of network locality combined with unstructured overlay properties to offer unprecedented scaling and fault tolerance.

This chapter begins by describing the proposed architecture for the Multi Level DHT and the associated routing protocol (Section 3.1). We continue by detailing the planning for the experimental evaluation (Section 3.2). Finally, we describe the work plan in its entirety (Section 3.3).

3.1 Multi Level DHT

Two of the main objectives for our design is to provide fast lookup times and give guarantees that participants are reachable in the presence of high rates of churn while not incurring in excessive management overhead. We are going to approach the first objective by creating multiple hierarchically organized levels of increasingly larger diameter that take into account network proximity similarly to topology created by Coral [6]. The challenge of creating accurate levels that reflect the network topology, is in the procedure that correctly assigns participants to each of the levels.

Our envisioned solution will be supported by the creation of identifiers that represent the participants' geographical position. In order to achieve this, the starting point will be to use a table that contains prefixes for the first identifier bits that are assigned based on the participant's IP address. With this scheme, we can create multiple levels based on the participants' country, state or even city that will allow the protocol to minimize lookup times for content that is stored nearby.

The second objective, related to the reachability of participants in dynamic environments, will be addressed by combining gossip-based mechanisms and a secondary unstructured overlay to reduce the dependency of the DHT topology from the distributed protocol that manages the topology itself. As we discussed in the previous chapter, one of the main benefits from using unstructured overlays is in their high tolerance to churn and failures. We can also leverage the use of unstructured overlay in the lower levels of the DHT to create clusters of nodes that replicate data, share load similarly to the approach used in Rollerchain [27], and provide redundancy to the established links as lookups might hit connectivity barriers like NAT.

Several aspects of this novel design are topics that have to be tackled by additional closer research and optimization. As mentioned, the scheme that maps participants to levels using a prefix table can be seen as a starting point and is subjected to changes in later optimizations. The other relevant aspect that requires further research is on how to build the levels' hierarchy itself. We have been exploring and discussing multiple alternatives, but it is still not clear the best way to leverage this design. Furthermore, the number of levels necessary to provide adequate robustness and efficiency is also a parameter that will need to be properly studied and evaluated. Further optimizations to the design might include but are not limited to, caching policies and prefetching for content that the participant is likely to request in the near future.

3.2 Evaluation Planning

We plan to evaluate our work in two main ways:

- Firstly, we will conduct an experimental evaluation of the proposed solution through simulation comparing multiple state of the art alternatives (e.g., Kademlia [22], Chord [33], etc.) with a variable system size, different churn rates and workloads (i.e., number of objects stored in the overlay, a varying number of queries and different popularities for the stored objects). In order to realistically simulate large scale systems, a novel simulator based on PeerSim [24] is being concurrently developed by other members of the NOVA LINCS IPFS project that allows to distribute the simulation across different machines unlike other state of the art simulators. This evaluation will also serve to validate the algorithm correctness and to ensure that the desired properties can be reached in a decentralized way with any type of global coordination.

- In a second phase, we will evaluate the performance of a fully functional prototype with realistic workloads. We are going to use a large scale cluster, namely the Grid5000 platform[1] that has machines distributed across France. In each cluster node we plan to run more than one process (i.e., participant) to upscale the scale of

[1] https://grid5000.fr

the evaluation. To make that possible, we will need to manipulate bandwidth and latency as to capture the system's behavior in real case scenarios. As far as we know, an experimental evaluation at this scale is unprecedented which will provide great insight on how these overlays scale to a large number of participants.

3.2.1 Relevant Metrics

During the experimental evaluation, we will gather relevant metrics that will be used to capture the effectiveness and performance of our proposed design. These metrics include but are not limited to, the following:

Latency Measures the time between the departure of a message from a participant and its arrival on the desired destination. It will be useful in observing if the proposed Multi Level DHT is routing messages correctly and efficiently to nearby participants first.

Success Rate Measures the percentage of messages that are correctly delivered. This metric can be useful in measuring how well the solution behaves under churn scenarios and to understand if, at any point, the overlay becomes disconnected making the routing of some queries impossible.

Induced Load Measures the load (i.e., CPU and memory usage) that nodes are subjected to during their participation in the overlay. With this metric, we can observe for instance, how the solution behaves when faced with a lookup surge for a particular resource.

Storage Load As our work is closely tied to a distributed storage system, this metric will allow us to measure how well the caching and replication mechanism is coping with the application's demand.

3.3 Work Plan

In this section we present the planning for the rest of our work. The plan has a duration of 7 months which is detailed in Table 3.1 and the Gantt graph in Figure 3.1. It is important to note that this timeline is subject to change, as some of the stages include activities that it is hard to estimate the amount of time required to complete.

Preliminary Solution We organize the solution's design and prototyping into two main phases. The first solution focuses on devising an algorithm to build the overlay's topology and the creation of an accurate method to assign identifiers to participants. After the preliminary design is established, we will start implementing a solution that reflects the chosen design. At the end of this phase, we will validate the solution using simulation, as previously mentioned.

Final Solution After the first solution is validated, we will move on to the design and implementation of a full prototype that includes all the planed optimizations. As in the first stage, this solution will also need to be validated using simulation.

Final Evaluation In this phase, we will use the mentioned Grid5000 cluster to perform an extended and thorough evaluation to understand how the solution behaves in a scenario close to reality. It is likely, that during this task, the solution's implementation might suffer changes due to the obtained preliminary results.

Writing This phase is divided into two tasks. The first one is the writing of the dissertation that marks the end of this work. The second consists of a paper, to be submitted to an international conference, reporting the design and evaluation of our work.

Table 3.1: Work Plan Schedule

Task	Start Date	End Date	Weeks
Multi-Level DHT: Preliminary Solution Design Implementation Evaluation	1st March	25th April	7
Multi-Level DHT: Final Solution Design Implementation Evaluation	26th April	4th July	10
Final Evaluation	5th July	8th August	5
Writing Dissertation Paper	9th August	30th September	6

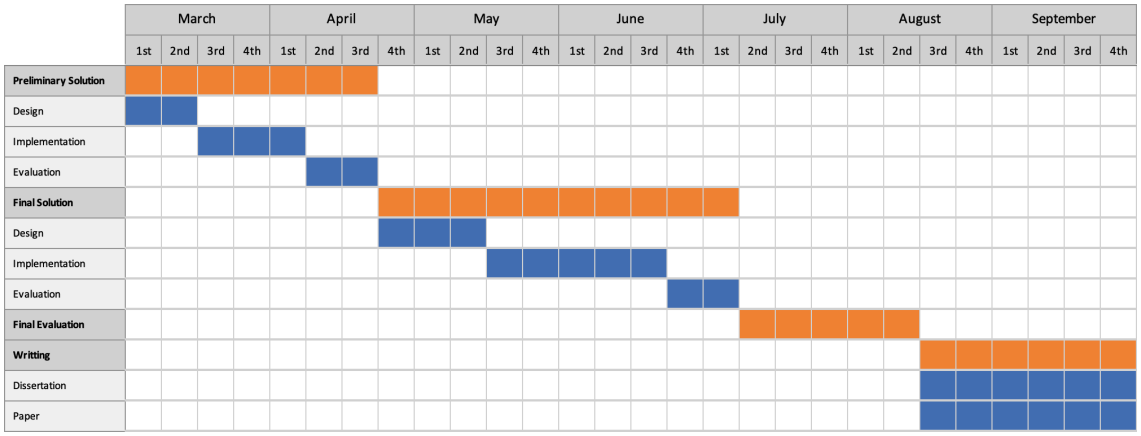


Figure 3.1: Gantt Chart for the proposed work plan

Bibliography

[1] J. Benet. "IPFS - Content Addressed, Versioned, P2P File System". In: Draft 3 (2014). arXiv: 1407.3561. URL: http://arxiv.org/abs/1407.3561.

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

[3] B. Cohen. *The BitTorrent Protocol Specification*. 2008. URL: https://www.bittorrent.org/beps/bep_0003.html.

[4] F. Dabek et al. "Vivaldi: A decentralized network coordinate system". In: *Computer Communication Review* 34.4 (2004), pp. 15–26. ISSN: 01464833. DOI: 10.1145/1030194.1015471.

[5] A. Forestiero et al. "Self-Chord: A Bio-inspired P2P framework for self-organizing distributed systems". In: *IEEE/ACM Transactions on Networking* 18.5 (2010), pp. 1651–1664. ISSN: 10636692. DOI: 10.1109/TNET.2010.2046745.

[6] M. J. Freedman and D. Mazières. "Sloppy hashing and self-organizing clusters". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2735 (2003), pp. 45–55. ISSN: 16113349. DOI: 10.1007/978-3-540-45172-3_4.

[7] A. J. Ganesh, A. M. Kermarrec, and L. Massoulié. "Scamp: Peer-to-peer lightweight membership service for large-scale group communication". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2001. ISBN: 3540428240. DOI: 10.1007/3-540-45546-9_4.

[8] I. Gupta et al. "Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2735.1 (2003), pp. 160–169. ISSN: 16113349. DOI: 10.1007/978-3-540-45172-3_15.

[9] S. Henningsen et al. "Mapping the Interplanetary Filesystem". In: *IFIP Networking 2020 Conference and Workshops, Networking 2020* (2020), pp. 289–297. arXiv: 2002.07747.

[10] M. Jelasity, A. Montresor, and O. Babaoglu. "T-Man: Gossip-based fast overlay topology construction". In: *Computer Networks* 53.13 (2009), pp. 2321–2339. ISSN: 13891286. DOI: 10.1016/j.comnet.2009.03.013. URL: http://dx.doi.org/10.1016/j.comnet.2009.03.013.

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

[12] J. Leitão, J. Pereira, and L. Rodrigues. "HyParView: A membership protocol for reliable gossip-based broadcast". In: *Proceedings of the International Conference on Dependable Systems and Networks* (2007), pp. 419–428. DOI: 10.1109/DSN.2007.56.

[13] J. Leitão, R. van Renesse, and L. Rodrigues. "Balancing gossip exchanges in networks with firewalls". In: *Proceedings of the 9th International Workshop on Peer-to-Peer Systems, IPTPS 2010* (2010), pp. 1–8.

[14] J. Leitão and L. Rodrigues. "Overnesia: A resilient overlay network for virtual super-peers". In: *Proceedings of the IEEE Symposium on Reliable Distributed Systems* 2014-January (2014), pp. 281–290. ISSN: 10609857. DOI: 10.1109/SRDS.2014.40.

[15] J. Leitão et al. "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: 10459219. DOI: 10.1109/TPDS.2012.29.

[16] J. Leitão et al. "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: 10459219. DOI: 10.1109/TPDS.2012.29.

[17] J. Leitão. "Topology Management for Unstructured Overlay Networks". PHD Thesis. Instituto Superior Técnico, 2012.

[18] B. T. Loo et al. "The Case for a Hybrid P2P Search Infrastructure". In: *Peer-to-Peer Systems III*. Ed. by G. M. Voelker and S. Shenker. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 141–150. ISBN: 978-3-540-30183-7.

[19] E. K. Lua et al. *A survey and comparison of peer-to-peer overlay network schemes*. 2005. DOI: 10.1109/COMST.2005.1610546.

[20] A. Malatras. "State-of-the-art survey on P2P overlay networks in pervasive computing environments". In: *Journal of Network and Computer Applications* 55 (2015), pp. 1–23. ISSN: 10958592. DOI: 10.1016/j.jnca.2015.04.014. URL: http://dx.doi.org/10.1016/j.jnca.2015.04.014.

[21] B. Maniymaran, M. Bertier, and A. M. Kermarrec. "Build one, get one free: Leveraging the coexistence of multiple P2P overlay networks". In: *Proceedings - International Conference on Distributed Computing Systems* (2007). DOI: 10.1109/ICDCS.2007.88.

[22] P. Maymounkov and D. Mazières. "Kademlia: A peer-to-peer information system based on the XOR metric". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2429 (2002), pp. 53–65. ISSN: 16113349. DOI: 10.1007/3-540-45748-8_5.

[23] A. G. Medrano-Chávez, E. Pérez-Cortés, and M. Lopez-Guerrero. "A performance comparison of Chord and Kademlia DHTs in high churn scenarios". In: *Peer-to-Peer Networking and Applications* 8.5 (2015), pp. 807–821. ISSN: 19366450. DOI: 10.1007/s12083-014-0294-y.

[24] A. Montresor and M. Jelasity. "PeerSim: A Scalable P2P Simulator". In: *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*. Seattle, WA, Sept. 2009, pp. 99–100.

[25] A. Montresor, M. Jelasity, and O. Babaoglu. "Chord on demand". In: *Proceedings - Fifth IEEE International Conference on Peer-to-Peer Computing, P2P 2005*. 2005. ISBN: 0769523765. DOI: 10.1109/P2P.2005.4.

[26] *Multi-Level DHT Design and Evaluation*. 2020. URL: https://github.com/protocol/research-RFPs/blob/master/RFPs/rfp-7-MLDHT.md.

[27] J. Paiva, J. Leitao, and L. Rodrigues. "Rollerchain: A DHT for efficient replication". In: *Proceedings - IEEE 12th International Symposium on Network Computing and Applications, NCA 2013* (2013), pp. 17–24. DOI: 10.1109/NCA.2013.29.

[28] S. Ratnasamy, I. Stoica, and S. Shenker. "Routing algorithms for DHTs: Some open questions". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2429 (2002), pp. 45–52. ISSN: 16113349. DOI: 10.1007/3-540-45748-8_4.

[29] A. de la Rocha, D. Dias, and P. Yiannis. "Accelerating Content Routing with Bitswap: A multi-path file transfer protocol in IPFS and Filecoin". 2021.

[30] R. Rodrigues and P. Druschel. "Peer-to-peer systems". In: *Communications of the ACM* 53.10 (2010), pp. 72–82. ISSN: 00010782. DOI: 10.1145/1831407.1831427.

[31] *Routing at Scale*. 2020. URL: https://github.com/protocol/ResNetLab/blob/master/OPEN_PROBLEMS/ROUTING_AT_SCALE.md.

[32] A. Rowstron and P. Druschel. "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2001. ISBN: 3540428003. DOI: 10.1007/3-540-45518-3_18.

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

[34] D. Stutzbach and R. Rejaie. "Understanding churn in peer-to-peer networks". In: *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. 2006, pp. 189–202.

[35] S. Voulgaris, D. Gavidia, and M. Van Steen. "CYCLON: Inexpensive membership management for unstructured P2P overlays". In: *Journal of Network and Systems Management* 13.2 (2005), pp. 197–217. ISSN: 10647570. DOI: 10.1007/s10922-005-4441-x.

[36] B. Y. Zhao et al. "Tapestry: A resilient global-scale overlay for service deployment". In: *IEEE Journal on Selected Areas in Communications* 22.1 (2004), pp. 41–53. ISSN: 07338716. DOI: 10.1109/JSAC.2003.818784.

[37] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. "Tapestry : An Infrastructure for Fault-tolerant Wide-area Location and Routing". In: *Science* 74.April (2001), p. 46. ISSN: 01464833. arXiv: CSD-01-1141 [UCB]. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.24.7439{\&}rep=rep1{\&}type=pdf.